UNIVERSIDAD
DE MÁLAGA

LENGUAJES Y
CIENCIAS DE LA
COMPUTACIÓN
UNIVERSIDAD DE MÁLAGA

TESIS DOCTORAL

Metaheurísticas e Ingeniería del Software

Autor
José Francisco Chicano García

Director
Dr. Enrique Alba Torres

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

Julio de 2007

El Dr. **Enrique Alba Torres**, Titular de Universidad del Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga,

Certifica

que D. **José Francisco Chicano García**, Ingeniero en Informática por la Universidad de Málaga, ha realizado en el Departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga, bajo su dirección, el trabajo de investigación correspondiente a su Tesis Doctoral titulada

Metaheurísticas e Ingeniería del Software

Revisado el presente trabajo, estimo que puede ser presentado al tribunal que ha de juzgarlo, y autorizo la presentación de esta Tesis Doctoral en la Universidad de Málaga.

En Málaga, Julio de 2007

Firmado: Dr. Enrique Alba Torres
Titular de Universidad
Dpto. de Lenguajes y Ciencias de la Computación
Universidad de Málaga

Agradecimientos

La realización de esta tesis doctoral ha sido posible gracias, en parte, a muchas personas e instituciones que han contribuido de una forma u otra durante todos estos años. Las primeras palabras de agradecimiento son para Enrique, quien me ha mostrado el maravilloso mundo de la investigación y me ha guiado por él magistralmente. Gracias a su constante apoyo, paciencia y dedicación, ha sido posible que este volumen exista aquí y ahora.

No puedo dejar de mencionar a mis compañeros de laboratorio, que siempre me han ofrecido su ayuda, especialmente cuando la he necesitado. Debo agradecer también a Antonio J. Nebro y Juan Miguel Molina sus enseñanzas y sugerencias.

Dedico un muy especial agradecimiento a Eva, una de las personas que más ha "sufrido" esta tesis, por su comprensión, su ayuda y su incondicional apoyo. Agradezco también el apoyo de mi familia, que desde el primer momento confió en mí y me ayudó en todo lo posible.

Finalmente, me gustaría dedicar un especial reconocimiento a la Junta de Andalucía, por la confianza que mostró al concederme una beca para la formación de doctores con la cual me fue posible iniciar este camino.

¿Por qué esta magnífica tecnología científica,
que ahorra trabajo y nos hace la vida mas fácil,
nos aporta tan poca felicidad?
La repuesta es esta, simplemente: porque aún
no hemos aprendido a usarla con tino.

Albert Einstein (1879-1955)

Índice general

Capítulo 1

Introducción

1.1. Planteamiento

El diseño de algoritmos cada vez más eficientes para resolver problemas complejos (tanto de optimización como de búsqueda) ha sido tradicionalmente uno de los aspectos más importantes de la investigación en Informática. El objetivo perseguido en este campo es, fundamentalmente, el desarrollo de nuevos métodos capaces de resolver los mencionados problemas complejos con el menor esfuerzo computacional posible, mejorando así a los algoritmos existentes. En consecuencia, esto no sólo permite afrontar problemas actuales de forma más eficiente, sino también tareas vedadas en el pasado debido a su alto coste computacional. En este contexto, la actividad investigadora tanto en algoritmos exactos como en heurísticos *ad hoc* y metaheurísticos para resolver problemas complejos de optimización está creciendo de forma evidente en estos días. La razón es que continuamente se están afrontando nuevos problemas de ingeniería, mientras que, al mismo tiempo, cada vez se dispone de mejores recursos computacionales, como nuevos tipos de ordenadores, redes, y entornos como Internet.

La principal ventaja de la utilización de algoritmos exactos es que garantizan encontrar el óptimo global de cualquier problema, pero tienen el grave inconveniente de que en problemas reales (que suelen ser NP-duros en la mayoría de los casos) su tiempo de ejecución crece de forma exponencial con el tamaño del problema. En cambio, los algoritmos heurísticos *ad hoc* son normalmente bastante rápidos, pero la calidad de las soluciones encontradas está habitualmente lejos de ser óptima. Otro inconveniente de los heurísticos *ad hoc* es que no son fáciles de definir en determinados problemas. Las metaheurísticas[1] ofrecen un equilibrio adecuado entre ambos extremos: son métodos genéricos que ofrecen soluciones de buena calidad (el óptimo global en muchos casos) en un tiempo moderado.

La Ingeniería del Software, a pesar de su corta edad, es en la actualidad una importante fuente de problemas de optimización. Los ingenieros y gestores de proyectos software se enfrentan diariamente a diversos problemas de optimización en los que las técnicas exactas no tienen cabida por el corto intervalo de tiempo en que se requiere una respuesta. Cabe la posibilidad, por tanto, de aplicar a estos problemas algoritmos metaheurísticos que ofrezcan al ingeniero una solución de cierta calidad en un breve periodo de tiempo: un compromiso entre calidad de la solución y rapidez.

[1] Algunos autores consideran los algoritmos metaheurísticos una subclase de los algoritmos heurísticos. Por ese motivo usamos el término "heurístico *ad hoc*" para referirnos a los algoritmos aproximados pensados específicamente para un problema concreto.

Es en este contexto donde se enmarca esta tesis doctoral. Nos proponemos aplicar técnicas metaheurísticas a problemas de optimización surgidos en el seno de la Ingeniería del Software, analizando distintas posibilidades para sacar el máximo partido a dichas técnicas y ofrecer así soluciones de gran calidad con recursos computacionales al alcance de cualquier institución. Los problemas abordados son concretamente la planificación de proyectos software, la generación automática de casos de prueba y la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes.

Además, estas metaheurísticas han sido diseñadas e implementadas utilizando, a su vez, técnicas y herramientas propias de la Ingeniería del Software con el objetivo de crear software de calidad. De esta forma, cerramos el círculo: Ingeniería del Software para desarrollar herramientas que resolverán problemas de Ingeniería del Software.

1.2. Objetivos y fases

Los objetivos principales de esta tesis doctoral son: aplicar técnicas metaheurísticas a problemas de optimización de Ingeniería del Software, analizar los resultados para comprender el comportamiento de estos algoritmos y proponer nuevos métodos para resolver los problemas de manera más eficaz y eficiente. Estos objetivos globales han sido descompuestos en objetivos parciales más concretos:

- Identificación de problemas de optimización en Ingeniería del Software.
- Descripción y formalización de varios de estos problemas de optimización.
- Descripción y formalización de las técnicas metaheurísticas propuestas.
- Aplicación de técnicas metaheurísticas a problemas de optimización de la Ingeniería del Software y análisis de resultados.

Para conseguir estos objetivos se siguen las fases resumidas en la Figura 1.1. Inicialmente revisamos la literatura en busca de problemas de optimización en Ingeniería del Software. Posteriormente, seleccionamos tres problemas que, de acuerdo a la revisión bibliográfica realizada, representan a los dominios de mayor interés en el campo de la optimización de problemas de Ingeniería del Software. Estos problemas son la planificación de proyectos software, la generación automática de casos de prueba y la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. En el siguiente paso estudiamos las técnicas metaheurísticas y seleccionamos para cada problema aquéllas que puedan resultar más interesantes desde el punto de vista de la comunidad científica. Las técnicas seleccionadas son los algoritmos evolutivos, la optimización basada en cúmulos de partículas y la optimización basada en colonias de hormigas. Por último, escogidos los problemas y las técnicas, aplicamos los algoritmos metaheurísticos a los problemas de optimización y analizamos los resultados obtenidos.

1.3. Contribuciones

En este apartado se listan las contribuciones de la presente tesis. De forma esquemática, estas contribuciones se pueden resumir como sigue:

- Modelo formal de metaheurísticas secuenciales que extiende al propuesto por Gabriel Luque en su tesis doctoral [183] para hacerlo más operativo desde un punto de vista teórico.

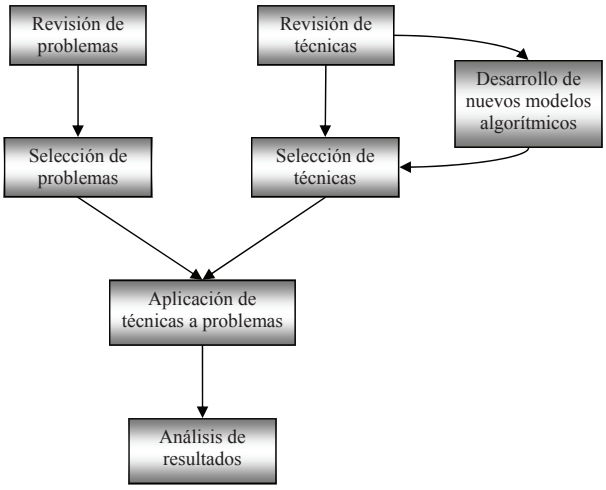


Figura 1.1: Fases seguidas durante la elaboración de esta tesis.

- Estudio de la utilidad de las metaheurísticas para la planificación de proyectos software con ayuda de un generador de instancias.
- Aplicación por primera vez de dos técnicas metaheurísticas a la generación de casos de prueba: estrategias evolutivas y optimización basada en cúmulos de partículas. Estas técnicas resultan ser más eficaces que los algoritmos genéticos, ampliamente utilizados en la literatura.
- Estudio detallado de metaheurísticas con población descentralizada para la generación automática de casos de prueba y comparación con metaheurísticas con población centralizada.
- Desarrollo de un nuevo modelo algorítmico de optimización basado en colonias de hormigas que permite trabajar con problemas en los que el grafo de construcción tiene tamaño desconocido o suficientemente grande para que no quepa en la memoria de una máquina.
- Aplicación del nuevo modelo algorítmico al problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes con un enfoque basado en *model checking*. Se realiza un análisis detallado de la aplicación explorando alternativas.
- Combinación del nuevo modelo algorítmico con una técnica propia de *model checking*, la reducción de orden parcial, para el problema mencionado en el punto anterior.
- Diseño e implementación de los algoritmos utilizados en esta tesis siguiendo el paradigma de la orientación a objetos. Estos algoritmos se han incorporado en dos bibliotecas públicamente disponibles: JEAL y MALLBA.

Además de los puntos más importantes arriba mencionados, se ha realizado una revisión y clasificación de trabajos para identificar problemas de optimización en Ingeniería del Software. También se ha diseñado e implementado un generador de instancias para el problema de planificación de proyectos software. Este

generador ha permitido analizar los resultados obtenidos con las técnicas metaheurísticas en proyectos software ficticios automáticamente generados con diferentes características.

Finalmente, se ha llevado a cabo una importante labor de diseminación del contenido de las investigaciones desarrolladas en este trabajo. Para tal fin, además de las publicaciones aparecidas en revistas y congresos tanto nacionales como internacionales, se han desarrollado páginas web de dominio público con la descripción de los problemas, el generador de instancias y los resultados de las investigaciones.

1.4. Organización de la tesis

Este documento de tesis se estructura en cuatro partes. En la primera se presentan los preliminares de las metaheurísticas y la Ingeniería del Software, detallando los problemas y las técnicas que se utilizan en la tesis. En la segunda parte se presentan las propuestas metodológicas y los resultados de la aplicación de las metaheurísticas a los problemas de Ingeniería del Software seleccionados. En la tercera parte mostramos las principales conclusiones que se desprenden de este trabajo, así como las líneas de trabajo futuro inmediatas que surgen de nuestro estudio. Finalmente, la cuarta parte contiene los apéndices con resultados y estudios secundarios que pueden ser de interés tan sólo a algunos lectores. A continuación describimos detalladamente el contenido de los capítulos.

- **Parte I: Fundamentos de las metaheurísticas y la Ingeniería del Software**

  El Capítulo 2 introduce los conceptos básicos de Ingeniería del Software. Tras esto, ofrece una breve revisión de problemas de optimización que surgen en el seno de la Ingeniería del Software y que han sido abordados en la literatura con técnicas de optimización.

  El Capítulo 3 detalla los tres problemas de optimización pertenecientes al campo de la Ingeniería del Software que se han resuelto con técnicas metaheurísticas en la presente tesis.

  El Capítulo 4 proporciona una introducción genérica sobre el campo de las metaheurísticas. Posteriormente describe con mayor nivel de detalle las técnicas metaheurísticas concretas que se usan a lo largo de la tesis.

- **Parte II: Resolución de los problemas de Ingeniería del Software seleccionados**

  El Capítulo 5 describe las propuestas metodológicas empleadas para resolver los problemas seleccionados. Se detallan los algoritmos empleados y las modificaciones a éstos, así como los nuevos modelos desarrollados especialmente para resolver los problemas.

  El Capítulo 6 presenta los resultados obtenidos al resolver el problema de planificación de proyectos software. Se resuelven distintas instancias del problema con diferentes características y se discuten los resultados para llegar a una comprensión profunda del problema que pueda servir a jefes de proyectos en su trabajo diario.

  El Capítulo 7 muestra y discute los resultados obtenidos tras la aplicación de las técnicas metaheurísticas al problema de generación de casos de prueba. Se analizan distintos algoritmos y se comparan los resultados con la literatura.

  El Capítulo 8 estudia los resultados obtenidos para el problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. Se analizan distintas configuraciones de los algoritmos y se evalúan sobre un gran conjunto de sistemas concurrentes.

- **Parte III: Conclusiones y trabajo futuro**

  Terminamos esta tesis con unas conclusiones sobre todo lo expuesto en el resto del documento. Asimismo, se describen también las principales líneas de trabajo futuro que surgen del presente estudio. Esta parte está redactada tanto en español como en inglés para cumplir los requisitos exigidos por la Universidad de Málaga para optar a la mención de Doctorado Europeo.

- **Parte IV: Apéndices**

  El Apéndice A presenta un análisis de parámetros de algunas de las técnicas empleadas en la tesis doctoral. Los resultados de este análisis se utilizaron para decidir la configuración de los algoritmos.

  El Apéndice B describe el tratamiento estadístico utilizado para validar las observaciones realizadas en los experimentos de la tesis y muestra los resultados de dicha validación.

  El Apéndice C ofrece una rápida descripción de las bibliotecas de algoritmos metaheurísticos que se han usado y diseñado durante el desarrollo de la tesis.

  El Apéndice D presenta la publicaciones del doctorando realizadas durante la elaboración de la tesis. Estas publicaciones sustentan la calidad de la presente tesis doctoral.

  El Apéndice E contiene un resumen en inglés de la tesis doctoral, necesario para optar a la mención de Doctorado Europeo.

Parte I

Fundamentos de las metaheurísticas y la Ingeniería del Software

Capítulo 2

Problemas de optimización en Ingeniería del Software

El diccionario de la Real Academia Española [231] define *software* como un "conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora" mientras que en el estándar IEEE 610.12-1990 [150] se define la *Ingeniería del Software* como "la aplicación de un método sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software"[1]. La Ingeniería del Software[2] tiene una corta historia debido a que su existencia está ligada a la de los computadores digitales, que hicieron aparición en la década de 1940. Al comienzo de la era de los ordenadores el software era relativamente simple y, por tanto, no resultaba complicado asegurar la calidad del mismo. Con el aumento de las prestaciones de los ordenadores, se abría la posibilidad al desarrollo de software cada vez más complejo apoyado en todo momento por lenguajes de programación y herramientas de más alto nivel. A lo largo de su historia, el desarrollo de software ha pasado de ser una tarea realizada por una sola persona en pocas horas a convertirse en un conjunto de actividades interrelacionadas que deben realizarse en grandes equipos de trabajo durante meses. Los *procesos de desarrollo de software* o *ciclos de vida del software* surgieron como una herramienta para poner orden en este conjunto de actividades. Se han propuesto diversos procesos de desarrollo de software en las últimas décadas entre los que destacan el ciclo de vida en cascada [239] y el ciclo de vida en espiral [39]. Estos procesos de desarrollo de software determinan un conjunto de actividades y un orden entre ellas.

Un *problema de optimización* consiste en escoger una opción de entre un conjunto de ellas que sea mejor o igual que las demás (véase el comienzo del Capítulo 4). Los problemas de optimización surgen en cualquier ámbito desde la vida cotidiana hasta la industria. A lo largo de la historia se ha dedicado mucho esfuerzo a encontrar técnicas que permitan resolver tales problemas. Es un factor común a todas las ingenierías la existencia de problemas de optimización. Un ejemplo son los problemas de diseño de componentes (ala de avión, tubería, antena de telecomunicaciones, etc.). La Ingeniería del Software, a pesar de ser una ingeniería joven, no es una excepción. De hecho, en la actualidad existe un creciente interés por aplicar técnicas de optimización a problemas de Ingeniería del Software. Una muestra palpable

[1] Traducción tomada de la wikipedia en la entrada "Ingeniería de software".

[2] El término *Ingeniería del Software* fue utilizado por primera vez por Fritz Bauer en la primera conferencia sobre desarrollo de software patrocinada por el Comité de Ciencia de la OTAN que se celebró en Garmisch, Alemania, en Octubre de 1968.

de este interés ha sido la creación del término *Search Based Software Engineering*, abreviado *SBSE*, por parte de Harman y Jones [132] para referirse a este nuevo campo de investigación.

En este capítulo se presenta una breve descripción de un conjunto de problemas surgidos dentro de la Ingeniería del Software que han sido planteados en la literatura como problemas de optimización. No se trata de una revisión exhaustiva de trabajos sino, más bien, de un escaparate en el que se puede apreciar la diversidad de los problemas que pueden definirse dentro de la Ingeniería del Software. La mayoría de los trabajos presentados han sido tomados del repositorio de publicaciones sobre SBSE mantenido por el doctor Afshin Mansouri como parte del proyecto SEBASE (`www.sebase.org`). Para presentar este conjunto de problemas de forma ordenada, proponemos una clasificación de los mismos cuyos detalles, junto con la descripción de los problemas, se encuentran en la siguiente sección.

2.1. Clasificación de los problemas de optimización en Ingeniería del Software

La clasificación que proponemos para los problemas de optimización en Ingeniería del Software se basa en la fase en la que aparece dicho problema dentro del proceso de desarrollo de software. Las fases que consideramos son: análisis de requisitos, diseño, implementación, pruebas, implantación, mantenimiento y gestión. Todas estas son fases que, bien con el mismo nombre o bien con otro, aparecen en todo proceso de desarrollo de software. A veces no está claro dónde colocar un problema en concreto; existen algunos problemas que se encuentran en el borde entre dos fases y, por tanto, podrían aparecer en ambas. Dentro de las categorías de mantenimiento y gestión hemos realizado una segunda clasificación de problemas, ya que la diversidad de éstos sugería tal división. En la Figura 2.1 mostramos la clasificación completa aquí propuesta.

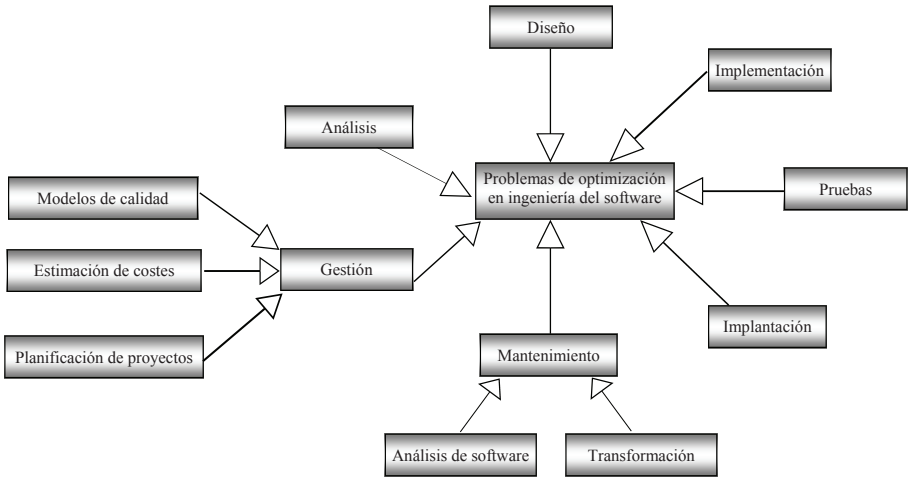


Figura 2.1: Clasificación de problemas de optimización en Ingeniería del Software.

2.1.1. Análisis de requisitos

En la fase de análisis de requisitos es donde el ingeniero software y el cliente discuten sobre lo que tiene que hacer el producto software. Es una etapa fundamental del desarrollo en la que se recoge la información del cliente y se plasma en un conjunto de requisitos para la aplicación.

En los procesos de desarrollo software utilizados actualmente se tiende a desarrollar y entregar el software de forma incremental. Del conjunto total de requisitos se seleccionan algunos y se realiza una iteración del proceso de desarrollo para, finalmente, entregar al usuario un producto software que cumpla dichos requisitos. Tras esto, se vuelven a seleccionar algunos de los requisitos no cubiertos y se realiza otra iteración del proceso de desarrollo. Se sigue de esta forma hasta que el usuario esté completamente satisfecho con el producto. La decisión de qué requisitos deben cubrirse en una iteración concreta del proceso no es algo trivial. Existen numerosas restricciones y prioridades entre los requisitos que afectan al coste del producto y a la satisfacción del usuario. Por ejemplo, un requisito puede necesitar que otro requisito previo esté ya cubierto en el software, o que otro requisito se cubra a la vez que él. Es decir, existen dependencias entre requisitos. Por su parte, el usuario puede necesitar varios requisitos con urgencia mientras que otros son menos importantes para él. A esto se suma el hecho de que cada requisito tiene un coste de implementación asociado. En definitiva, la elección de requisitos a implementar en una iteración concreta del proceso de desarrollo software es una tarea no trivial que puede beneficiarse de las técnicas de optimización existentes.

Greer y Ruhe abordan el problema de la selección de requisitos para cada iteración del proceso de desarrollo software en [116]. Baker *et al.* [30] resuelven el problema de la "siguiente versión", que consiste en seleccionar los requisitos que van a implementarse en la siguiente iteración del proceso de desarrollo. Una versión multiobjetivo de este último problema, planteada por Zhang, Harman y Mansouri, puede encontrarse en [297].

2.1.2. Diseño

En la fase de diseño los ingenieros software desarrollan una solución que cumple los requisitos recolectados en el análisis. En esta fase se decide el diseño arquitectónico y de comportamiento del software con suficiente detalle como para que pueda ser implementado, pero sin llegar a ser una implementación.

Cuando, en el desarrollo del software, se usa el paradigma de orientación a objetos, uno de los objetivos de la fase de diseño consiste en proponer el conjunto de clases que formarán la aplicación. Las clases son entidades con cierta autonomía que colaboran entre ellas. En un buen diseño, se espera que el acoplamiento (dependencia) entre clases sea bajo y la cohesión de cada clase (relación entre las tareas de los miembros de la clase) sea alta. Persiguiendo estos objetivos, Simons y Parmee [256] plantean el diseño conceptual del software como un problema de optimización. Partiendo de un conjunto de métodos con parámetros y atributos que han sido identificados mediante los casos de uso, los autores abordan el problema de encontrar una asignación de los métodos a clases que minimize el acoplamiento y maximize la cohesión.

Durante el desarrollo del software se generan distintos componentes, clases o módulos que deben ser integrados conjuntamente para formar el producto final. Estos componentes son testados de forma aislada (prueba unitaria o *unit testing*) conforme se desarrollan y, tras integrarlos con el resto de componentes ya terminados, se prueban de nuevo para comprobar si la interacción entre ellos es correcta (prueba de integración o *integration testing*). En este último caso, es posible que un componente necesite otro que aún no está completo, siendo necesario sustituir el componente incompleto por un componente temporal que tiene la misma interfaz, conocido como *stub*. La implementación de estos *stubs* es costosa y puede evitarse

o reducirse si se implementan los componentes siguiendo un orden adecuado, que dependerá fundamentalmente del grafo de interrelaciones de componentes. La determinación de este orden es un problema de optimización que ha sido abordado en [45]. Hemos colocado este problema en la sección de diseño, a pesar de estar a caballo entre el diseño y la implementación, porque el objetivo del problema no es mejorar ningún aspecto de la implementación, sino más bien ahorrar costes del proyecto.

2.1.3. Implementación

En la fase de implementación es donde el diseño previamente creado se convierte en código fuente. Los ingenieros tienen que hacer realidad el diseño desarrollado en la fase anterior.

Los problemas de optimización que se han planteado en la fase de implementación tienen que ver con la mejora o creación de herramientas que ayuden al programador en su labor. Por ejemplo, Reformat *et al.* [235] plantean el problema de la generación automática de clones. Este problema consiste en generar automáticamente un fragmento de código fuente (una función por ejemplo) que sea equivalente a otro dado. Esta técnica tiene aplicaciones en el desarrollo de software tolerante a fallos.

Los compiladores son herramientas imprescindibles en la implementación del software y, a su vez, una fuente de gran cantidad de problemas de optimización que deben abordarse de algún modo a la hora de generar el código objeto. La forma habitual de resolver estos problemas es mediante una heurística *ad hoc* que, si bien no ofrece una solución óptima en todos los casos, su tiempo de ejecución es corto, lo cual es una propiedad deseable para poder incorporarla en un compilador. Stephenson *et al.* [259] plantean el problema de optimizar estas heurísticas.

En general, las optimizaciones que aplica un compilador no son conmutativas, es decir, diferentes secuencias de optimización dan lugar a distintos resultados (código objeto). Cooper *et al.* [63] abordan el problema de encontrar una secuencia de optimización óptima para reducir el espacio que ocupa el código resultante, lo cual tiene una importante aplicación en el desarrollo de software para sistemas empotrados.

Otro problema resuelto de forma no óptima es la gestión de memoria dinámica. Del Rosso [74] aborda el problema de ajustar la configuración del montículo (*heap*) para reducir la fragmentación interna en listas libres segregadas mientras que Cohen *et al.* [60] aplica técnicas de *clustering* a las hebras con el objetivo de identificar grupos de hebras que accedan a los mismos objetos. Estos grupos compartirán un montículo, haciendo que el recolector de basura no tenga que detener todas las hebras para llevar a cabo su labor, sino tan sólo las del grupo afectado.

Otro interesante problema planteado en el seno de los compiladores es la generación automática de código paralelo óptimo a partir de código secuencial. Este problema ha sido abordado por Nisbet [220], Williams [290] y Ryan [241].

Dada una unidad de código fuente (función, clase, etc.) es posible aplicar una secuencia de transformaciones a dicho código que preserva la semántica pero que altera la sintaxis. Esta secuencia de transformaciones podría reducir el número de líneas de código. El problema de encontrar la secuencia de transformaciones que minimice la longitud del código resultante ha sido abordado en [97] y [98].

2.1.4. Pruebas

En la fase de pruebas es donde el software implementado es testado para descubrir errores y, en caso contrario, comprobar que se comporta de acuerdo a las especificaciones.

Para testar un método o función del software es necesario decidir los valores de sus argumentos de entrada. Un *caso de prueba* es una combinación concreta de valores de para dichos argumentos. La labor

del ingeniero de pruebas es crear un conjunto de casos de prueba y testar el software con él. Este conjunto de casos de prueba debe ser tal que maximice la probabilidad de descubrir los posibles errores del objeto de la prueba. El enfoque más popular para conseguir conjuntos de casos de prueba adecuados es usar criterios de cobertura: un conjunto de casos de prueba se considera adecuado si consigue cubrir una gran cantidad de elementos (instrucciones, ramas, predicados atómicos, etc.) del objeto de prueba. El problema de la generación automática de casos de prueba es, con diferencia, el más estudiado de todos los problemas de optimización relacionados con la ingeniería el software. Desde el primer trabajo de Miller y Spooner de 1976 [207] hasta nuestros días, se han propuesto diversas formas de abordar esta tarea. Uno de los paradigmas empleados, el paradigma *estructural*, hace uso de información estructural del programa para generar los casos de prueba [5, 19, 20, 34, 194, 206]. Esta información proviene generalmente del grafo de control de flujo. En numerosos trabajos se han estudiado y propuesto diferentes funciones objetivo [33, 41, 55, 180, 274]. Así mismo, para aumentar la eficiencia y eficacia del proceso de generación de casos de prueba, se han combinado las estrategias de búsqueda con otras técnicas tales como *slicing* [131], transformación de código [130, 137, 166, 197], análisis de dependencia de datos [165], uso de medidas software [169, 170, 243], búsqueda en dominios variables [244], y otros [198, 200, 292]. Se han propuesto soluciones para algunos de los problemas principales que surgen en la generación de casos de prueba: la presencia de *flags* [31, 32, 40, 129], la existencia de estados internos en el programa [199, 201, 296], las particularidades del software orientado a objetos [279, 280] y la presencia de excepciones [271]. Entre las técnicas más populares para la resolución del problema se encuentran los algoritmos genéticos [103, 152, 153, 191, 224, 237, 238, 261, 291], la búsqueda tabú [70], la búsqueda dispersa [246] y los algoritmos de estimación de distribuciones [242, 245]. También se han desarrollado herramientas para la generación automática de casos de prueba usando un enfoque basado en optimización [71, 267, 268, 269, 281]. La generación automática de casos de prueba usando información estructural del software se ha usado con éxito para detectar desbordamientos de *buffers* [72, 73] y comprobar software crítico [272].

Un segundo paradigma de generación de casos de prueba, denominado *funcional*, considera el software como una caja negra [193, 247, 283]. También se ha estudiado la generación de casos de prueba para realizar *pruebas de conformidad* de máquinas de estado finitas [76, 77, 124, 125, 126, 138, 177]. Se han usado técnicas de programación con restricciones [62] y se han evaluado los conjuntos de casos de prueba usando *mutantes* [1, 95, 252, 295]. Los mutantes son programas que contienen pequeñas variaciones con respecto al objeto de la prueba. Normalmente, las variaciones que se incluyen en los mutantes representan los errores más comunes que se cometen al programar. En este contexto, un conjunto de casos de prueba es adecuado si es capaz de distinguir entre el programa original y los mutantes observando únicamente la salida de ambos programas. La evaluación de conjuntos de casos de prueba con esta técnica constituye otro criterio de adecuación alternativo a los criterios de cobertura.

Las *pruebas de interacción* consisten en comprobar todas las combinaciones posibles de entornos de ejecución del software: distintos sistemas operativos, distintas configuraciones de memoria, etc. Cuantos más factores se consideren mayor es el número posible de combinaciones, creciendo éste de forma exponencial. Por este motivo, se han propuesto estrategias basadas en optimización para reducir este enorme conjunto de combinaciones [47, 48, 61].

Por otro lado, existe un conjunto de trabajos en el que se comprueban aspectos no funcionales del software. La mayoría de ellos se centran en el tiempo de ejecución, con aplicaciones a los sistemas en tiempo real [46, 79, 215, 228, 266, 284, 285, 286]. Algunos trabajos extienden las técnicas para abordar software orientado a objetos y basado en componentes [123, 120]. También se han definido medidas que permiten conocer, tras un examen del código fuente del programa, si la aplicación de técnicas basadas en computación evolutiva es adecuada o no para comprobar las restricciones temporales [119, 121, 122].

Cuando el software es modificado, es necesario volver a aplicar algunas pruebas para comprobar que no se han introducido nuevos errores. Esto es lo que se conoce como *pruebas de regresión* (*regression testing*). Muchas de estas pruebas pueden reutilizarse de una etapa anterior, cuando el software pasó por primera vez la fase de pruebas. El problema de seleccionar las pruebas que se van a aplicar de nuevo ha sido planteado como un problema de optimización en dos recientes trabajos de Yoo y Harman [293] y Li *et al.* [178].

Por último, el problema de verificar software concurrente ha sido tradicionalmente resuelto mediante *model checkers*. Este tipo de técnicas suele reservarse exclusivamente a modelos de software concurrente crítico, y no se aplica al software que se desarrolla en un proyecto de mediana o gran envergadura por su escasa escalabilidad. Sin embargo, existen algunos trabajos en la literatura que, basándose en la exploración que realizan los *model checkers*, tratan de buscar errores en software concurrente planteando dicho problema como una búsqueda en un grafo [13, 15, 16, 17, 92, 93, 94, 111]. Este enfoque tiene la ventaja de que puede escalar más fácilmente que el *model checking* clásico, permitiendo su aplicación al software producido normalmente en la compañías. En otros trabajos, el objetivo no es buscar errores, sino generar casos de prueba para sistemas concurrentes [294].

2.1.5. Implantación

Tras desarrollar y probar el software, procede la fase de implantación en la que el sistema software es instalado y desplegado en su entorno definitivo, donde será utilizado.

Monnier *et al.* [214] abordan el problema de la planificación de tareas en un sistema distribuido de tiempo real. Dado un conjunto de tareas cuyo tiempo de ejecución está acotado y con dependencias entre ellas, el problema consiste en asignar a cada tarea una máquina para su ejecución y planificar los instantes de tiempo en los que se comunicarán las tareas usando la red.

2.1.6. Mantenimiento

La fase de mantenimiento es donde el software se modifica como consecuencia de las sugerencias de los usuarios o, simplemente, con el objetivo de mejorar su calidad. Al igual que ocurría en la fase de pruebas, existen muchos trabajos que afrontan problemas surgidos en la fase de mantenimiento usando técnicas de optimización. Para realizar un clasificación más fina de todos estos trabajos, los dividimos en dos grupos: *análisis de software* y *transformación*.

Análisis de software

Una de las técnicas usadas durante la fase de mantenimiento para llegar a una rápida comprensión de la estructura y los objetivos de un determinado fragmento de código fuente es la *vinculación de conceptos* (*concept binding*). Esta técnica consiste en asignar automáticamente a cada segmento del código fuente una palabra o concepto perteneciente a un mapa conceptual previamente definido. Durante una primera etapa, se asigna a cada línea o instrucción del código fuente un *indicador*. Estos indicadores señalan la posible vinculación de un determinado concepto al segmento de código en el que se encuentra el indicador. La siguiente etapa consiste en asignar conceptos a segmentos de código en función de los indicadores que se encuentren. Esta segunda etapa ha sido planteada como problema de optimización por Gold *et al.* [113].

Otras técnicas utilizadas para extraer automáticamente una estructura de alto nivel del software a partir del código fuente son las técnicas de *software clustering*. La idea detrás de ellas es agrupar distintos elementos del software (clases, funciones, etc.) en módulos o subsistemas de acuerdo a su cohesión y

acoplamiento. Este problema, originalmente propuesto por Mancoridis *et al.* en [188], ha sido investigado en profundidad en numerosos trabajos [90, 128, 133, 185, 186, 187, 210]. Mitchell et at. [211, 212] proponen un procedimiento completo de ingeniería inversa compuesto por una primera etapa en la que se aplican técnicas de *software clustering* seguida de una segunda etapa de reconocimiento de estilos de interconexión entre los subsistemas inferidos en la primera etapa.

La detección de clones consiste en descubrir fragmentos del código fuente que son idénticos o similares sintáctica o semánticamente. Sutton *et al.* [262] resuelven el problema de detección de grupos de clones en el código fuente.

Transformación

Basados en la idea del *software clustering*, Seng *et al.* [250] plantean el problema de mejorar la descomposición del software en subsistemas. Se trata de encontrar una partición de los elementos del software (clases o funciones) formando subsistemas de manera que se optimicen una serie de parámetros como la cohesión, el acoplamiento o la complejidad.

Slicing es una técnica usada para extraer automáticamente un fragmento no contiguo del código que resulta de valor para una acción concreta. Una forma de realizar esta extracción es seleccionar un conjunto de variables y obtener una proyección del software (un subconjunto de instrucciones) que posea la misma semántica que el original con respecto a esas variables. Una variante de esta técnica, denominada *slicing amorfo* (*amorphous slicing*), consiste en generar un fragmento de código que preserve la semántica del programa con respecto a las variables seleccionadas pero sin necesidad de preservar la sintaxis. En [96], Fatiregun *et al.* abordan el *slicing* amorfo como un problema de optimización.

Refactoring es una técnica para reestructurar el código fuente, cambiando su estructura interna, sin modificar la semántica. La base de esta técnica se encuentra en una serie de pequeñas transformaciones del código que preservan el comportamiento del software. El objetivo del *refactoring* es encontrar una secuencia de transformaciones que den lugar a una estructura más clara, legible y fácil de mantener. Este problema se ha resuelto utilizando técnicas de optimización en numerosas ocasiones [42, 134, 222, 251].

2.1.7. Gestión

La gestión de un proyecto software es una tarea que debe llevarse a cabo durante todo el proceso de desarrollo, controlándolo en todo momento y realizando los ajustes pertinentes. Para ello, es necesario medir diversos aspectos del producto y del proceso, y tratar de predecir otros como la calidad del software. Asimismo, entre las labores de gestión se encuentra la asignación de personal a tareas. De nuevo aquí dividimos los trabajos en tres categorías: modelos de calidad, estimación de costes y planificación de proyectos.

Modelos de calidad

Uno de los problemas planteados en la gestión de un proyecto software es el de la predicción de la calidad del software. Bouktif *et al.* [43] plantean el problema de encontrar un modelo preciso capaz de predecir la calidad del software en base a una serie de atributos externos e internos de éste.

Al equipo encargado de garantizar la calidad de un proyecto software le puede interesar la posibilidad de obtener una lista de los módulos del software ordenados de acuerdo a su calidad, para así acometer las acciones de mejora adecuadas. El problema de ajustar un modelo para la obtención de estas listas de módulos es abordado por Khoshgoftaar *et al.* en [160] y [159].

Por otro lado, Vivanco y Pizzi [277] resuelven el problema de seleccionar las medidas adecuadas para determinar la mantenibilidad de un proyecto software, un aspecto de los modelos de calidad.

Estimación de costes

La predicción del coste de un proyecto software a partir de un conjunto de datos básicos del mismo es una muy preciada técnica que ha sido resuelta mediante algoritmos de optimización. Sheta en [254] plantea el problema de ajustar los parámetros de un modelo de predicción ya existente: COCOMO. Por otro lado, varios autores han abordado el problema de la creación de un modelo desde cero [52, 83, 84, 85, 162, 175, 253, 255].

Planificación de proyectos

Aguilar Ruiz *et al.* [2, 3] han estudiado el problema de extraer reglas a partir de una simulación de un proyecto software. Estas reglas ayudan al gestor del proyecto a realizar los ajustes pertinentes para conseguir reducir el coste y la duración del proyecto.

Chang *et al.* [54] abordan el problema de asignar empleados a tareas teniendo en cuenta sus habilidades, su salario y su dedicación con el objetivo de minimizar el coste y la duración del proyecto. Alba y Chicano [6, 18] abordan este mismo problema analizando distintos proyectos automáticamente generados con un generador de instancias.

La asignación de *work packages* (WP) de un proyecto a equipos de programadores con el objetivo de reducir la duración del proyecto es un problema de optimización que ha sido abordado por Antoniol *et al.* [25, 26]. Los mismos autores amplían el problema en [24] para optimizar también la asignación de programadores a equipos y considerar soluciones robustas que sean capaces de tolerar el posible abandono de WPs, errores, revisiones e incertidumbre en las estimaciones.

2.2. Conclusiones

Como se ha podido comprobar a lo largo de este capítulo, la diversidad de problemas de optimización que surgen en el seno de la Ingeniería del Software es alta. Los investigadores e ingenieros del software aplican cada vez más algoritmos de optimización a tales problemas y, debido a esto, el número de trabajos de investigación relacionados ha ido creciendo progresivamente en los últimos años. Para ilustrar esto, mostramos en la Figura 2.2 el número de trabajos presentados en este capítulo ordenados por año. Como se puede apreciar, existe un aumento casi lineal de trabajos entre los años 1999 y 2004. El descenso que se observa para el año 2006 puede ser debido a que la revisión que hemos hecho no es exhaustiva. Por otro lado, los últimos trabajos que recogemos en este capítulo pertenecen a principios de Julio de 2007 y, por tanto, se espera que el número de trabajos en 2007 sea aproximadamente el doble del que se refleja en la figura.

Si bien es cierto que se han abordado problemas de todas las fases del ciclo de vida del software, existe una fase que ha recibido mayor atención que el resto: la fase de pruebas. En la Figura 2.3 mostramos el número de trabajos presentados en este capítulo ordenados por la categoría en la que lo hemos clasificado. El número de trabajos que se enmarcan en la fase de pruebas alcanza casi el centenar (97 desde 1992) mientras que la segunda categoría con más trabajos es la de gestión con 21. La mayoría de los 97 trabajos de la categoría de pruebas abordan el problema de la generación automática de casos de prueba. Esto da una idea de las principales preocupaciones en Ingeniería del Software. Como dice Glenford Myers en [217],

aproximadamente la mitad del tiempo de un proyecto software y más de la mitad de su coste se dedica a la fase de pruebas, lo cual explica por qué la mayoría de los trabajos se dedican a mejorar dicha fase.

Esta tesis aborda tres de los problemas arriba presentados: la planificación de proyectos software, la generación automática de casos de prueba y la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. Estos problemas pertenecen a las dos categorías que más han interesado a la comunidad científica: pruebas y gestión. Con la elección de dos problemas pertenecientes a la fase de pruebas frente a uno de gestión, contribuimos también a mantener las proporciones.

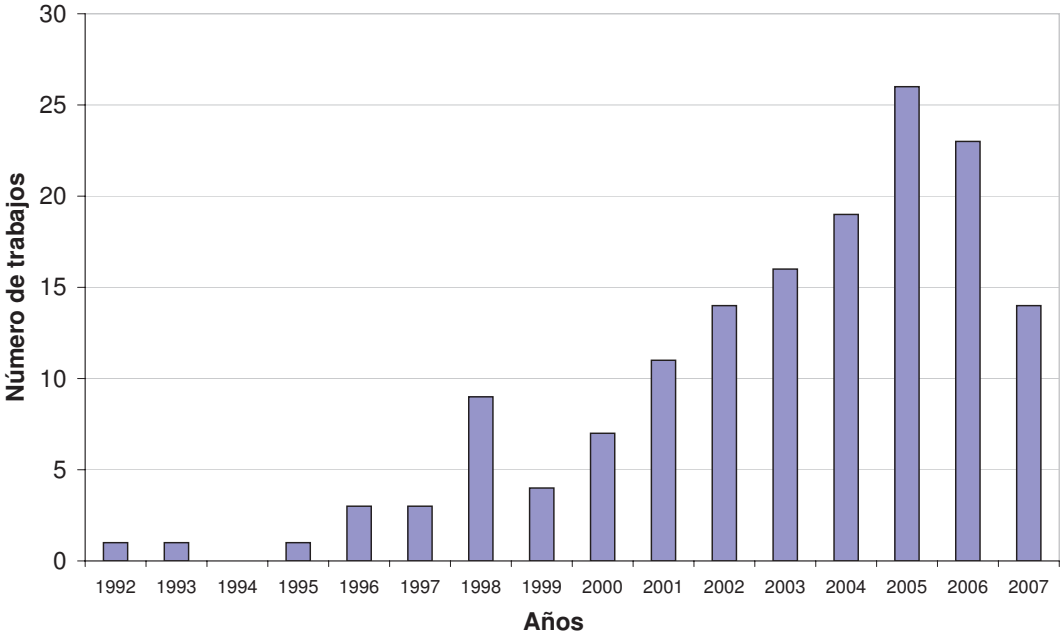


Figura 2.2: Resumen de los trabajos discutidos en este capítulo. Número de trabajos por año.

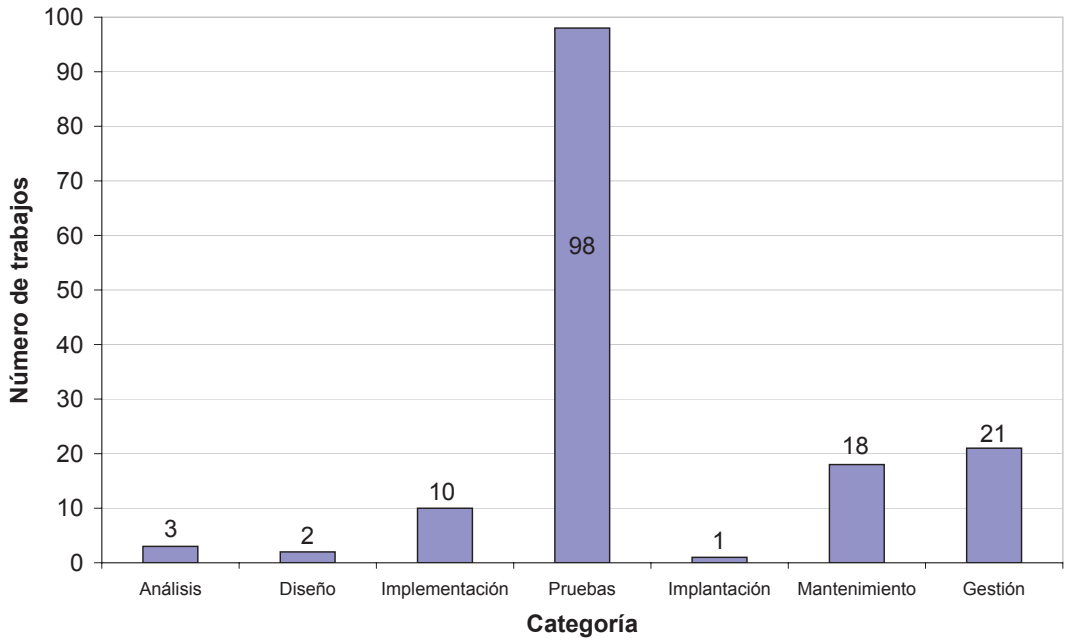


Figura 2.3: Resumen de los trabajos discutidos en este capítulo. Número de trabajos por categoría.

Capítulo 3

Problemas abordados

En este capítulo presentamos con detalle los tres problemas de Ingeniería del Software que se han escogido para aplicarles técnicas metaheurísticas. Estos tres problemas son la planificación de proyectos software, la generación de casos de prueba y la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes.

Es necesaria una buena planificación de los recursos humanos involucrados en un proyecto software para poder conseguir un ahorro en tiempo y coste a la vez que se asegura que el producto posee una calidad mínima. De esta forma, el tiempo o el presupuesto sobrante puede utilizarse para otros proyectos o, incluso, para aumentar la calidad del producto software. El problema de planificación de proyectos software se detalla en la Sección 3.1.

La fase de pruebas del software es siempre una tarea necesaria que consume gran cantidad de recursos en todo proyecto. Una de las labores más complejas de esta tarea es la generación de casos de prueba adecuados que permitan detectar posibles errores, comprobar el correcto funcionamiento de la mayor parte del código generado y, en definitiva, asegurar con alta probabilidad que el software cumple con precisión la especificación inicial. El problema de generación de casos de prueba se detalla en la Sección 3.2.

En los sistemas reactivos, la comprobación del software con casos de prueba no es suficiente para asegurar su correcto funcionamiento, ya que, debido a la naturaleza concurrente y reactiva del sistema, el comportamiento del software ante un caso de prueba puede ser diferente en instantes distintos. En estos sistemas se recurre a técnicas formales que comprueban si el software cumple una serie de propiedades en cualquier posible ejecución del mismo y, cuando no es así, devuelven una ejecución en la que la propiedad es violada. Los detalles sobre el problema de búsqueda de este tipo de violaciones en sistemas concurrentes se dan en la Sección 3.3.

3.1. Planificación de proyectos software

La gran complejidad de los proyectos de software actuales justifica la investigación en herramientas asistidas por computador para planificar apropiadamente el desarrollo del proyecto. En la actualidad, este tipo de proyectos demanda normalmente una gestión compleja que involucra la programación temporal, planificación y monitorización de tareas. Por ejemplo, es necesario controlar los procesos que componen el proyecto y el personal disponible, y asignar recursos eficientemente para conseguir objetivos específicos, a la vez que se cumple un conjunto de restricciones. De forma general, el problema de planificación

de proyectos software consiste en definir qué recursos se usan para realizar cada tarea y cuándo debería llevarse a cabo. Las tareas pueden ser muy diversas: desde mantener documentos hasta escribir programas. Entre los recursos se incluyen personal, material informático, tiempo, etc. Los objetivos son normalmente minimizar la duración del proyecto, el coste, y maximizar la calidad del producto [54]. En los proyectos reales, el gestor del proyecto desearía poder obtener de forma automática planificaciones que reconcilien en la medida de lo posible estos tres objetivos conflictivos.

En las siguientes secciones definiremos el problema de planificación de proyectos (Sección 3.1.1), lo ubicaremos dentro de la clasificación de Travassos y Barros [273] de experimentos en Ingeniería del Software empírica (Sección 3.1.2) y mencionaremos algunos trabajos relacionados (Sección 3.1.3).

3.1.1. Definición del problema

En el problema de planificación de proyectos (*Project Scheduling Problem*, PSP[1]), los recursos gestionados son personas con un conjunto de habilidades y un salario. Estos empleados tienen un grado máximo de dedicación al proyecto. Formalmente, cada persona (empleado) se denota con e_i, donde i va de 1 a E (el número total de empleados). Sea SK el conjunto de habilidades consideradas en el proyecto y s_j la j-ésima habilidad con j variando entre 1 y $S = |SK|$. Las habilidades del empleado e_i se denotarán con $e_i^{skills} \subseteq SK$, el salario mensual con e_i^{salary}, y el grado máximo de dedicación al proyecto con e_i^{maxded}. El salario y la dedicación máxima son números reales. El primero se expresa en una unidad monetaria ficticia, mientras que la segunda es el cociente entre la cantidad de horas dedicadas al proyecto y las horas de trabajo de una jornada laboral completa. Las tareas del proyecto se denotan con t_k donde k va de 1 a T (el número de tareas). Cada tarea t_k tiene asociada un conjunto de habilidades requeridas que denotamos con t_k^{skills} y un esfuerzo t_k^{effort} expresado en personas-mes (PM). Las tareas deben ser realizadas de acuerdo al orden establecido por el grafo de precedencia de tareas (*Task Precedence Graph*, TPG). Éste indica qué tareas debe completarse antes de que una nueva tarea comience. El TPG es un grafo orientado acíclico $G(V, A)$ con un conjunto de vértices $V = \{t_1, t_2, \ldots, t_T\}$ que coincide con el de tareas y un conjunto de arcos $A \in V \times V$, donde $(t_i, t_j) \in A$ si la tarea t_i debe completarse antes de que la tarea t_j pueda iniciarse.

Los objetivos del problema son minimizar el coste y la duración del proyecto. Además, la solución debe cumplir tres restricciones:

- **R1**: cada tarea debe ser realizada por al menos una persona.
- **R2**: el conjunto de las habilidades requeridas por una tarea debe estar incluido en la unión de las habilidades de los empleados que realizan la tarea.
- **R3**: ningún empleado debe exceder su dedicación máxima al proyecto.

La primera restricción es necesaria para completar el proyecto: si hay tan sólo una tarea sin hacer, el proyecto no se completará. La segunda restricción requiere una discusión más profunda que retrasamos hasta el final de esta sección. Por último, la tercera restricción resulta obvia de acuerdo a la definición de dedicación máxima.

Una vez que conocemos los elementos del problema, podemos pasar a describir los elementos de una solución al mismo. Una solución se puede representar con una matriz $\mathbf{X} = (x_{ij})$ de tamaño $E \times T$ donde $x_{ij} \geq 0$. El elemento x_{ij} es el grado de dedicación del empleado e_i en la tarea t_j. Si el empleado e_i realiza

[1] Seguiremos la norma en esta tesis de usar los acrónimos de los problemas y los algoritmos en inglés.

la tarea t_j con un grado de dedicación 0.5, éste trabaja durante media jornada laboral en la tarea. Si un empleado no realiza una tarea tendrá un grado de dedicación 0 en ella.

Para evaluar la calidad de una solución, tendremos en cuenta tres aspectos: la duración del proyecto, el coste del proyecto, y la factibilidad de la solución. En primer lugar necesitamos calcular la duración de cada tarea individual (t_j^{dur}). Esto se hace con la expresión:

$$t_j^{dur} = \frac{t_j^{effort}}{\sum_{i=1}^{E} x_{ij}} \ . \tag{3.1}$$

A continuación se puede calcular el tiempo de inicio y finalización de cada tarea (t_j^{start} y t_j^{end}) teniendo en cuenta el TPG, con lo cual se obtiene la planificación temporal de las tareas (diagrama de Gantt)[2]. Una vez conocidos los tiempos de finalización de todas las tareas, es inmediato obtener la duración del proyecto p_{dur}, que se define como el máximo de los tiempos de finalización de las tareas:

$$p_{dur} = max\left\{t_j^{end} | 1 \leq j \leq T\right\} \ . \tag{3.2}$$

El coste del proyecto p_{cost} es la suma de las cantidades de dinero pagadas a los empleados por su dedicación al mismo. Estas cantidades se calculan multiplicando el salario del empleado por el tiempo dedicado al proyecto, el cual, a su vez, es la suma de la dedicación a cada tarea multiplicada por la duración de la misma. En resumen:

$$p_{cost} = \sum_{i=1}^{E} \sum_{j=1}^{T} e_i^{salary} \cdot x_{ij} \cdot t_j^{dur} \ . \tag{3.3}$$

Ahora detallamos cómo se verifican las restricciones. Para determinar si una solución es factible debemos comprobar primero que todas las tareas son realizadas por alguien, es decir, ninguna tarea queda sin hacer (**R1**). Formalmente esto es:

$$\sum_{i=1}^{E} x_{ij} > 0 \quad \forall j \in \{1, 2, \ldots, T\} \ . \tag{3.4}$$

La restricción **R2** se formaliza con la siguiente expresión:

$$t_j^{skills} \subseteq \bigcup_{\{i | x_{ij} > 0\}} e_i^{skills} \quad \forall j \in \{1, 2, \ldots, T\} \ . \tag{3.5}$$

Antes de formalizar la restricción **R3** necesitamos una expresión para el exceso de trabajo de los empleados p_{over}. Definimos la función de trabajo de cada empleado e_i^{work} como:

$$e_i^{work}(t) = \sum_{\{j | t_j^{start} \leq t \leq t_j^{end}\}} x_{ij} \ . \tag{3.6}$$

Si $e_i^{work}(t) > e_i^{maxded}$ el empleado e_i excede su máxima dedicación en el instante t. El trabajo extra del empleado e_i^{over} es:

$$e_i^{over} = \int_{t=0}^{t=p_{dur}} ramp(e_i^{work}(t) - e_i^{maxded}) dt \ , \tag{3.7}$$

[2]Asumimos que las tareas t_j con grado de entrada $ge(t_j) = 0$ en el TPG comienzan en el instante 0.

donde *ramp* es la función definida por:

$$ramp(x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{si } x \leq 0 \end{cases} . \tag{3.8}$$

Con todo esto, la restricción **R3** se expresa formalmente con:

$$p_{over} = \sum_{i=1}^{E} e_i^{over} = 0 \ . \tag{3.9}$$

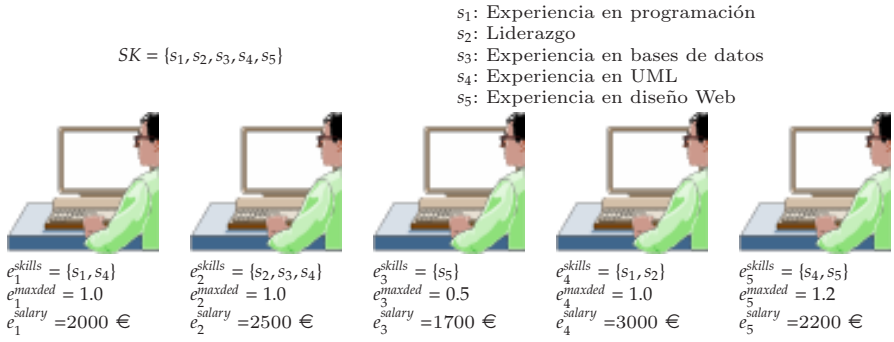


Figura 3.1: Plantilla de una compañía ficticia.

Consideremos un ejemplo para clarificar los conceptos. Supongamos que tenemos una empresa de software con los cinco empleados mostrados en la Figura 3.1 que debe desarrollar una aplicación para un banco. En dicha figura suministramos información sobre las diferentes habilidades de los empleados, su máxima dedicación al proyecto, y su salario mensual. Por ejemplo, el empleado e_2, que gana 2500 € cada mes, es experto en bases de datos (s_3), en UML (s_4), y es capaz de liderar un grupo de trabajadores (s_2). Su colega, el empleado e_4, es también capaz de liderar un grupo (s_2) y, además, es un gran programador (s_1). Estos dos empleados y el empleado e_1 pueden dedicar su jornada laboral completa al desarrollo de la aplicación (máxima dedicación 1). Por otro lado, el empleado e_3 sólo puede dedicar la mitad de una jornada laboral al proyecto. Podemos encontrar varias razones para este hecho: quizá el empleado tiene un contrato a media jornada, tiene tareas administrativas durante parte del día, o trabaja en otro proyecto de la empresa. El empleado e_5 puede trabajar más de una jornada completa en el proyecto, su dedicación máxima es mayor que uno ($e_5^{maxded} = 1.2$). De este modo, podemos modelar las horas extra de los empleados, una característica del mundo real incluida en la definición del problema. No obstante, el gestor del proyecto debe tener en cuenta que una sobrecarga de trabajo puede aumentar también la tasa de error del empleado y, con ello, el número de errores del software desarrollado. Esto conduce a una baja calidad del producto final y, posiblemente, a la necesidad de corregir o desarrollar de nuevo las partes afectadas. En cualquier caso, la consecuencia puede ser un aumento de la duración del proyecto. Esto no afecta a la definición del problema, es un asunto de la condición humana de los empleados, pero es una importante cuestión que un gestor de proyecto deberá tener en cuenta.

En la Figura 3.2 mostramos todas las tareas del proyecto software. Para cada tarea indicamos el esfuerzo en personas-mes y el conjunto de habilidades requeridas. Por ejemplo, la tarea t_1, que consiste en realizar los diagramas UML del proyecto para usarlos en las siguientes etapas, requiere experiencia en

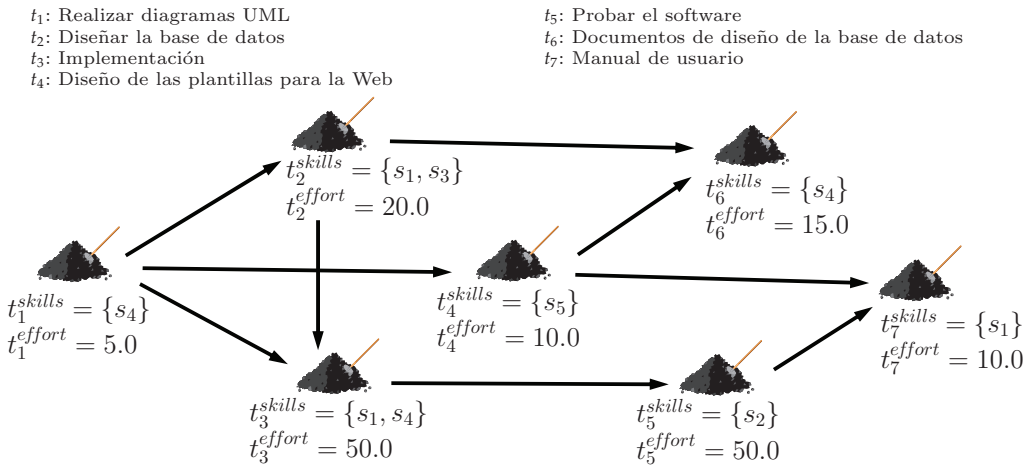


Figura 3.2: Grafo de precedencia de tareas para la aplicación bancaria de ejemplo.

UML (habilidad s_4) y cinco personas-mes. En la misma figura mostramos el TPG del proyecto. Podemos observar, por ejemplo, que después de completar los diagramas UML de la aplicación (t_1), se puede iniciar el diseño de las plantillas para las páginas web (t_4) y el diseño de la base de datos (t_2). No obstante, estas dos tareas deben terminarse antes de producir la documentación del diseño de la base de datos (t_6).

En la Figura 3.3 mostramos una posible solución para el problema de planificación que plantea el ejemplo desarrollado. La suma de los elementos de la columna asociada a la tarea t_2 es la dedicación total de los empleados a esa tarea: 4 personas-mes en el ejemplo. El esfuerzo requerido para realizar la tarea t_2^{effort} se divide por dicha dedicación total para calcular la duración de la tarea (t_2^{dur} en la figura). Con las duraciones de todas las tareas, y teniendo en cuenta el TPG, se puede calcular el diagrama de Gantt que se muestra en la Figura 3.3. Por otro lado, mostramos en la Figura 3.4 la función de trabajo del empleado e_5 asociada a esa solución. En esta última figura hemos incluido también las tareas que realiza el empleado en cada momento. La línea gruesa es la función de trabajo del empleado, $e_5^{work}(t)$, y la línea discontinua indica la máxima dedicación (1.2). Cuando la función de trabajo pasa por encima de la dedicación máxima hay exceso de trabajo.

En este punto podemos hablar sobre el número de habilidades involucradas en un proyecto. Este número puede verse como una medida del grado de especialización de la experiencia requerida para la realización del proyecto. Es decir, si el número de habilidades es alto, el conocimiento requerido para realizar el proyecto se divide en un mayor número de porciones que si el número de habilidades es bajo. En nuestro ejemplo podríamos aún dividir más algunas de las habilidades. Por ejemplo, podemos dividir la experiencia en programación en tres habilidades más pequeñas: experiencia en Java, experiencia en C/C++, y experiencia en Visual Basic. Por otro lado, el número de habilidades se puede ver como una medida de la cantidad de conocimiento (experiencia) necesaria para realizar el proyecto. Por ejemplo, en el desarrollo de software para controlar una aeronave se requiere una gran cantidad de conocimiento de distintas disciplinas. La cantidad de conocimiento necesaria para elaborar dicho software es mayor que la requerida para la aplicación bancaria de nuestro ejemplo. Por tanto, el proyecto de desarrollo del controlador tendría un mayor número de habilidades que el proyecto de la aplicación bancaria.

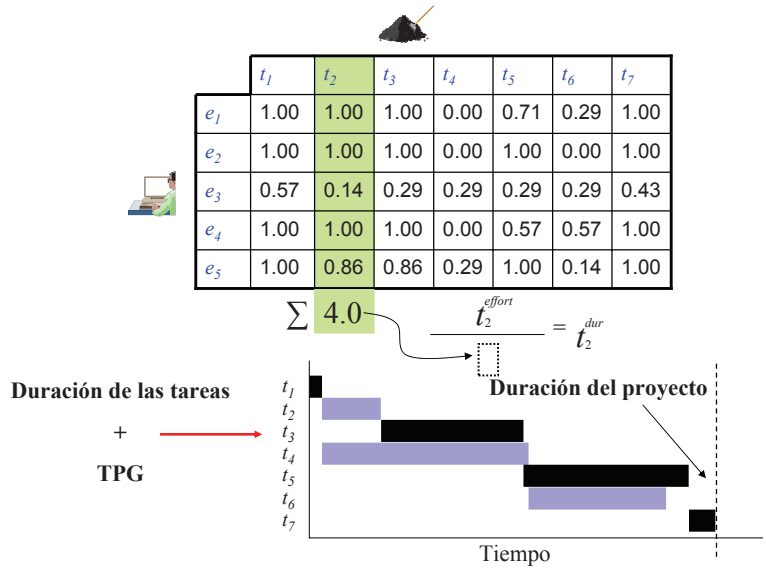


Figura 3.3: Una solución tentativa para el ejemplo anterior. Usando las duraciones de las tareas y el TPG se puede calcular el diagrama de Gantt del proyecto.

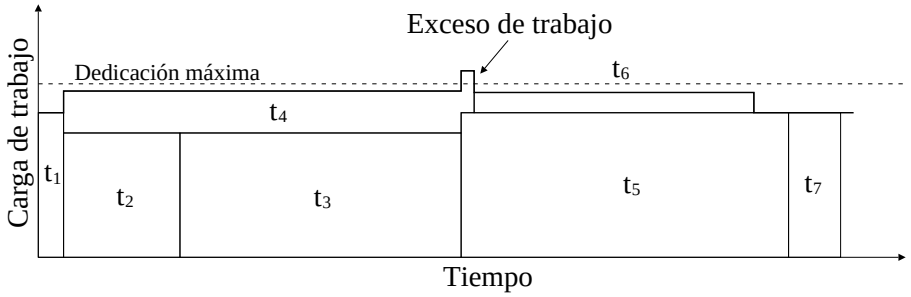


Figura 3.4: Función de trabajo del empleado e_5 en nuestro ejemplo (línea gruesa).

En definitiva, el número de habilidades de un proyecto en nuestro modelo tiene una interpretación dual en el mundo real: grado de especialización del conocimiento frente a la cantidad de conocimiento. La interpretación correcta depende del proyecto específico. Desde el punto de vista del gestor, las habilidades asignadas a cada tarea y empleado dependen de la division del conocimiento requerido para el proyecto en cuestión. Por ejemplo, podemos hacer una división muy fina del conocimiento si nuestros empleados están muy especializados (son expertos en dominios muy concretos). En esta situación tendremos muchas habilidades muy específicas en el proyecto. Cada tarea requerirá muchas de estas habilidades y los empleados tendrán unas pocas habilidades cada uno. En el sentido opuesto, si los empleados tienen algunas nociones de varios temas, tendremos unas pocas habilidades asociadas con vastos dominios. Por tanto, el número de habilidades requeridas por las tareas en este caso será menor que en el escenario anterior.

Para finalizar esta sección profundizaremos en el significado de la restricción **R2** (relacionada con las habilidades). Obsérvese que si una tarea requiere una habilidad, la restricción demanda que al menos un empleado que trabaje en esa tarea, no necesariamente todos ellos, tengan dicha habilidad. Esto tiene sentido en algunas situaciones, por ejemplo cuando la habilidad es la capacidad de liderazgo y la tarea requiere un único líder. Es posible que un empleado que trabaje en la tarea no tenga ninguna habilidad. De este modo podemos modelar escenarios donde algunos empleados no tienen la habilidad requerida para la tarea en la que trabajan, pero están en contacto con otros empleados que tienen dichas habilidades y pueden aprenderlas si es necesario. No obstante, en algunos escenarios necesitamos que toda la gente que trabaje en la tarea tenga las habilidades requeridas. Por ejemplo, volviendo a nuestra aplicación bancaria, podemos exigir que todos los empleados que implementen la aplicación (t_3) tengan experiencia en programación (lo cual es algo más que razonable). Para abordar este escenario asignamos un grado de dedicación cero en la tarea a todos aquellos empleados sin la habilidad requerida. Siguiendo con el ejemplo, podemos hacer $x_{i3} = 0.0$ para $i \in \{2, 3, 5\}$ (empleados e_2, e_3 y e_5). Los elementos de la matriz solución con un valor cero impuesto de esta forma no se consideran cuando se resuelve el problema usando algún algoritmo de optimización, reduciendo de este modo el número de variables del problema. No obstante, para evaluar una solución se introducirá un cero en las posiciones correspondientes de la matriz.

3.1.2. Experimentos *in silico*

Tal y como se ha definido el problema, su resolución resulta útil para que el gestor de proyectos software pueda estudiar distintos escenarios y realizar simulaciones simplificadas dentro de un ordenador. Todos los elementos considerados de un proyecto software han sido caracterizados dentro del problema mediante un modelo matemático. Los experimentos que se pueden realizar con este tipo de formulación son denominados *in silico* de acuerdo a la taxonomía propuesta por Travassos y Barros [273]. Su denominación atiende al hecho de que tanto los sujetos involucrados en el proyecto (personal) como las tareas y recursos no humanos del mismo son modelados dentro de un ordenador.

De acuerdo a la taxonomía de Travassos y Barros existen otros tres tipos de experimentos dentro del dominio de la Ingeniería del Software empírica: *in vivo*, *in vitro* e *in virtuo*. Los experimentos *in vivo* son los que se llevan a cabo en circunstancias reales mientras se realiza un proyecto software. Los experimentos *in vitro* se realizan en entornos controlados. La mayoría de estos experimentos se realizan en universidades o con grupos pequeños dentro de compañías software. En los experimentos *in virtuo*, parte de los elementos del proyecto son virtuales y se encuentran modelados dentro de un ordenador. Los participantes interaccionan con los modelos computerizados.

Las categorías de experimentos *in virtuo* e *in silico* tienen sentido desde el uso de modelos matemáticos computerizados para simular parte o todo el proyecto software. De hecho, la aportación de Travassos y Barros fue la incorporación de dichas categorías a la tradicional clasificación de experimentos *in vivo*/*in vitro* usada en el campo de la Ingeniería del Software empírica. De acuerdo a estos autores, el uso de experimentos *in silico* es poco habitual en Ingeniería del Software, hecho que dota de mayor importancia a la aportación que realizamos en esta tesis al analizar la aplicación de técnicas metaheurísticas al problema de planificación de proyectos software anteriormente definido.

3.1.3. Trabajos relacionados

Existen trabajos que proponen y discuten técnicas de gestión avanzadas [38, 236] y herramientas [174, 179] que pueden ayudar a los gestores de proyectos software en su trabajo. Las computadoras se aplican

normalmente en varias etapas del proceso de gestión software. Podemos encontrar sistemas expertos para diagnosticar problemas en el desarrollo software [230], redes neuronales para decidir cuándo entregar el software a los usuarios [82], algoritmos genéticos para planificación software [54], herramientas CASE para la gestión del conocimiento del desarrollo software [135], todos ellos formando un nuevo campo de conocimiento relacionado con la gestión de proyectos asistida por computador.

El problema de planificación de proyectos que definimos en la Sección 3.1.1 está relacionado con la planificación de proyectos con restricción de recursos (*Resource-Constrained Project Scheduling*, RCPS), un problema que ha sido frecuentemente tratado en la literatura y que se ha resuelto con técnicas exactas [75, 209, 265] y metaheurísticas [139, 204, 223]. No obstante, hay algunas diferencias entre PSP y RCPS. En primer lugar, en PSP hay un coste asociado con los empleados y un coste de proyecto que debe ser minimizado además de la duración del proyecto; mientras que en RCPS el objetivo es minimizar únicamente la duración del proyecto (no se considera el coste económico). Es más, en RCPS hay varios tipos de recursos, mientras que en el caso de PSP sólo existe uno (el empleado) con varias posibles habilidades. Debemos destacar que el papel de las habilidades en PSP es diferente del de los tipos de recurso en RCPS. Además, cada actividad en RCPS requiere diferentes cantidades de cada recurso, mientras que las habilidades de PSP no son entidades cuantificables. El problema tal y como se define aquí es más adecuado que RCPS para modelar un proyecto software real, ya que incluye el concepto de empleado con un salario y habilidades personales capaz de realizar varias tareas durante una jornada laboral.

3.2. Generación automática de casos de prueba

Desde el principio de la Informática, los ingenieros están interesados en técnicas que permitan conocer si un módulo software cumple una serie de requisitos (la especificación). El software actual es muy complejo y estas técnicas se han convertido en una necesidad en muchas compañías de software. La *verificación formal* constituye un ejemplo de estos métodos donde las propiedades del software se demuestran como si de un teorema matemático se tratase. Una lógica muy conocida usada en esta verificación es la *lógica de Hoare* [140]. No obstante, la verificación formal usando lógicas no es completamente automática. Aunque los demostradores automáticos de teoremas pueden ayudar en el proceso, es necesaria aún la intervención humana. Otra técnica bien conocida y completamente automática es *model checking* [58], usada especialmente en el caso de sistemas concurrentes. En este caso todos los posibles estados del modelo se analizan (de un modo directo o indirecto) para demostrar (o negar) que satisface una determina propiedad. Describiremos esta técnica con detalle en la Sección 3.3, ya que constituye la base de nuestro tercer problema abordado.

Sin duda, la técnica más popular para testar el software es ejecutarlo con un conjunto de casos de prueba (*software testing*). El ingeniero selecciona un conjunto de configuraciones iniciales para el programa, lo que se denomina conjunto de *casos de prueba*, y comprueba su comportamiento con todos ellos. Si el comportamiento es el esperado, el programa pasa la fase de pruebas, ya que no hay evidencia de que contenga errores. Puesto que el tamaño del conjunto de casos de prueba es una decisión del ingeniero, puede controlar el esfuerzo dedicado a esta tarea. Para asegurar la corrección del software con esta técnica sería necesario ejecutarlo con todas las posibles configuraciones iniciales y en todos los posibles entornos, pero en la práctica esto suele ser inviable y la alternativa consiste en probar el programa con un conjunto representativo de casos de prueba. Esta es una tarea muy importante, costosa en tiempo y dura del desarrollo software [21, 208, 217]. La *generación automática de casos de prueba* (*automatic software testing*) consiste en proponer de forma automática un conjunto adecuado de casos de

prueba para un programa: el *programa objeto*[3]. Esto supone una forma de liberar a los ingenieros de la labor de seleccionar un conjunto adecuado de casos de prueba para probar el programa. Este proceso de automatización exige definir con precisión qué es un "conjunto adecuado" de casos de prueba. Para esto se recurre al concepto de *criterio de adecuación*, el cual es una condición que debe cumplir el conjunto de casos de prueba para ser considerado "adecuado". Existen distintos criterios de adecuación definidos en la literatura. Formalizaremos algunos de ellos en la Sección 3.2.1.

3.2.1. Definición del problema

Antes de formalizar el problema necesitamos aclarar la notación y los conceptos usados en la formalización. Tras esto, definiremos las *medidas de cobertura* de programas más populares en la literatura y presentaremos algunos resultados teóricos relacionados con ellas. Los criterios de adecuación suelen estar basados en algunas de estas medidas de cobertura. Por último, definiremos el problema de generación automática de casos de prueba que abordamos en esta tesis. Estamos interesados aquí únicamente en programas escritos con un lenguaje de programación imperativo y con un comportamiento determinista. Esto último significa que asumimos una relación funcional entre el conjunto de las posibles entradas al programa (casos de prueba) y las ejecuciones del mismo.

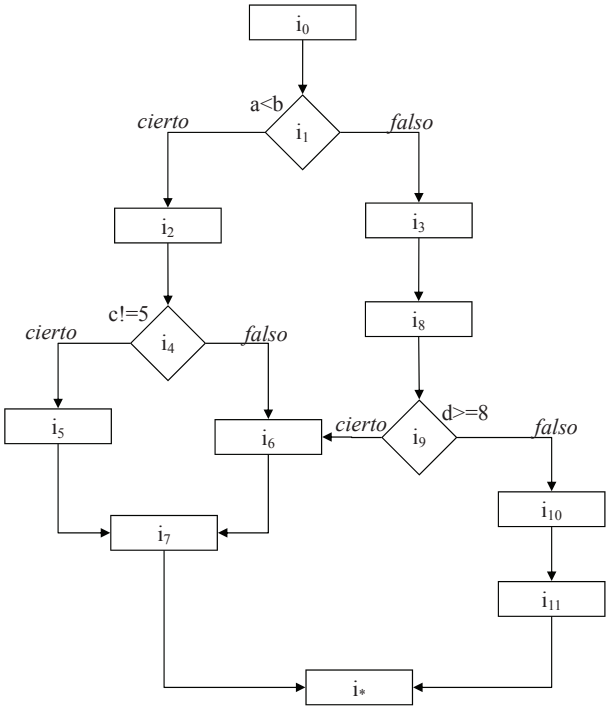


Figura 3.5: Grafo de control de flujo de un programa.

[3]Usamos el término *programa objeto* para referirnos al programa para el que generamos los casos de prueba (programa objeto de la prueba). No debe confundirse con el fichero que contiene el *código objeto* de un programa compilado. A este fichero lo llamamos *fichero objeto*.

El *grafo de control de flujo* de un programa P (véase la Figura 3.5) es un grafo orientado $G_P = (I, T)$ en el que I es el conjunto de las instrucciones del programa (nodos) y T es el conjunto de pares de instrucciones tal que $(i, j) \in T$ si tras la ejecución de la instrucción i puede ejecutarse inmediatamente la instrucción j[4]. En dicho grafo existe un nodo y sólo uno con grado de entrada cero que denotaremos con i_0, $ge(i_0) = 0$. Éste es el nodo inicial del grafo y representa la primera instrucción del programa. Para todo nodo i del grafo existe un camino orientado que parte del nodo inicial y termina en él. Esto se corresponde con el hecho de que cualquier instrucción del programa es alcanzable *a priori*. Asumimos, por tanto, que no tratamos con programas que poseen instrucciones no alcanzables, como instrucciones detrás de un bucle infinito, instrucciones detrás de un `return`, etc. Si así fuera, bastaría con eliminar tales instrucciones para obtener un programa semánticamente equivalente sin instrucciones no alcanzables. Distinguimos en I un nodo especial $i_* \in I$ que no se corresponde con ninguna instrucción y que representa la terminación del programa. Este nodo tendrá grado de salida cero, $gs(i_*) = 0$. Llamaremos $\mathcal{V}_P$ al conjunto de las variables del programa P. Cada variable $v_i \in \mathcal{V}_P$ toma valores en un dominio D_i con $i = 1, \ldots, |\mathcal{V}_P|$. Distinguimos una variable especial $pc \in \mathcal{V}_P$ que será el contador de programa y tomará valores en el conjunto I de nodos del grafo de control de flujo. Un *estado* ρ del programa P es una aplicación que asigna a cada variable del programa un valor de su dominio:

$$\begin{array}{l} \rho : \mathcal{V}_P \to \bigcup_{i=1}^{|\mathcal{V}_P|} D_i \\ v_i \mapsto \rho(v_i) \in D_i \ . \end{array} \tag{3.10}$$

Diremos que un estado ρ es *de terminación* si $\rho(pc) = i_*$. Asimismo, diremos que un estado ρ es *inicial* si $\rho(pc) = i_0$. Denotaremos el conjunto de todos los posibles estados de un programa con $\mathcal{E}_P$. Asumimos la existencia de un *transformador de estados* $\mathcal{S}_P : \mathcal{E}_P \nrightarrow \mathcal{E}_P$. Esta función asocia a cada estado ρ en que está definida el estado ρ' que resulta tras ejecutar la instrucción $\rho(pc)$. Supondremos que el transformador de estados no está definido en ningún estado determinación.

Definimos una *ejecución* del programa P a partir del estado inicial ρ y lo denotamos con $Ejec_P(\rho)$ como una sucesión de estados $\{\rho_n\}$ donde $\rho = \rho_1$ y para todo $i \geq 1$ se tiene $\rho_{i+1} = \mathcal{S}_P(\rho_i)$. Cada ejecución $\{\rho_n\}$ del programa P determina un camino en el grafo de control de flujo G_P del programa. Este camino está formado por una sucesión de nodos $\{s_n\}$ donde cada elemento $s_i \in I$ se define como $s_i = \rho_i(pc)$. Llamaremos a esta sucesión de nodos $\{s_n\}$ determinada por la sucesión de estados $\{\rho_n\}$ *proyección de la ejecución en el grafo de control de flujo* y lo denotaremos con $Proy_P(\{\rho_n\})$. Denotaremos con $Ins_P(\{s_n\})$ el conjunto de elementos de I que aparecen en la sucesión $\{s_n\}$. Un *caso de prueba* de un programa P es un estado inicial ρ del mismo.

Definición 1 (Cobertura de instrucciones)**.** *Dado un conjunto C de casos de prueba para un programa P, definimos la* cobertura de instrucciones *de C, $cobIns_P(C)$, como el cociente entre las instrucciones ejecutadas en las ejecuciones del programa que parten de los estados de C y el número total de instrucciones del programa, esto es,*

$$cobIns_P(C) = \frac{\left|\bigcup_{\rho \in C} Ins_P(Proy_P(Ejec_P(\rho))) / \{i_*\}\right|}{|I / \{i_*\}|} \ . \tag{3.11}$$

Ahora estamos en disposición de definir el *criterio de adecuación de cobertura de instrucciones*, que establece que un conjunto de casos de prueba C para un programa P es adecuado cuando $cobIns_P(C) = 1$.

[4]Existen definiciones más completas de grafo de control de flujo en la literatura (véase [288]), pero la que se presenta en este capítulo es suficiente para los objetivos que perseguimos.

Las instrucciones condicionales para el control de flujo (como `if-then-else`, `switch`, `for` o `while`) se caracterizan en el grafo de control de flujo por tener grado de salida mayor que uno. Llamaremos CCF_P al conjunto de nodos del grafo de control de flujo que representan este tipo de instrucciones. Esto es, $CCF_P = \{i \in I | gs(i) > 1\}$. Diremos que un arco del grafo de control de flujo (i, j) es una *rama* del programa P si el nodo origen del arco pertenece al conjunto CCF_P. Denotamos con B_P el conjunto de ramas del programa P, es decir,

$$B_P = \{(i, j) \in T | gs(i) > 1\} \quad , \tag{3.12}$$

y con $Ram_P(\{s_n\})$ el conjunto de ramas de que aparecen de forma implícita en la sucesión de nodos $\{s_n\}$:

$$Ram_P(\{s_n\}) = \{(i, j) \in B_P | \exists k \geq 1, i = s_k \wedge j = s_{k+1}\} \quad . \tag{3.13}$$

Definición 2 (Cobertura de ramas)**.** *Dado un conjunto C de casos de prueba para un programa P, definimos la* cobertura de ramas *de C, $cobRam_P(C)$, como el cociente entre las ramas recorridas en las ejecuciones del programa que parten de los estados de C y el número de ramas del programa, es decir,*

$$cobRam_P(C) = \frac{\left|\bigcup_{\rho \in C} Ram_P(Proy_P(Ejec_P(\rho)))\right|}{|B_P|} \quad . \tag{3.14}$$

El criterio de adecuación de cobertura de ramas establece que un conjunto de casos de prueba C para un programa P es adecuado cuando $cobRam_P(C) = 1$.

Ya hemos introducido la notación suficiente para dar el siguiente resultado, que es bien conocido en el dominio de la generación de casos de prueba.

Proposición 1. *Sea un programa P cualquiera y sea C un conjunto de casos de prueba para P. Si $cobRam_P(C) = 1$ entonces $cobIns_P(C) = 1$. Es decir, una cobertura total de ramas implica una cobertura total de instrucciones.*

Demostración. La demostración es sencilla partiendo del hecho de que dos estados sucesivos de una ejecución se proyectan en nodos del grafo de control de flujo que son sucesor uno del otro. Dada una instrucción cualquiera $i \in I/\{i_*\}$, se busca un camino en el grafo de control de flujo que comience en el nodo inicial i_0 y termine en i. En dicho camino se busca hacia atrás el primer nodo con grado de salida mayor que dos; esto determina una rama a tomar. Finalmente, el estado inicial $\rho \in C$ que ejecuta dicha rama ejecutará la instrucción i. Como esto puede hacerse para cada instrucción del conjunto $I/\{i_*\}$ de instrucciones, la cobertura de instrucciones del conjunto C será uno. ■

La decisión de qué rama tomar cuando una ejecución de un programa P alcanza una instrucción $i \in CCF_P$ se hace de acuerdo al valor de una expresión asociada a la instrucción i. A partir de ahora consideraremos una versión del grafo de control de flujo etiquetada por dos funciones $lIns_P : CCF_P \rightarrow Expr$ y $lRam_P : B_P \rightarrow Val$, donde $Expr$ es el conjunto de todas las expresiones que pueden formarse en el lenguaje en que está implementado P y Val es el conjunto de todos los valores que pueden tomar las expresiones de $Expr$. La primera función, $lIns_P$, etiqueta cada instrucción condicional de control de flujo con la expresión que ha de evaluarse para determinar la rama a escoger. La segunda, $lRam_P$, etiqueta cada rama con el valor que debe tener la expresión asociada a la instrucción origen para que la rama sea tomada. Cuando la expresión asociada a una instrucción toma valores lógicos, la expresión puede denominarse también *decisión*. Esta expresión lógica estará formada por una combinación de predicados atómicos sobre las variables del programa unidos mediante operadores lógicos de acuerdo a la gramática de la Figura 3.6.

```
BoolExpr :=  AtPred |
             BoolExpr 'AND' BoolExpr |
             BoolExpr 'OR' BoolExpr |
             'NOT' BoolExpr |
             '(' BoolExpr ')'
AtPred :=    NumExpr < NumExpr |
             NumExpr > NumExpr |
             Numexpr >= NumExpr |
             numExpr <= NumExpr |
             NumExpr == NumExpr |
             NumExpr != NumExpr
```

Figura 3.6: Gramática BNF para las expresiones lógicas y los predicados atómicos. `BoolExpr` y `NumExpr` son expresiones lógicas y numéricas, respectivamente.

A partir de ahora asumimos que trabajamos sólo con programas en los que las instrucciones condicionales de control de flujo tienen asociadas solamente expresiones lógicas (decisiones), y por tanto, sólo habrá dos ramas con origen en ellas, la rama asociada al valor *cierto* y la asociada al valor *falso*. Esto no representa una limitación importante, ya que siempre es posible transformar un programa en otro equivalente en el que sólo se utilizan expresiones lógicas para decidir una rama. Por otro lado, la mayoría de las instrucciones condicionales de control de flujo de los programas imperativos más populares basan su decisión en la evaluación de una expresión lógica. A modo de ilustración diremos que en los lenguajes, C, C++ y Java existe tan sólo una instrucción que necesita evaluar una expresión numérica: `switch`.

Un tercer criterio de adecuación consiste en hacer que todos los predicados atómicos tomen los dos valores lógicos: *cierto* y *falso*. Cada predicado atómico de una expresión lógica lo identificaremos con el par (j,k), donde $j \in CCF_P$ es el nodo del grafo de control de flujo en el que aparece y k es un número natural que indica su posición en la expresión lógica $lIns_P(j)$ asociada a dicho nodo. Este número k variará entre 1 y $Npred(lIns_P(j))$, que es el número de predicados atómicos de la expresión lógica. La numeración de los predicados atómicos se realizará de izquierda a derecha en la cadena lineal de la expresión o, equivalentemente, de acuerdo al recorrido en inorden del árbol correspondiente a la expresión. Denotaremos con $Pred_P$ el conjunto de todas las referencias a los predicados atómicos del programa P, lo que escrito formalmente es

$$Pred_P = \{(j,k) | j \in CCF_P, 1 \leq k \leq Npred(lIns_P(j))\} \quad . \tag{3.15}$$

Abusando de la notación, denotaremos con $\rho((j,k))$ el valor lógico que el estado ρ asigna al predicado atómico referenciado por (j,k). Entonces, el conjunto de pares (predicado atómico, valor lógico) que se cubren en una ejecución que comienza en el estado ρ es

$$Cond_P(\rho) = \{((j,k),b) | (j,k) \in Pred_P, \exists \rho' \in Ejec_P(\rho) \bullet (\rho'(pc) = j, \rho'((j,k)) = b)\} \quad . \tag{3.16}$$

Definición 3 (Cobertura de condiciones)**.** *Dado un conjunto C de casos de prueba para un programa P, definimos la* cobertura de condiciones[5] *de C, $cobCon_P(C)$, como el cociente entre los pares (predicado*

[5]Mantenemos el nombre de "cobertura de condiciones" por la tradición existente en este dominio de llamar condición a los predicados atómicos.

atómico, valor lógico) cubiertos en las ejecuciones del programa que parten de los estados de C y el número total de pares, es decir,

$$cobCon_P(C) = \frac{\left|\bigcup_{\rho \in C} Cond_P(\rho)\right|}{2 \cdot |Pred_P|} \ . \tag{3.17}$$

El criterio de adecuación de cobertura de condiciones establece que un conjunto C de casos de prueba para un programa P es adecuado cuando $cobCon_P(C) = 1$. La cobertura de condiciones requiere cubrir un mayor número de elementos que la cobertura de ramas, ya que $2 \cdot |Pred_P| \geq |B_P|$. Podría parecer, por tanto, que una cobertura total de condiciones implica la cobertura total de ramas. Sin embargo, en general, esto no es cierto. Para comprobarlo basta analizar la expresión `(a>0 && b<1)`. Si empleamos los casos de prueba (`a`=0,`b`=0) y (`a`=1,`b`=1) conseguimos cobertura total de condiciones con ellos pero no conseguimos cobertura total de ramas, ya que la expresión lógica es falsa en ambos casos de prueba. Por este motivo se define en la literatura otro criterio de adecuación que exige cumplir simultáneamente la cobertura de condiciones y de ramas.

Definición 4 (Cobertura de condiciones-decisiones)**.** *Dado un conjunto C de casos de prueba para un programa P, definimos la* cobertura de condiciones-decisiones *del siguiente modo:*

$$cobConDec_P(C) = \frac{\left|\bigcup_{\rho \in C} Ram_P(Proy_P(Ejec_P(\rho)))\right| + \left|\bigcup_{\rho \in C} Cond_P(\rho)\right|}{|B_P| + 2 \cdot |Pred_P|} \ . \tag{3.18}$$

Existe un caso en el que la cobertura de condiciones implica a la cobertura de ramas y, por tanto, a la cobertura de condiciones-decisiones. Esto ocurre cuando los operadores lógicos utilizados en el programa se *evalúan en cortocircuito*. La definición de cobertura de condiciones cambia en este caso. Para definirla, acudiremos a la ayuda de una función auxiliar que asocia a cada par (expresión lógica, estado) el conjunto de los pares (predicado, valor lógico) que se cubren en la evaluación de la expresión. Esta función, $CeSc_P$, se define recursivamente como sigue:

$$CeSc_P((j,k),\rho) = \{((j,k),\rho((j,k)))\} \text{ para } (j,k) \in Pred_P \ , \tag{3.19}$$

$$CeSc_P(\text{NOT } expr,\rho) = CeSc_P(expr,\rho) \ , \tag{3.20}$$

$$CeSc_P(expr1 \text{ AND } expr2,\rho) = \begin{cases} CeSc_P(expr1,\rho) & \text{si } \rho(expr1) = falso \\ CeSc_P(expr1,\rho) \cup CeSc_P(expr2,\rho) & \text{si } \rho(expr1) = cierto \end{cases} , \tag{3.21}$$

$$CeSc_P(expr1 \text{ OR } expr2,\rho) = \begin{cases} CeSc_P(expr1,\rho) & \text{si } \rho(expr1) = cierto \\ CeSc_P(expr1,\rho) \cup CeSc_P(expr2,\rho) & \text{si } \rho(expr1) = falso \end{cases} . \tag{3.22}$$

Con ayuda de esta función auxiliar se define el conjunto de pares (predicado, valor) cubierto en el programa P a partir del estado inicial ρ de la siguiente forma:

$$CondSc_P(\rho) = \{((j,k),b) | \exists \rho' \in Ejec_P(\rho) \bullet (\rho'(pc) = j, j \in CCF_P, ((j,k),b) \in CeSc_P(lIns_P(j),\rho'))\} \ . \tag{3.23}$$

Definición 5 (Cobertura de condiciones en programas con evaluación en cortocircuito)**.** *Dado un conjunto C de casos de prueba para un programa P implementado usando operadores lógicos con evaluación en cortocircuito, definimos la* cobertura de condiciones *del siguiente modo:*

$$cobConSc_P(C) = \frac{\left|\bigcup_{\rho \in C} CondSc_P(\rho)\right|}{2 \cdot |Pred_P|} \ . \tag{3.24}$$

A continuación demostraremos tres enunciados que formalizan lo que anteriormente hemos solamente comentado: la cobertura de condiciones implica a la de ramas en programas con evaluación en cortocircuito de expresiones lógicas.

Proposición 2. *Sea P un programa escrito usando operadores lógicos con evaluación en cortocircuito y* $expr = lIns_P(j)$ *la expresión lógica asociada a la instrucción* $j \in CCF_P$*. Sea el par* (j,k) *el último predicado atómico que aparece en la expresión expr, esto es,* $k = Npred(expr)$ *y sean* σ_1 *y* σ_2 *dos estados iniciales del programa tales que existen* $\sigma_1' \in Ejec_P(\sigma_1)$ *y* $\sigma_2' \in Ejec_P(\sigma_2)$ *que evalúan el predicado* (j,k) *y además,* $\sigma_1'((j,k)) \neq \sigma_2'((j,k))$*. Entonces, las dos ramas que parten de la instrucción j son cubiertas por el conjunto de estados iniciales* $\{\sigma_1, \sigma_2\}$*.*

Demostración. Para probar esto, basta con demostrar que dada una expresión lógica *expr*, si ρ_1 y ρ_2 son dos estados que evalúan en cortocircuito el último predicado atómico de *expr* y además producen valores diferentes, entonces $\rho_1(expr) \neq \rho_2(expr)$. Para llevar a cabo la demostración, procederemos por inducción sobre la estructura de la expresión *expr*.

- Caso base: Si la expresión está formada por un único predicado el enunciado es trivialmente cierto. El valor de la expresión coincide con el valor del predicado y, por tanto, los estados ρ_1 y ρ_2 asignarán valores *cierto* y *falso* a la expresión, tomándose ambas ramas.
- Paso inductivo: Sea n el número de operadores lógicos que aparecen en *expr*. Suponemos que el enunciado es cierto en el caso de expresiones lógicas con menos de n operadores lógicos. A continuación identificamos tres casos posibles en función del operador raíz en *expr*:

  1. $expr =$ NOT $expr'$: en este caso, el predicado (j,k) es el último de $expr'$, la cual tiene menos operadores lógicos que *expr*. De acuerdo con la hipótesis de inducción, $\rho_1(expr') \neq \rho_2(expr')$ y, por tanto, $\rho_1(expr) \neq \rho_2(expr)$.
  2. $expr = expr_1$ AND $expr_2$: el predicado (j,k) es el último de $expr_2$ y, puesto que la evaluación es en cortocircuito, se debe cumplir $\rho_1(expr_1) = \rho_2(expr_1) = cierto$[6] (si no fuera así no se habría alcanzado el predicado (j,k)). Por otro lado, de acuerdo con la hipótesis de inducción, tenemos $\rho_1(expr_2) \neq \rho_2(expr_2)$, lo que finalmente implica $\rho_1(expr) \neq \rho_2(expr)$.
  3. $expr = expr_1$ OR $expr_2$: este caso se demuestra de forma idéntica al anterior, teniendo en cuenta, esta vez, que se debe cumplir $\rho_1(expr_1) = \rho_2(expr_1) = falso$.

■

[6] Obsérvese que esta igualdad es la responsable de que la cobertura de condiciones implique a la de ramas en los lenguajes con evaluación en cortocircuito.

Una consecuencia directa de esta proposición es la siguiente:

Corolario 1. *La cobertura de condiciones implica la cobertura de ramas en programas cuyos operadores lógicos se evalúen en cortocircuito.*

Demostración. Sea P un programa escrito usando operadores lógicos con evaluación en cortocircuito y C un conjunto de casos de prueba que cubre todos los predicados atómicos del programa, esto es, $cobConSc_P(C) = 1$. El conjunto C cubrirá todos los últimos predicados de todas las expresiones del programa y, de acuerdo con la Proposición 2, todas las ramas serán cubiertas, esto es, $cobRam_P(C) = 1$. ■

Corolario 2. *La cobertura de condiciones implica a la cobertura de condiciones-decisiones en programas cuyos operadores lógicos se evalúen en cortocircuito.*

Demostración. De acuerdo a las definiciones previas, un conjunto de casos de prueba C cumple la cobertura de condiciones-decisiones si y sólo si C cumple simultáneamente la cobertura de condiciones y la cobertura de ramas. De acuerdo al Corolario 1, en programas que usen solamente operadores lógicos con evaluación en cortocircuito, la cobertura de condiciones implica a la de ramas. Así pues, se sigue que en estos programas la cobertura de condiciones implica a la cobertura de condiciones-decisiones. ■

Con todo lo anterior, podemos enunciar el problema de la generación de casos de prueba como un problema de optimización en el que el objetivo es encontrar un conjunto de casos de prueba que maximice la cobertura considerada (de instrucciones, de ramas, de condiciones, de condición-decisión). Nosotros trabajamos con programas objeto implementados en lenguaje C (que realiza evaluación en cortocircuito de expresiones lógicas) y consideramos como criterio de adecuación la cobertura de condiciones porque implica a las demás. Así pues, podemos formalizar el problema abordado como sigue.

Definición 6 (Problema de generación de casos de prueba con cobertura de condiciones)**.** *Dado un programa P escrito usando únicamente operadores lógicos con evaluación en cortocircuito, el problema de generación de casos de prueba que planteamos consiste en encontrar un conjunto de casos de prueba C que maximice $cobConSc_P(C)$.*

3.2.2. Trabajos relacionados

La generación automática de casos de prueba para los programas de ordenador ha sido y sigue siendo estudiada en la literatura desde hace décadas [59, 207]. Podemos distinguir cuatro grandes paradigmas dentro de la generación automática de casos de prueba: el *paradigma estructura*l (*structural software testing*), el *paradigma funcional* (*functional software testing*), el *paradigma de caja gris* (*grey-box software testing*), y el *paradigma no funcional* (*non-functional software testing*) [196].

En el paradigma estructural [112, 190, 199, 206, 249] el generador de casos de prueba usa información de la estructura del programa para guiar la búsqueda de casos de prueba (por esta razón se conoce también como paradigma de caja blanca). Normalmente, esta información se toma del *grafo de control de flujo* del programa. El objetivo es conseguir un conjunto de casos de prueba que ejecute todos los elementos considerados del programa (instrucciones, ramas, predicados atómicos, etc.). En el paradigma funcional [49, 50] la información que se usa es una especificación del comportamiento del programa. El objetivo es comprobar que el software se comporta exactamente como se especifica sin usar ninguna información de la estructura interna del programa (por esto se conoce también como paradigma de caja negra). El paradigma de caja gris [163] es una combinación de los dos anteriores (caja blanca y caja

negra). Usa información estructural y funcional para generar los casos de prueba. Por ejemplo, se pueden introducir asertos en el código fuente para comprobar el comportamiento funcional, mientras que se usan técnicas propias del paradigma estructural para generar los casos de prueba que violan dichos asertos. Finalmente, en el paradigma no funcional [260, 287] el objetivo es comprobar cualquier otro aspecto del programa que no esté relacionado con su comportamiento funcional. Algunos ejemplos son la usabilidad, la portabilidad, el uso de memoria, la eficiencia, etc.

Centrándonos en el paradigma estructural, que es el que usamos en nuestro planteamiento del problema, podemos encontrar varias alternativas en la literatura. En la *generación aleatoria*, los casos de prueba se crean de forma completamente aleatoria hasta que se satisface el criterio de adecuación o se genera un número máximo de casos de prueba. Podemos encontrar experimentos con generación aleatoria en [36] y más recientemente en [206]. La *generación simbólica* de casos de prueba [59] consiste en asignar valores simbólicos a las variables en lugar de valores concretos para crear una *ejecución simbólica*. De esta ejecución simbólica se pueden extraer restricciones algebraicas que se pueden usar para encontrar casos de prueba. Godzilla [221] es un generador automático de casos de prueba que usa esta técnica. Un tercer enfoque (ampliamente utilizado) es la *generación dinámica* de casos de prueba. En este caso, se añaden una serie de instrucciones al programa para pasar información al generador de casos de prueba. El generador comprueba si se cumple el criterio de adecuación o no. Si el criterio de adecuación no se cumple genera nuevos casos de prueba que se usarán para probar el programa. El proceso de generación de casos de prueba se traduce así a un problema de minimización de una función que es algún tipo de "distancia" a una ejecución ideal donde el criterio de adecuación se cumple. Este paradigma se presentó en [207] y desde entonces muchos trabajos se han basado en él [151, 164, 206, 287]. Incluso existen técnicas híbridas combinando la ejecución concreta y simbólica con muy buenos resultados. Este es el caso de las herramientas DART [112] y CUTE [249].

Dentro de la generación dinámica de casos de prueba, está tomando especial relevancia el uso de algoritmos evolutivos hasta el punto de que se ha acuñado el término *evolutionary testing* para referirse a ello. Mantere y Alander en [192] presentan una reciente revisión de la aplicación de algoritmos evolutivos a la generación de casos de prueba. La mayoría de los trabajos incluidos en su revisión usan algoritmos genéticos (GA) como motor de búsqueda. De hecho, sólo unos pocos trabajos listados en la revisión incluyen otras técnicas como búsqueda basada en gradiente [267] y enfriamiento simulado [270].

Si ampliamos el conjunto de técnicas aplicadas a las metaheurísticas, podemos encontrar trabajos recientes como [78], donde los autores explican cómo usar un algoritmo de búsqueda tabú para generar casos de prueba obteniendo cobertura de ramas máxima. Sagarna y Lozano abordan el problema usando algoritmos de estimación de distribuciones (EDA) en [244], y comparan los resultados con una búsqueda dispersa (SS) en [246].

3.3. Búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes

Model checking[7] [58] es una técnica automática que permite comprobar si un determinado sistema concurrente satisface una propiedad dada, como por ejemplo, la ausencia de interbloqueos (*deadlocks*) y posposición indefinida (*starvation*), el cumplimiento de invariantes, etc. Su uso es necesario cuando se desarrolla software que controla sistemas críticos, como los controladores de un avión, o una nave

[7] El término *model checking* suele traducirse por *comprobación de modelos* y *model checker* por *comprobador de modelos*. Sin embargo, usaremos en esta tesis los términos ingleses escritos en cursiva.

espacial. A diferencia de la generación de casos de prueba, *model checking* es una técnica formal que puede demostrar que el software cumple la propiedad especificada. Sin embargo, la memoria requerida para realizar tal verificación suele crecer exponencialmente con el tamaño del modelo a verificar. A esto se le conoce como *problema de la explosión de estados* y limita el tamaño de los sistemas concurrentes que se pueden verificar.

En lugar de aplicar *model checking* al sistema concurrente directamente, se aplica a un modelo suyo, que será más pequeño, reduciendo así el problema de la explosión de estados. Hay varios lenguajes diseñados específicamente para modelar sistemas concurrentes, como por ejemplo, Promela (*Process Meta-Language*) [145], el lenguaje de SMV [195] o el lenguaje intermedio de SAL [35]. Una interesante excepción es la del *model checker* Java PathFinder [117], que usa el lenguaje Java para modelar los sistemas concurrentes y, en sus últimas versiones, trabaja directamente sobre ficheros de *bytecodes* de Java.

Existen varias técnicas desarrolladas para aliviar el problema de la explosión de estados. Éstas reducen la memoria necesaria para la búsqueda siguiendo diferentes enfoques. Por un lado están las técnicas que reducen la cantidad de estados a explorar para realizar la verificación como, por ejemplo, reducción de orden parcial [157] (véase la Sección 3.3.5) y reducción de simetría [57]. Por otro, tenemos las técnicas que reducen el espacio que ocupa un estado en memoria: compresión de estados [147], representación mínima de autómatas [148] y tablas hash de un bit [143].

El *model checking* simbólico (*symbolic model checking*) [51] es otra alternativa muy popular al *model checking* explícito que puede reducir la cantidad de memoria requerida para la verificación. En este caso se usa una estructura de datos muy eficiente, el *diagrama de decisión binaria ordenado* (*Ordered Binary Decision Diagram*, OBDD) para almacenar conjuntos de estados. Estos estados quedan caracterizados por una fórmula proposicional finita que se representa con un OBDD. No obstante, las técnicas de búsqueda exhaustiva siempre tienen problemas para verificar sistemas concurrentes reales porque la mayoría de estos programas son demasiado complejos incluso para las técnicas más avanzadas. Por esto, se necesitan técnicas de baja complejidad, como las metaheurísticas, para los programas de tamaño mediano y grande que se presentan en los escenarios reales.

3.3.1. Autómatas de Büchi

Antes de verificar un modelo determinado hay que transformarlo en una estructura a partir de la cual se pueda aplicar el algoritmo de *model checking*. En esta sección describimos la estructura que será utilizada para la formalización del problema en esta tesis: el *autómata de Büchi*. Otras estructuras utilizadas también en este contexto son las *estructuras de Kripke* y los *sistemas de transiciones*.

Un autómata de Büchi A es una tupla $A = \langle \Sigma, Q, \Delta, Q^0, F \rangle$ donde Σ es un alfabeto finito, Q es un conjunto finito de estados, $\Delta \in Q \times \Sigma \times Q$ es una relación de transición, $Q^0 \in Q$ es el conjunto de estados iniciales y $F \in Q$ es el conjunto de estados de aceptación. Sea $v \in \Sigma^\omega$ una palabra de longitud infinita sobre el alfabeto Σ, una *ejecución* de A sobre v es una función $\rho : \mathbb{N} \to Q$ tal que $\rho(0) \in Q^0$ y para todo $i \geq 0$ se tiene $(\rho(i), v(i), \rho(i+1)) \in \Delta$. Una ejecución ρ de A sobre una palabra infinita v es de aceptación si $inf(\rho) \cap F \neq \emptyset$, donde $inf(\rho)$ denota el conjunto de estados que aparecen un número infinito de veces en la ejecución ρ. En ese caso decimos que el autómata A acepta la palabra infinita v o que la palabra v está contenida en el lenguaje $\mathcal{L}(A$ reconocido por el autómata: $v \in \mathcal{L}(A)$.

Cuando un autómata de Büchi A se utiliza para modelar un sistema concurrente, todos los estados son de aceptación, esto es, $F = Q$. Por otro lado, sólo existe un estado inicial, $|Q^0| = 1$. El alfabeto está formado por todos los posibles subconjuntos de un conjunto de proposiciones atómicas AP (es decir, $\Sigma = 2^{AP}$) y para cada par de estados $s, s' \in Q$ existe a lo sumo una terna $(s, \alpha, s') \in \Delta$, donde $\alpha \subseteq AP$ es

el conjunto de proposiciones ciertas en el estado s'. Una ejecución ρ del autómata así definido representa una ejecución del sistema concurrente. Además, el conjunto de palabras aceptadas por el autómata $\mathcal{L}(A)$ es el conjunto de comportamientos del sistema concurrente.

3.3.2. Propiedades y lógicas temporales

Siguiendo con la terminología anterior, una *propiedad* es un conjunto $\mathcal{P}$ de palabras infinitas de Σ^ω. Si el lenguaje aceptado por un autómata A, $\mathcal{L}(A)$, está contenido en $\mathcal{P}$ el sistema cumple la propiedad. En caso contrario, existirá alguna ejecución del sistema que viola la propiedad, esto es, existe una palabra infinita $\nu \in \mathcal{L}(A)$ tal que $\nu \notin \mathcal{P}$. Para especificar las propiedades del sistema, se suele acudir al uso de alguna lógica temporal, como la lógica temporal lineal (LTL) [227] o las lógicas de árboles de computación (CTL y CTL*) [56].

La lógica LTL tiene la ventaja de que sus fórmulas pueden transformarse en autómatas de Büchi. Si φ es una fórmula LTL, llamaremos $\mathcal{A}(\varphi)$ al autómata que la representa. El lenguaje aceptado por este autómata $\mathcal{L}(\mathcal{A}(\varphi))$ contiene las palabras infinitas $\nu \in \Sigma^\omega$ que satisfacen la fórmula φ. Así pues, comprobar que un sistema concurrente con autómata de Büchi A satisface la propiedad especificada mediante la fórmula LTL φ se convierte en comprobar si es cierta la inclusión $\mathcal{L}(A) \subseteq \mathcal{L}(\mathcal{A}(\varphi))$, o equivalentemente, si $\mathcal{L}(A) \cap \overline{\mathcal{L}(\mathcal{A}(\varphi))} = \emptyset$.

Para terminar la presentación de las propiedades de un sistema concurrente, hablaremos de la clasificación de propiedades LTL. Existen diversas clasificaciones de las propiedades LTL en la literatura. En [189], Manna y Pnueli presentan una clasificación bastante detallada. Hay una clasificación que tiene especial relevancia para *model checking* explícito usando autómatas de Büchi. Esta clasificación, descrita por primera vez por Lamport en [171] y formalizada en [172] y [22] divide el conjunto de las propiedades LTL en dos subconjuntos: las propiedades de *seguridad* y las de *viveza*. Informalmente, las propiedades de seguridad son las que se utilizan para comprobar que nada malo va a pasar en el sistema, mientras que las de viveza se usan para comprobar que algo bueno sucede durante la ejecución. Formalmente, una propiedad $\mathcal{P}$ es de seguridad si para toda palabra infinita σ que no la satisface existe un prefijo finito σ_i cuyas extensiones a ejecuciones infinitas no satisfacen la propiedad, esto es,

$$\forall \sigma \in \Sigma^\omega : \sigma \nvdash \mathcal{P} \Rightarrow (\exists i \geq 0 : \forall \beta \in \Sigma^\omega : \sigma_i \beta \nvdash \mathcal{P}) \ , \tag{3.25}$$

donde σ_i es un prefijo de σ formado por los primeros i símbolos de σ. Ejemplos de propiedades de seguridad son la ausencia de interbloqueo y los invariantes. Por otro lado, una propiedad P es de viveza si para toda palabra finita existe una extensión infinita que la satisface, esto es,

$$\forall \alpha \in \Sigma^* : \exists \beta \in \Sigma^\omega, \alpha\beta \vdash \mathcal{P} \ , \tag{3.26}$$

donde Σ^* es el conjunto de todas las palabras finitas formadas con Σ. Un ejemplo de propiedad de viveza es la ausencia de posposición indefinida.

No existen propiedades que sean a la vez de seguridad y de viveza, son conjuntos disjuntos de propiedades LTL. Además, cualquier propiedad LTL es de uno de estos tipos. Así pues, esta clasificación induce una partición en el conjunto de propiedades LTL. Existe una caracterización sintáctica de las propiedades de seguridad. Toda propiedad de seguridad puede expresarse mediante una fórmula LTL de la forma $\Box\gamma$ donde γ es una fórmula pasada (contiene únicamente operadores temporales pasados) [189].

3.3.3. *Model checking* LTL con autómatas de Büchi

Como mencionamos anteriormente, para comprobar si un sistema concurrente cumple una propiedad especificada mediante una fórmula LTL φ se debe calcular la intersección $\mathcal{L}(A) \cap \overline{\mathcal{L}(\mathcal{A}(\varphi))}$ y comprobar si es vacía. La operación de intersección de los lenguajes se traduce en una operación entre los autómatas implicados: la *intersección de autómatas de Büchi* $A \cap \overline{\mathcal{A}(\varphi)}$. El autómata de Büchi $\overline{\mathcal{A}(\varphi)}$ es el autómata asociado a la negación de la fórmula φ, es decir, $\overline{\mathcal{A}(\varphi)} = \mathcal{A}(\neg\varphi)$. Por lo tanto, para comprobar si el sistema cumple la propiedad basta con comprobar si el autómata $A \cap \mathcal{A}(\neg\varphi)$ acepta alguna palabra $\nu \in \Sigma^\omega$. Se demuestra que el lenguaje reconocido por un autómata es no vacío si y sólo si existen dos secuencias finitas de estados del autómata ρ_1 y ρ_2 tales que $\rho_1\rho_2^\omega$ es una ejecución de aceptación del autómata. La existencia de tales secuencias se comprueba buscando un camino (ρ_1) desde el estado inicial a un estado de aceptación s y un ciclo de estados (ρ_2) que incluya a s (véase la Figura 3.7). Si se encuentran, $\rho_1\rho_2^\omega$ es una ejecución del modelo que viola la propiedad especificada. En caso contrario, el modelo cumple la propiedad.

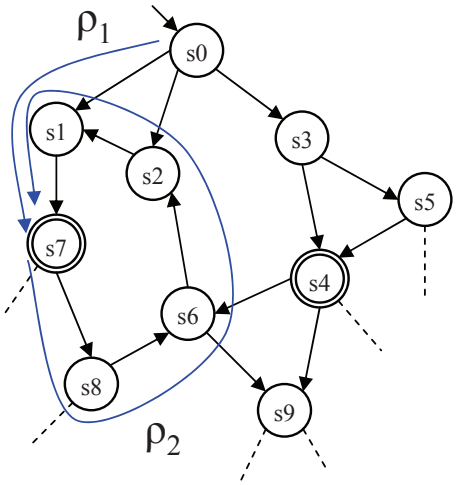


Figura 3.7: Autómata de Büchi intersección. Se puede apreciar una ejecución de aceptación con un ciclo que contiene al estado de aceptación *s7*.

No es necesario construir completamente el autómata intersección para comprobar una fórmula LTL: en lugar de eso, se construye conforme se va requiriendo durante la propia búsqueda del estado de aceptación. Esto se conoce como *model checking* "al vuelo" (*on-the-fly*), y permite ahorrar gran cantidad de memoria durante la verificación. Uno de los *model checkers* de estados explícitos más conocidos que sigue esta estrategia es SPIN [146], que comprueba propiedades especificadas mediante fórmulas LTL en modelos codificados en Promela.

Para verificar un modelo, SPIN usa el algoritmo de *búsqueda primero en profundidad anidada* (*Nested Depth First Search*, Nested-DFS) [144]. El algoritmo intenta encontrar primero un estado de aceptación usando *búsqueda primero en profundidad* (*Depth First Search*, DFS). Cuando lo encuentra, trata de alcanzar el mismo estado comenzando desde él mismo, es decir, busca un ciclo que contenga al estado de aceptación encontrado. Si dicho ciclo no se encuentra, busca otro estado de aceptación usando DFS

y repite el proceso. Nested-DFS es un algoritmo exhaustivo: si no encuentra un contraejemplo, éste no existe y el modelo cumple la propiedad.

Las propiedades de seguridad se pueden comprobar buscando tan sólo un estado de aceptación en el autómata de Büchi intersección. Es decir, cuando se comprueban propiedades de seguridad, no es necesario encontrar un ciclo adicional que contenga el estado de aceptación. Esto significa que la verificación de propiedades de seguridad se puede transformar en una búsqueda de un nodo objetivo (un estado de aceptación) en un grafo (el autómata de Büchi). Es más, el camino desde el nodo inicial hasta el nodo objetivo representa una ejecución del sistema concurrente en el que la propiedad de seguridad dada se viola: una *traza de error*. El problema que resolvemos en esta tesis es, precisamente, la búsqueda de este camino. Desde un punto de vista práctico, es preferible descubrir trazas de error cortas. La razón es que un programador humano que la analice puede comprender una traza corta con menos esfuerzo que una larga. Sin embargo, en esta tesis estamos interesados únicamente en encontrar una traza de error, no pretendemos optimizar su longitud.

3.3.4. *Model checking* heurístico

La simplificación en la exploración del grafo cuando se trabaja con propiedades de seguridad permite aplicar algoritmos clásicos de exploración de grafos como búsqueda primero en profundidad (DFS) y *búsqueda primero en anchura* (*Breadth First Search*, BFS) para buscar violaciones de propiedades de seguridad. También permite aplicar algoritmos de búsqueda heurística como A*, *Weighted* A* (WA*), *Iterative Deeping* A* (IDA*), y *búsqueda primero del mejor* (*Best-First search*, BF). Para realizar la búsqueda heurística se asocia a cada estado un valor heurístico que depende de la propiedad a verificar y que indica la preferencia por explorar ese estado.

El uso de heurísticas para guiar la búsqueda de errores en *model checking* se conoce como *model checking heurístico* o *guiado*. Las funciones heurísticas se diseñan para dirigir la exploración primero a la región del espacio de estados en el que es probable encontrar un estado de aceptación. De este modo, el tiempo y la memoria requeridos para encontrar un error en sistemas concurrentes con errores se reduce en término medio. No obstante, el uso de las heurísticas no supone ninguna ventaja cuando el objetivo es verificar que un programa dado cumple una determinada propiedad. En este caso, todo el espacio de estados se debe explorar exhaustivamente. Los algoritmos exhaustivos como A* o BF que usan heurísticas pueden encontrar errores en programas con menos recursos que los enfoques tradicionales como DFS. Además, pueden asegurar que un programa es correcto si no encuentran error (ya que son algoritmos exhaustivos).

Las funciones heurísticas suelen ser una estimación optimista de la longitud de una traza de error a partir del estado en que se evalúa. El algoritmo debe dar preferencia a la exploración de estados con valores heurísticos menores. Existen diferentes tipos de heurísticas. En [117] se introducen heurísticas estructurales que tratan de explorar la estructura del programa de forma que conduzca a encontrar errores. Ejemplos de este tipo de funciones heurísticas son las medidas de cobertura de código (instrucciones, ramas, predicados atómicos, etc.), medidas muy bien conocidas en el dominio de la generación automática de casos de prueba (véase la Sección 3.2.1). Otro ejemplo de este tipo de heurísticas es el entrelazado de hebras, en el que se premia a los estados que dan lugar a una planificación con un alto número de cambios de contexto.

A diferencia de las heurísticas estructurales, las heurísticas para propiedades específicas [117] se basan en características de la propiedad particular a comprobar. Las *heurísticas basadas en fórmulas*, por ejemplo, se basan en la fórmula LTL que define la propiedad [93]. Usando la expresión lógica que debe ser

falsa en el nodo objetivo, estas heurísticas estiman el número de transiciones requeridas para alcanzar el estado de aceptación desde el estado actual. Dada una fórmula φ (sin operadores temporales), la función heurística para la fórmula H_φ se define usando sus subfórmulas. La Tabla 3.1 reproduce la definición recursiva de una heurística basada en fórmula definida por Edelkamp *et al.* [93].

Tabla 3.1: Heurística basada en fórmula.

φ	$H_\varphi(s)$	$\overline{H}_\varphi(s)$
cierto	0	∞
falso	∞	0
p	**si** p **entonces** 0 **sino** 1	**si** p **entonces** 1 **sino** 0
$a \otimes b$	**si** $a \otimes b$ **entonces** 0 **sino** 1	**si** $a \otimes b$ **entonces** 1 **sino** 0
$\neg\psi$	$\overline{H}_\psi(s)$	$H_\psi(s)$
$\psi \vee \xi$	$\min\{H_\psi(s), H_\xi(s)\}$	$\overline{H}_\psi(s) + \overline{H}_\xi(s)$
$\psi \wedge \xi$	$H_\psi(s) + H_\xi(s)$	$\min\{\overline{H}_\psi(s), \overline{H}_\xi(s)\}$
llena(q)	$cap(q) - long(q)$	**si** *llena*(q) **entonces** 1 **sino** 0
vacía(q)	$long(q)$	**si** *vacía*(q) **entonces** 1 **sino** 0
$q?[t]$	prefijo mínimo de q sin t	**si** *cabeza*(q) $\neq t$ **entonces** 0 **sino** prefijo máximo de t's
$i@s$	$D_i(pc_i, s)$	**si** $pc_i = s$ **entonces** 1 **sino** 0

ψ, ξ: fórmulas sin operadores temporales
p: proposición lógica
a, b: variables o constantes
$\otimes$: operador relacional $(=, \neq, <, \leq, \geq, >)$
q: cola
$cap(q)$: capacidad de la cola q
$long(q)$: longitud de la cola q
cabeza(q): mensaje de la cabeza de la cola q
t: etiqueta de un mensaje
i: proceso
s: estado del autómata local de un proceso
pc_i: estado actual del proceso i en su autómata local
$D_i(u, v)$: número mínimo de transiciones para llegar de u a v en el autómata local del proceso i
$H_\varphi(s)$: Función heurística para la fórmula φ
$\overline{H}_\varphi(s)$: Función heurística para la fórmula $\neg\varphi$

Para buscar interbloqueos (propiedad de seguridad) se pueden usar varias funciones heurísticas. Por un lado, se puede usar el número de procesos activos como valor heurístico de un estado. Denotamos esta heurística con H_{ap}. Por otro lado, se puede usar también el número de transiciones ejecutables en un estado, denotado con H_{ex}. Otra opción consiste en aproximar la situación de interbloqueo con un predicado lógico [93] y derivar la función heurística usando las reglas de la Tabla 3.1.

Hay otro grupo de funciones heurísticas que están *basadas en estados* y que se pueden usar cuando se conoce el estado objetivo. De este grupo podemos destacar la distancia de Hamming y la distancia de máquinas de estados finitas. En el primer caso, el valor heurístico se calcula como la distancia de Hamming entre las representaciones binarias del estado actual y el objetivo. En el segundo, el valor heurístico es el mínimo número de transiciones requeridas para alcanzar el estado objetivo desde el estado actual en el correspondiente autómata de cada proceso. Las heurísticas basadas en estados se pueden usar junto con las anteriores para conseguir trazas de error más cortas del siguiente modo: una vez que se encuentra un estado de aceptación, una segunda búsqueda intenta encontrar el camino más corto al estado previamente encontrado. En esta segunda búsqueda, el estado de aceptación se conoce y se pueden usar heurísticas basadas en estados.

3.3.5. Reducción de orden parcial

La *reducción de orden parcial* (*Partial Order Reduction*, POR) es un método que explota la conmutatividad de los sistemas asíncronos para reducir el tamaño del espacio de estados del sistema concurrente. El modelo de intercalación de instrucciones para sistemas concurrentes impone una ordenación arbitraria entre eventos concurrentes. Cuando se calcula el autómata de Büchi del sistema, los eventos se intercalan de todas las formas posibles. Para la mayoría de las propiedades que se desean comprobar, la ordenación entre instrucciones independientes carece de sentido. Por esto, podemos considerar sólo una de las ordenaciones para comprobar si se cumple una determinada propiedad, ya que las otras ordenaciones son equivalentes. Este hecho se puede usar para construir un autómata de Büchi reducido y, probablemente, más fácil de explorar comparado con el autómata de Büchi original. Esta reducción permite verificar modelos de mayor tamaño que los que se pueden verificar cuando no se usa. El *model checker* SPIN incorpora esta técnica [145].

Dentro de las propiedades que pueden especificarse mediante lógica LTL, existe un subconjunto en el que la ordenación de los eventos concurrentes no tiene importancia. Este subconjunto está formado por todas aquellas propiedades que pueden especificarse mediante el sublenguaje LTL_{-X}, que corresponde a todas las fórmulas LTL que no usan el operador *next*.

La base de la reducción de orden parcial se encuentra en los conceptos de *activación*, *conmutatividad* e *invisibilidad* de instrucciones[8]. Formalmente, una instrucción $\alpha \in \mathit{Ins}$ es una relación binaria definida en el conjunto de estados del autómata de Büchi del sistema concurrente, es decir, $\alpha \subseteq Q \times Q$. Se dice que una instrucción $\alpha \in \mathit{Ins}$ está *activa* en un estado $s \in Q$ si existe un estado s' para el que se cumple $\alpha(s, s')$. El conjunto de las instrucciones activas en un estado s se denota con $\mathit{enabled}(s)$. En lo que sigue consideraremos tan sólo instrucciones deterministas, que son aquellas instrucciones α en las que para cada estado s existe a lo sumo un estado s' que satisfaga $\alpha(s, s')$. En lugar de una relación binaria general, las instrucciones deterministas definen una función parcial que transforma estados en estados. Así pues, usaremos $s' = \alpha(s)$ para indicar que la instrucción α transforma el estado s en s'.

Una vez introducida la notación, podemos definir el concepto de independencia de instrucciones. Dos instrucciones $\alpha, \beta \in \mathit{Ins}$ son *independientes* si para cada estado $s \in Q$ se cumplen las dos condiciones siguientes:

- si $\alpha, \beta \in \mathit{enabled}(s)$ entonces $\alpha \in \mathit{enabled}(\beta(s))$ y
- si $\alpha, \beta \in \mathit{enabled}(s)$ entonces $\alpha(\beta(s)) = \beta(\alpha(s))$.

De manera informal podemos decir que dos instrucciones α y β son independientes cuando ninguna de ellas desactiva a la otra y el estado al que se llega tras ejecutar ambas es el mismo independientemente del orden en que se ejecuten (Figura 3.8). En la Sección 3.3.1 se definió la relación de transición de estados como una relación ternaria en la que se asociaba a cada par de estados un subconjunto de AP. En dicha relación ternaria se cumple que todos los arcos que tienen como destino un mismo estado s tienen asociado el mismo subconjunto de AP, es decir,

$$\forall (r, \alpha, s), (r', \alpha', s') \in \Delta, s = s' \Rightarrow \alpha = \alpha' \ , \tag{3.27}$$

y por tanto, es posible asociar dicho subconjunto al estado s. De hecho, el valor de las proposiciones depende del estado del sistema concurrente, lo que significa que esta asociación de subconjuntos de AP

[8] En el contexto de los *sistemas de transición de estados* se suele usar el término *transición* para hacer referencia a lo que aquí denominamos instrucción.

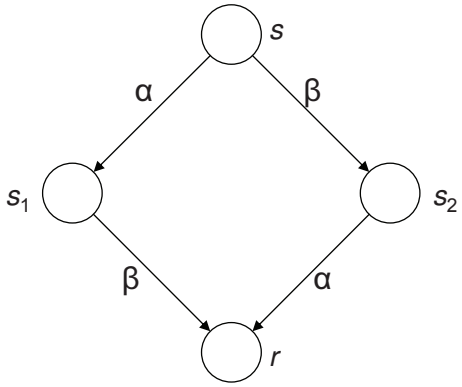


Figura 3.8: Ejecución de instrucciones independientes.

a estados aparece de forma natural. La asociación de conjuntos de proposiciones a arcos en el autómata de Büchi es un artificio necesario para representar el sistema concurrente mediante autómatas de Büchi. Así pues, llamaremos $L : Q \rightarrow 2^{AP}$ a la función que etiqueta estados del autómata con subconjuntos de AP de manera consistente con la relación de transición Δ del autómata. Es decir, se cumple:

$$\forall (s, \alpha, s') \in \Delta, \alpha = L(s') \ . \tag{3.28}$$

Se dice que una instrucción $\alpha \in Ins$ es invisible con respecto a un subconjunto de proposiciones atómicas $AP' \subseteq AP$ si para cada par de estados $s, s' \in Q$ tales que $s' = \alpha(s)$ se cumple que $L(s) \cap AP' = L(s') \cap AP'$. Informalmente esto quiere decir que la ejecución de la instrucción no modifica el valor de las proposiciones del conjunto AP'.

Una vez introducida la notación y los conceptos básicos, podemos detallar la técnica de reducción de orden parcial basada en *conjuntos amplios* (*ample sets*) [225]. La idea principal es explorar para cada estado s sólo un subconjunto $\mathit{ample}(s) \subseteq \mathit{enabled}(s)$ de las instrucciones activas en dicho estado, de forma que el autómata de Büchi reducido sea equivalente al autómata de Büchi original de acuerdo a la propiedad a verificar. Esta reducción puede realizarse conforme el autómata de Büchi se explora.

Para mantener la equivalencia entre el autómata de Büchi completo y el reducido, la función que asigna a cada estado el conjunto reducido de transiciones a explorar, *ample*, debe cumplir cuatro condiciones que pasamos a enumerar [58].

- **C0**: para todo estado s, $\mathit{ample}(s) = \emptyset$ si y sólo si $\mathit{enabled}(s) = \emptyset$.
- **C1**: para todo estado s y todo camino que parta de s en el autómata original, una instrucción α que dependa de otra instrucción $\beta \in \mathit{ample}(s)$ no se puede ejecutar sin que se haya ejecutado una instrucción de $\mathit{ample}(s)$ antes.
- **C2**: para todo estado s, si $\mathit{enabled}(s) \neq \mathit{ample}(s)$ entonces toda instrucción $\alpha \in \mathit{ample}(s)$ es invisible con respecto a las proposiciones que aparecen en la fórmula a verificar.
- **C3**: no existe ningún ciclo en el autómata reducido que contenga un estado en el que una instrucción α está activa pero no se incluye en el conjunto $\mathit{ample}(s)$ para ningún estado s del ciclo.

Las tres primeras condiciones no dependen del algoritmo de búsqueda particular utilizado para verificar la propiedad. Sin embargo, la forma de asegurar que se cumple **C3** depende del algoritmo de búsqueda. En [181] se proponen tres alternativas. De ellas, la única que se puede aplicar a todos los algoritmos de exploración es la denominada $\mathbf{C3}_{static}$ y ésa es la que utilizamos en los experimentos correspondientes.

3.3.6. Trabajos relacionados

La búsqueda de violaciones de propiedades de seguridad usando algoritmos clásicos de exploración de grafos, como DFS y BFS, ha sido estudiada por Edelkamp, Lluch-Lafuente y Leue [92, 93, 94]. No obstante, sus principales avances se concentran en el uso de búsqueda heurística para esta tarea. Para ello implementaron un *model checker* llamado HSF-SPIN que es una combinación de SPIN y una biblioteca para búsqueda heurística, HSF. Estudian la aplicación de algoritmos como A*, WA*, IDA* y BF. Los resultados muestran que, usando búsqueda heurística, la longitud de los contraejemplos se puede acortar y la cantidad de memoria requerida para obtener una traza de error se reduce, permitiendo la exploración de modelos mayores. Además, muestran que el uso de búsqueda heurística se puede combinar con reducción de orden parcial [181] y reducción por simetría [167]. También extienden el uso de información heurística para guiar la búsqueda de violaciones de propiedades de viveza [92, 94].

Cuando la búsqueda de errores con una cantidad baja de recursos computacionales (memoria y tiempo) es una prioridad (por ejemplo, en las primeras etapas de la implementación de un programa), se pueden usar algoritmos no exhaustivos que usan información heurística. Un ejemplo de esta clase de algoritmos es *Beam-search*, incluido en el *model checker* Java PathFinder [117, 118]. Los algoritmos no exhaustivos pueden encontrar errores en programas usando menos recursos computacionales que los exhaustivos (como veremos más adelante en el Capítulo 8), pero no pueden usarse para verificar una propiedad: cuando no se encuentra ninguna traza de error usando un algoritmo no exhaustivo no podemos asegurar que no existan dichas trazas. Debido a esto podemos establecer algunas similitudes entre el *model checking* heurístico usando algoritmos no exhaustivos y la generación de casos de prueba [206]. En ambos casos se explora una gran región del espacio de estados de un programa para descubrir errores, pero no encontrar errores no implica la corrección del programa. Esta relación entre ambos dominios se ha usado en el pasado para generar casos de prueba usando *model checkers* [23].

La búsqueda de estados de aceptación en el autómata de Büchi se puede transformar en un problema de optimización, y por tanto, se pueden aplicar algoritmos metaheurísticos. De hecho, se han aplicado en el pasado algoritmos genéticos a la búsqueda de errores en sistemas concurrentes. En una primera propuesta, Alba y Troya [13] usaron algoritmos genéticos para detectar interbloqueos, estados inútiles y transiciones inútiles en protocolos de comunicación. En su trabajo, una trayectoria en el grafo de estados se representa mediante una secuencia finita de números. Para la evaluación de una solución, un simulador de protocolos sigue la trayectoria sugerida y cuenta el número de estados y transiciones no usadas en las máquinas de estados finitas durante la simulación. Hasta donde alcanza nuestro conocimiento, ésta es la primera vez que se aplicó una técnica metaheurística al problema de buscar errores en sistemas concurrentes usando un enfoque basado en *model checking*. Más tarde, Godefroid y Kurshid [110, 111], en un trabajo independiente, aplicaron algoritmos genéticos al mismo problema usando una codificación similar de las trayectorias en el cromosoma. Su algoritmo se integró dentro de VeriSoft [109], un *model checker* que puede verificar programas en C.

3.4. Conclusiones

Hemos detallado en este capítulo tres problemas de optimización de Ingeniería del Software que serán abordados con técnicas metaheurísticas en los siguientes capítulos. Estos tres problemas son la planificación de proyectos software, la generación de casos de prueba y la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. Como mencionamos en el Capítulo 2, estos tres problemas son una muestra representativa de los intereses de la comunidad software. En los tres casos hemos dado una definición formal y hemos presentado algunos resultados teóricos de gran interés para el planteamiento de los problemas. Una vez presentados los problemas en este capítulo, nos disponemos a presentar las técnicas en el siguiente.

Capítulo 4

Metaheurísticas

La optimización en el sentido de encontrar la mejor solución, o al menos una solución lo suficientemente buena, para un problema es un campo de vital importancia en la vida real y en ingeniería. Constantemente estamos resolviendo pequeños problemas de optimización, como el camino más corto para ir de un lugar a otro, la organización de una agenda, etc. En general, estos problemas son lo suficientemente pequeños y podemos resolverlos sin ayuda adicional. Pero conforme se hacen más grandes y complejos, el uso de los ordenadores para su resolución es inevitable.

Comenzaremos este capítulo dando una definición formal del concepto de optimización. Asumiendo el caso de la minimización, podemos definir un *problema de optimización* como sigue:

Definición 7 (Problema de optimización)**.** *Un problema de optimización se formaliza como un par* (S, f)*, donde* $S \neq \emptyset$ *representa el espacio de soluciones (o de búsqueda) del problema, mientras que* f *es una función denominada* función objetivo *o* función de fitness*, que se define como:*

$$f : S \rightarrow \mathbb{R} \ . \tag{4.1}$$

Así, resolver un problema de optimización consiste en encontrar una solución, $i^* \in S$*, que satisfaga la siguiente desigualdad:*

$$f(i^*) \leq f(i), \qquad \forall \ i \in S \ . \tag{4.2}$$

Asumir el caso de maximización o minimización no restringe la generalidad de los resultados, puesto que se puede establecer una igualdad entre tipos de problemas de maximización y minimización de la siguiente forma [27, 114]:

$$\text{máx}\{f(i) | i \in S\} \equiv \text{mín}\{-f(i) | i \in S\} \ . \tag{4.3}$$

En función del dominio al que pertenezca S, podemos definir problemas de *optimización binaria* ($S \subseteq \mathbb{B}^*$), *entera* ($S \subseteq \mathbb{N}^*$), *continua*($S \subseteq \mathbb{R}^*$), o *heterogénea* ($S \subseteq (\mathbb{B} \cup \mathbb{N} \cup \mathbb{R})^*$).

Debido a la gran importancia de los problemas de optimización, a lo largo de la historia de la Informática se han desarrollado múltiples métodos para tratar de resolverlos. Una clasificación muy simple de estos métodos se muestra en la Figura 4.1. Inicialmente, las técnicas las podemos clasificar en exactas (o enumerativas, exhaustivas, etc.) y aproximadas. Las técnicas exactas garantizan encontrar la solución

óptima para cualquier instancia de cualquier problema en un tiempo acotado. El inconveniente de estos métodos es que el tiempo necesario para llevarlos a cabo, aunque acotado, crece exponencialmente con el tamaño del problema, ya que la mayoría de éstos son NP-duros. Esto supone en muchos casos que el uso de estas técnicas sea inviable, ya que se requiere mucho tiempo para la resolución del problema (posiblemente miles de años). Por lo tanto, los algoritmos aproximados para resolver estos problemas están recibiendo una atención cada vez mayor por parte de la comunidad internacional desde hace unas décadas. Estos métodos sacrifican la garantía de encontrar el óptimo a cambio de encontrar una "buena" solución en un tiempo razonable.

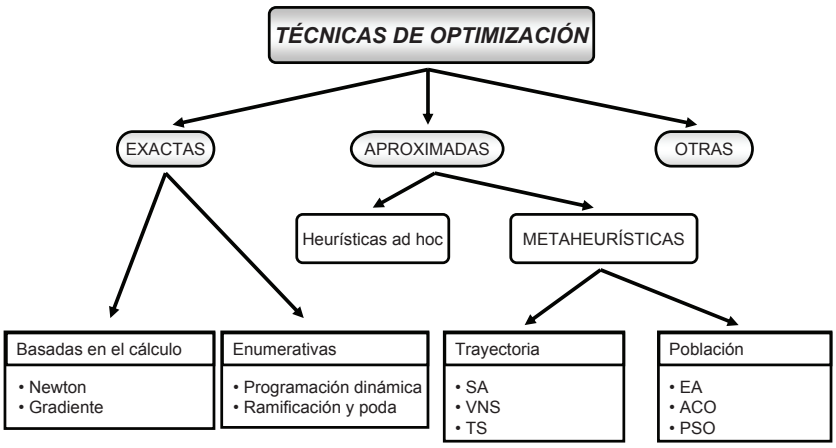


Figura 4.1: Clasificación de las técnicas de optimización.

Dentro de los algoritmos aproximados se pueden encontrar dos tipos: los heurísticos *ad hoc* y las metaheurísticas (en las que nos centramos en este capítulo). Los heurísticos *ad hoc*, a su vez, pueden dividirse en *heurísticos constructivos* y *métodos de búsqueda local*.

Los heurísticos constructivos suelen ser los métodos más rápidos. Construyen una solución mediante la incorporación de componentes hasta obtener una solución completa, que es el resultado del algoritmo. Aunque en muchos casos encontrar un heurístico constructivo es relativamente fácil, las soluciones ofrecidas suelen ser de muy baja calidad. Encontrar métodos de esta clase que produzca buenas soluciones es muy difícil, ya que dependen mucho del problema, y para su planteamiento se debe tener un conocimiento muy extenso del mismo. Por ejemplo, en problemas con muchas restricciones puede que la mayoría de las soluciones parciales sólo conduzcan a soluciones no factibles.

Los métodos de búsqueda local o seguimiento del gradiente parten de una solución ya completa y, usando el concepto de *vecindario*, recorren parte del espacio de búsqueda hasta encontrar un *óptimo local*. El vecindario de una solución s, que denotamos con $N(s)$, es el conjunto de soluciones que se pueden construir a partir de s aplicando un operador específico de modificación (generalmente denominado *movimiento*). Un óptimo local es una solución mejor o igual que cualquier otra solución de su vecindario. Estos métodos, partiendo de una solución inicial, examinan su vecindario y se quedan con el mejor vecino, continuando el proceso hasta que encuentran un óptimo local. En muchos casos, la exploración completa del vecindario es inabordable y se siguen diversas estrategias, dando lugar a diferentes variaciones del esquema genérico. Según el operador de movimiento elegido, el vecindario cambia y el modo de explorar el espacio de búsqueda también, pudiendo simplificarse o complicarse el proceso de búsqueda.

Finalmente, en los años setenta surgió una nueva clase de algoritmos aproximados, cuya idea básica era combinar diferentes métodos heurísticos a un nivel más alto para conseguir una exploración del espacio de búsqueda de forma eficiente y efectiva. Estas técnicas se han denominado *metaheurísticas*. Este término fue introducido por primera vez por Glover [105]. Antes de que el término fuese aceptado completamente por la comunidad científica, estas técnicas eran denominadas *heurísticas modernas* [234]. Esta clase de algoritmos incluye colonias de hormigas, algoritmos evolutivos, búsqueda local iterada, enfriamiento simulado, y búsqueda tabú. Se pueden encontrar revisiones de metaheurísticas en [4, 37, 107]. De las diferentes descripciones de metaheurísticas que se encuentran en la literatura se pueden destacar ciertas propiedades fundamentales que caracterizan a este tipo de métodos:

- Las metaheurísticas son estrategias o plantillas generales que "guían" el proceso de búsqueda.
- El objetivo es una exploración eficiente del espacio de búsqueda para encontrar soluciones (casi) óptimas.
- Las metaheurísticas son algoritmos no exactos y generalmente son no deterministas.
- Pueden incorporar mecanismos para evitar regiones no prometedoras del espacio de búsqueda.
- El esquema básico de cualquier metaheurística tiene una estructura predefinida.
- Las metaheurísticas hacen uso de conocimiento del problema que se trata de resolver en forma de heurísticos específicos que son controlados por una estrategia de más alto nivel.

Resumiendo estos puntos, se puede acordar que una metaheurística es una estrategia de alto nivel que usa diferentes métodos para explorar el espacio de búsqueda. En otras palabras, una metaheurística es una plantilla general no determinista que debe ser rellenada con datos específicos del problema (representación de las soluciones, operadores para manipularlas, etc.) y que permiten abordar problemas con espacios de búsqueda de gran tamaño. En este tipo de técnicas es especialmente importante el correcto equilibrio (generalmente dinámico) que haya entre *diversificación* e *intensificación*. El término diversificación se refiere a la evaluación de soluciones en regiones distantes del espacio de búsqueda (de acuerdo a una distancia previamente definida entre soluciones). También se conoce como *exploración* del espacio de búsqueda. El término intensificación, por otro lado, se refiere a la evaluación de soluciones en regiones acotadas y pequeñas con respecto al espacio de búsqueda centradas en el vecindario de soluciones concretas (*explotación* del espacio de búsqueda). El equilibrio entre estos dos aspectos contrapuestos es de gran importancia, ya que por un lado deben identificarse rápidamente las regiones prometedoras del espacio de búsqueda global y por otro lado no se debe malgastar tiempo en las regiones que ya han sido exploradas o que no contienen soluciones de alta calidad.

Dentro de las metaheurísticas podemos distinguir dos tipos de estrategias de búsqueda. Por un lado, tenemos las extensiones "inteligentes" de los métodos de búsqueda local. La meta de estas estrategias es evitar de alguna forma los mínimos locales y moverse a otras regiones prometedoras del espacio de búsqueda. Este tipo de estrategia es el seguido por la búsqueda tabú, la búsqueda local iterada, la búsqueda con vecindario variable y el enfriamiento simulado. Estas metaheurísticas trabajan sobre una o varias estructuras de vecindario impuestas por el espacio de búsqueda. Otro tipo de estrategia es el seguido por las colonias de hormigas o los algoritmos evolutivos. Éstos incorporan un componente de aprendizaje en el sentido de que, de forma implícita o explícita, intentan aprender la correlación entre las variables del problema para identificar las regiones del espacio de búsqueda con soluciones de alta calidad. Estos métodos realizan, en este sentido, un muestreo sesgado del espacio de búsqueda.

4.1. Definición formal

A continuación ofrecemos una definición formal de metaheurística basada en la desarrollada por Gabriel Luque en su tesis doctoral [183]. Una metaheurística se define como una tupla de elementos, los cuales, dependiendo de cómo se definan, darán lugar a una técnica concreta u otra.

Definición 8 (Metaheurística)**.** *Una metaheurística $\mathcal{M}$ es una tupla formada por los siguientes ocho componentes:*

$$\mathcal{M} = \langle \mathcal{T}, \Xi, \mu, \lambda, \Phi, \sigma, \mathcal{U}, \tau \rangle \ , \tag{4.4}$$

donde:

- *$\mathcal{T}$ es el conjunto de elementos que manipula la metaheurística. Este conjunto contiene al espacio de búsqueda y en la mayoría de los casos coincide con él.*
- *$\Xi = \{(\xi_1, D_1), (\xi_2, D_2), \ldots, (\xi_v, D_v)\}$ es un conjunto de v pares. Cada par está formado por una variable de estado de la metaheurística y el dominio de dicha variable.*
- *μ es el número de soluciones con las que trabaja $\mathcal{M}$ en un paso.*
- *λ es el número de nuevas soluciones generadas en cada iteración de $\mathcal{M}$.*
- *$\Phi : \mathcal{T}^\mu \times \prod_{i=1}^{v} D_i \times \mathcal{T}^\lambda \rightarrow [0,1]$ representa el operador que genera nuevas soluciones a partir de las existentes. Esta función debe cumplir para todo $x \in \mathcal{T}^\mu$ y para todo $t \in \prod_{i=1}^{v} D_i$,*

$$\sum_{y \in \mathcal{T}^\lambda} \Phi(x, t, y) = 1 \ . \tag{4.5}$$

- *$\sigma : \mathcal{T}^\mu \times \mathcal{T}^\lambda \times \prod_{i=1}^{v} D_i \times \mathcal{T}^\mu \rightarrow [0,1]$ es una función que permite seleccionar las soluciones que serán manipuladas en la siguiente iteración de $\mathcal{M}$. Esta función debe cumplir para todo $x \in \mathcal{T}^\mu$, $z \in \mathcal{T}^\lambda$ y $t \in \prod_{i=1}^{v} D_i$,*

$$\sum_{y \in \mathcal{T}^\mu} \sigma(x, z, t, y) = 1 \ , \tag{4.6}$$

$$\begin{aligned} &\forall y \in \mathcal{T}^\mu, \sigma(x, z, t, y) = 0 \vee \\ &\quad \vee \sigma(x, z, t, y) > 0 \wedge (\forall i \in \{1, \ldots, \mu\} \bullet (\exists j \in \{1, \ldots, \mu\}, y_i = x_j) \vee (\exists j \in \{1, \ldots, \lambda\}, y_i = z_j)) \ . \end{aligned} \tag{4.7}$$

- *$\mathcal{U} : \mathcal{T}^\mu \times \mathcal{T}^\lambda \times \prod_{i=1}^{v} D_i \times \prod_{i=1}^{v} D_i \rightarrow [0,1]$ representa el procedimiento de actualización de las variables de estado de la metaheurística. Esta función debe cumplir para todo $x \in \mathcal{T}^\mu$, $z \in \mathcal{T}^\lambda$ y $t \in \prod_{i=1}^{v} D_i$,*

$$\sum_{u \in \prod_{i=1}^{v} D_i} \mathcal{U}(x, z, t, u) = 1 \ . \tag{4.8}$$

- *$\tau : \mathcal{T}^\mu \times \prod_{i=1}^{v} D_i \rightarrow \{falso, cierto\}$ es una función que decide la terminación del algoritmo.*

La definición anterior recoge el comportamiento estocástico típico de las técnicas metaheurísticas. En concreto, las funciones Φ, σ y $\mathcal{U}$ deben interpretarse como probabilidades condicionadas. Por ejemplo, el valor de $\Phi(x,t,y)$ se interpreta como la probabilidad de que se genere el vector de hijos $y \in \mathcal{T}^\lambda$ dado que actualmente el conjunto de individuos con el que la metaheurística trabaja es $x \in \mathcal{T}^\mu$ y su estado interno viene definido por las variables de estado $t \in \prod_{i=1}^{v} D_i$. Puede observarse que las restricciones que se le imponen a las funciones Φ, σ y $\mathcal{U}$ permite considerarlas como funciones que devuelven estas probabilidades condicionadas.

Definición 9 (Estado de una metaheurística)**.** *Sea $\mathcal{M} = \langle \mathcal{T}, \Xi, \mu, \lambda, \Phi, \sigma, \mathcal{U}, \tau \rangle$ una metaheurística y $\Theta = \{\theta_1, \theta_2, \ldots, \theta_\mu\}$ el conjunto de variables que almacenarán las soluciones con las que trabaja la metaheurística. Utilizaremos la notación $first(\Xi)$ para referirnos al conjunto de las variables de estado de la metaheurística, $\{\xi_1, \xi_2, \ldots, \xi_v\}$. Un* estado *$s$ de la metaheurística es un par de funciones $s = (s_1, s_2)$ con*

$$s_1 : \Theta \to \mathcal{T}, \tag{4.9}$$

$$s_2 : first(\Xi) \to \bigcup_{i=1}^{v} D_i \ , \tag{4.10}$$

donde s_2 cumple

$$s_2(\xi_i) \in D_i \ \ \forall \xi_i \in first(\Xi) \ . \tag{4.11}$$

Denotaremos con $\mathcal{S}_\mathcal{M}$ el conjunto de todos los estados de una metaheurística $\mathcal{M}$.

Por último, una vez definido el estado de la metaheurística, podemos definir su dinámica.

Definición 10 (Dinámica de una metaheurística)**.** *Sea $\mathcal{M} = \langle \mathcal{T}, \Xi, \mu, \lambda, \Phi, \sigma, \mathcal{U}, \tau \rangle$ una metaheurística y $\Theta = \{\theta_1, \theta_2, \ldots, \theta_\mu\}$ el conjunto de variables que almacenarán las soluciones con las que trabaja la metaheurística. Denotaremos con $\overline{\Theta}$ a la tupla $(\theta_1, \theta_2, \ldots, \theta_\mu)$ y con $\overline{\Xi}$ a la tupla $(\xi_1, \xi_2, \ldots, \xi_v)$. Extenderemos la definición de estado para que pueda aplicarse a tuplas de elementos, esto es, definimos $\bar{s} = (\bar{s}_1, \bar{s}_2)$ donde*

$$\bar{s}_1 : \Theta^n \to \mathcal{T}^n \ , \tag{4.12}$$

$$\bar{s}_2 : first(\Xi)^n \to \left(\bigcup_{i=1}^{v} D_i \right)^n \ , \tag{4.13}$$

y además

$$\bar{s}_1(\theta_{i_1}, \theta_{i_2}, \ldots, \theta_{i_n}) = (s_1(\theta_{i_1}), s_1(\theta_{i_2}), \ldots, s_1(\theta_{i_n})) \ , \tag{4.14}$$

$$\bar{s}_2(\xi_{j_1}, \xi_{j_2}, \ldots, \xi_{j_n}) = (s_2(\xi_{j_1}), s_2(\xi_{j_2}), \ldots, s_2(\xi_{j_n})) \ , \tag{4.15}$$

para $n \geq 2$. Diremos que r es un estado sucesor de s si existe $t \in \mathcal{T}^\lambda$ tal que $\Phi(\bar{s}_1(\overline{\Theta}), \bar{s}_2(\overline{\Xi}), t) > 0$ y además

$$\sigma(\bar{s}_1(\overline{\Theta}), t, \bar{s}_2(\overline{\Xi}), \bar{r}_1(\overline{\Theta})) > 0 \ \ y \tag{4.16}$$

$$\mathcal{U}(\bar{s}_1(\overline{\Theta}), t, \bar{s}_2(\overline{\Xi}), \bar{r}_2(\overline{\Xi})) > 0 \ . \tag{4.17}$$

Denotaremos con $\mathcal{F}_\mathcal{M}$ la relación binaria "ser sucesor de" definida en el conjunto de estados de una metaheurística $\mathcal{M}$. Es decir, $\mathcal{F}_\mathcal{M} \subseteq \mathcal{S}_\mathcal{M} \times \mathcal{S}_\mathcal{M}$, y $\mathcal{F}_\mathcal{M}(s,r)$ si r es un estado sucesor de s.

Definición 11 (Ejecución de una metaheurística)**.** *Una ejecución de una metaheurística* $\mathcal{M}$ *es una secuencia finita o infinita de estados,* $s_0, s_1, \ldots$ *en la que* $\mathcal{F}_{\mathcal{M}}(s_i, s_{i+1})$ *para todo* $i \geq 0$ *y además:*

- *si la secuencia es infinita se cumple* $\tau(s_i(\overline{\Theta}), s_i(\overline{\Xi})) = falso$ *para todo* $i \geq 0$ *y*
- *si la secuencia es finita se cumple* $\tau(s_k(\overline{\Theta}), s_k(\overline{\Xi})) = cierto$ *para el último estado* s_k *y, además,* $\tau(s_i(\overline{\Theta}), s_i(\overline{\Xi})) = falso$ *para todo* $i \geq 0$ *tal que* $i < k$*.*

En las próximas secciones tendremos la oportunidad de comprobar cómo esta formulación general se puede adaptar a las técnicas concretas (obviando aquellos parámetros no fijados por la metaheurística o que dependen de otros aspectos como el problema o la implementación concreta).

4.2. Clasificación de las metaheurísticas

Hay diferentes formas de clasificar y describir las técnicas metaheurísticas [53, 65]. Dependiendo de las características que se seleccionen se pueden obtener diferentes taxonomías: basadas en la naturaleza y no basadas en la naturaleza, con memoria o sin ella, con una o varias estructuras de vecindario, etc. Una de las clasificaciones más populares las divide en metaheurísticas *basadas en trayectoria* y *basadas en población*. Las primeras manipulan en cada paso un único elemento del espacio de búsqueda, mientras que las segundas trabajan sobre un conjunto de ellos (población). Esta taxonomía se muestra de forma gráfica en la Figura 4.2, que además incluye las principales metaheurísticas.

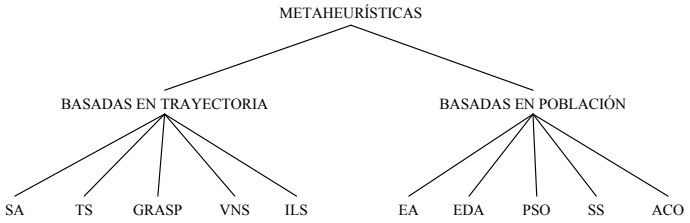


Figura 4.2: Clasificación de las metaheurísticas.

4.2.1. Metaheurísticas basadas en trayectoria

En esta sección repasaremos brevemente algunas metaheurísticas basadas en trayectoria. La principal característica de estos métodos es que parten de una solución y, mediante la exploración del vecindario, van actualizando la solución actual, formando una trayectoria. Según la notación de la Definición 8, esto se formaliza con $\mu = 1$. La mayoría de estos algoritmos surgen como extensiones de los métodos de búsqueda local simples a los que se les añade algún mecanismo para escapar de los mínimos locales. Esto implica la necesidad de una condición de parada más elaborada que la de encontrar un mínimo local. Normalmente se termina la búsqueda cuando se alcanza un número máximo predefinido de iteraciones, se encuentra una solución con una calidad aceptable, o se detecta un estancamiento del proceso.

Enfriamiento simulado (SA)

El *enfriamiento simulado* o *Simulated Annealing* (SA) es una de las técnicas más antiguas entre las metaheurísticas y posiblemente es el primer algoritmo con una estrategia explícita para escapar de los mínimos locales. Los orígenes del algoritmo se encuentran en un mecanismo estadístico, denominado *metropolis* [205]. La idea del SA es simular el proceso de enfriamiento del metal y del cristal. El SA fue inicialmente presentado en [161]. Para evitar quedar atrapado en un mínimo local, el algoritmo permite elegir una solución cuyo valor de *fitness* sea peor que el de la solución actual. En cada iteración se elige, a partir de la solución actual s, una solución s' del vecindario $N(s)$. Si s' es mejor que s (es decir, tiene un mejor valor en la función de *fitness*), se sustituye s por s' como solución actual. Si la solución s' es peor, entonces es aceptada con una determinada probabilidad que depende de la temperatura actual T y de la diferencia de *fitness* entre ambas soluciones, $f(s') - f(s)$ (caso de minimización).

Búsqueda tabú (TS)

La *búsqueda tabú* o *Tabu Search* (TS) es una de las metaheurísticas que se han aplicado con más éxito a la hora de resolver problemas de optimización combinatoria. Los fundamentos de este método fueron introducidos en [105], y están basados en las ideas formuladas en [104]. Un buen resumen de esta técnica y sus componentes se puede encontrar en [108]. La idea básica de la búsqueda tabú es el uso explícito de un historial de la búsqueda (una memoria a corto plazo), tanto para escapar de los mínimos locales como para implementar su estrategia de exploración y evitar buscar varias veces en la misma región. Esta memoria a corto plazo se implementa con una lista tabú, donde se mantienen las soluciones visitadas más recientemente para excluirlas de los próximos movimientos. En cada iteración se elige la mejor solución entre las permitidas y la solución es añadida a la lista tabú. Desde el punto de vista de la implementación, mantener una lista de soluciones completas no suele ser práctico debido a su ineficiencia. Por lo tanto, en general, se suelen almacenar los movimientos que ha llevado al algoritmo a generar esa solución o los componentes principales que definen la solución. En cualquier caso, los elementos de esta lista permite filtrar el vecindario, generando un conjunto reducido de soluciones elegibles denominado $N_a(s)$. El almacenamiento de los movimientos en vez de las soluciones completas es bastante más eficiente, pero introduce una perdida de información. Para evitar este problema, se define un criterio de aspiración que permite incluir una solución en $N_a(s)$ incluso si está prohibida debido a la lista tabú. El criterio de aspiración más ampliamente usado es permitir soluciones cuyo *fitness* sea mejor que el de la mejor solución encontrada hasta el momento.

GRASP

El *procedimiento de búsqueda miope aleatorizado y adaptativo* o *Greedy Randomized Adaptive Search Procedure* (GRASP) [99] es una metaheurística simple que combina heurísticos constructivos con búsqueda local. GRASP es un procedimiento iterativo, compuesto de dos fases: primero la construcción de una solución y después un proceso de mejora. La solución mejorada es el resultado del proceso de búsqueda. El mecanismo de construcción de soluciones es un heurístico constructivo aleatorio. Va añadiendo paso a paso diferentes componentes c a la solución parcial s^p, que inicialmente está vacía. Los componentes que se añaden en cada paso son elegidos aleatoriamente de una lista restringida de candidatos (RCL). Esta lista es un subconjunto de $N(s^p)$, el conjunto de componentes permitidos para la solución parcial s^p. Para generar esta lista, los componentes de la solución en $N(s^p)$ se ordenan de acuerdo a alguna función dependiente del problema (η). La lista RCL está compuesta por los α mejores componentes de ese conjunto. En

el caso extremo de $\alpha = 1$, siempre se añade el mejor componente encontrado de manera determinista, con lo que el método de construcción es equivalente a un algoritmo voraz. En el otro extremo, con $\alpha = |N(s^p)|$ el componente a añadir se elige de forma totalmente aleatoria de entre todos los disponibles. Por lo tanto, α es un parámetro clave que influye en cómo se va a muestrear el espacio de búsqueda. La segunda fase del algoritmo consiste en aplicar un método de búsqueda local para mejorar la solución generada. Este mecanismo de mejora puede ser una técnica de mejora simple o algoritmos más complejos como SA o TS.

Búsqueda con vecindario variable (VNS)

La *búsqueda con vecindario variable* o *Variable Neighborhood Search* (VNS) es una metaheurística propuesta en [213] que aplica explícitamente una estrategia para cambiar entre diferentes vecindarios durante la búsqueda. Este algoritmo es muy general y con muchos grados de libertad a la hora de diseñar variaciones e instanciaciones particulares. El primer paso a realizar es definir un conjunto de vecindarios. Esta elección puede hacerse de muchas formas: desde ser elegidos aleatoriamente hasta utilizar complejas ecuaciones deducidas del problema. Cada iteración consiste en tres fases: la elección del candidato, una fase de mejora y, finalmente, el movimiento. En la primera fase, se elige aleatoriamente un vecino s' de s usando el k-ésimo vecindario. Esta solución s' es utilizada como punto de partida de la búsqueda local de la segunda fase. Cuando termina el proceso de mejora, se compara la nueva solución s'' con la original s. Si es mejor, s'' se convierte en la solución actual y se inicializa el contador de vecindarios ($k \leftarrow 1$); si no es mejor, se repite el proceso pero utilizando el siguiente vecindario ($k \leftarrow k+1$). La búsqueda local es el paso de intensificación del método y el cambio de vecindario puede considerarse como el paso de diversificación.

Búsqueda local iterada (ILS)

La *búsqueda local iterada* o *Iterated Local Search* (ILS) [127, 263] es una metaheurística basada en un concepto simple pero muy efectivo. En cada iteración, la solución actual es perturbada y a esta nueva solución se le aplica un método de búsqueda local para mejorarla. Este nuevo mínimo local obtenido por el método de mejora puede ser aceptado como nueva solución actual si pasa un test de aceptación. La importancia del proceso de perturbación es obvia: si es demasiado pequeña puede que el algoritmo no sea capaz de escapar del mínimo local; por otro lado, si es demasiado grande, la perturbación puede hacer que el algoritmo sea como un método de búsqueda local con un reinicio aleatorio. Por lo tanto, el método de perturbación debe generar una nueva solución que sirva como inicio a la búsqueda local, pero que no debe estar muy lejos del actual para que no sea una solución aleatoria. El criterio de aceptación actúa como contra-balance, ya que filtra la aceptación de nuevas soluciones dependiendo de la historia de búsqueda y de las características del nuevo mínimo local.

4.2.2. Metaheurísticas basadas en población

Los métodos basados en población se caracterizan por trabajar con un conjunto de soluciones (denominado población) en cada iteración (es decir, generalmente $\mu > 1$ y/o $\lambda > 1$), a diferencia de los métodos basados en trayectoria, que únicamente manipulan una solución del espacio de búsqueda por iteración.

Algoritmos evolutivos (EA)

Los *algoritmos evolutivos* o *Evolutionary Algorithms* (EA) están inspirados en la teoría de la evolución natural. Esta familia de técnicas sigue un proceso iterativo y estocástico que opera sobre una población

de soluciones, denominadas en este contexto *individuos*. Inicialmente, la población es generada aleatoriamente (quizás con ayuda de un heurístico de construcción). El esquema general de un algoritmo evolutivo comprende tres fases principales: selección, reproducción y reemplazo. El proceso completo es repetido hasta que se cumpla un cierto criterio de terminación (normalmente después de un número dado de iteraciones). En la fase de selección se seleccionan generalmente los individuos más aptos de la población actual para ser posteriormente recombinados en la fase de reproducción. Los individuos resultantes de la recombinación se alteran mediante un operador de mutación. Finalmente, a partir de la población actual y/o los mejores individuos generados (de acuerdo a su valor de *fitness*) se forma la nueva población, dando paso a la siguiente generación del algoritmo.

Algoritmos de estimación de la distribución (EDA)

Los *algoritmos de estimación de la distribución* o *Estimation of Distribution Algorithms* (EDA) [216] muestran un comportamiento similar a los algoritmos evolutivos presentados en la sección anterior y, de hecho, muchos autores consideran los EDA como otro tipo de EA. Los EDA operan sobre una población de soluciones tentativas como los algoritmos evolutivos pero, a diferencia de estos últimos, que utilizan operadores de recombinación y mutación para mejorar las soluciones, los EDA infieren la distribución de probabilidad del conjunto seleccionado y, a partir de ésta, generan nuevas soluciones que formarán parte de la población. Los modelos gráficos probabilísticos son herramientas comúnmente usadas en el contexto de los EDA para representar eficientemente la distribución de probabilidad. Algunos autores [173, 226, 258] han propuesto las redes bayesianas para representar la distribución de probabilidad en dominios discretos, mientras que las redes gaussianas se emplean usualmente en los dominios continuos [289].

Búsqueda dispersa (SS)

La *búsqueda dispersa* o *Scatter Search* (SS) [106] es una metaheurística cuyos principios fueron presentados en [104] y que actualmente está recibiendo una gran atención por parte de la comunidad científica [168]. El algoritmo mantiene un conjunto relativamente pequeño de soluciones tentativas (llamado conjunto de referencia o *RefSet*) que se caracteriza por contener soluciones de calidad y diversas (distantes en el espacio de búsqueda). Para la definición completa de SS hay que concretar cinco componentes: creación de la población inicial, generación del conjunto de referencia, generación de subconjuntos de soluciones, método de combinación de soluciones y método de mejora.

Optimización basada en colonias de hormigas (ACO)

Los *algoritmos de optimización basados en colonias de hormigas* o *Ant Colony Optimization* (ACO) [86, 87] están inspirados en el comportamiento de las hormigas reales cuando buscan comida. Este comportamiento es el siguiente: inicialmente, las hormigas exploran el área cercana a su nido de forma aleatoria. Tan pronto como una hormiga encuentra comida, la lleva al nido. Mientras que realiza este camino, la hormiga va depositando una sustancia química denominada feromona. Esta sustancia ayudará al resto de las hormigas a encontrar la comida. La comunicación indirecta entre las hormigas mediante el rastro de feromona las capacita para encontrar el camino más corto entre el nido y la comida. Este comportamiento es el que intenta simular este método para resolver problemas de optimización. La técnica se basa en dos pasos principales: construcción de una solución basada en el comportamiento de una hormiga y actualización de los rastros de feromona artificiales. El algoritmo no fija ninguna planificación o sincronización *a priori* entre las fases, pudiendo ser incluso realizadas simultáneamente.

Optimización basada en cúmulos de partículas (PSO)

Los *algoritmos de optimización basados en cúmulos de partículas* o *Particle Swarm Optimization* (PSO) [158] están inspirados en el comportamiento social del vuelo de las bandadas de aves o el movimiento de los bancos de peces. El algoritmo PSO mantiene un conjunto de soluciones, también llamadas *partículas*, que son inicializadas aleatoriamente en el espacio de búsqueda. Cada partícula posee una posición y velocidad que cambia conforme avanza la búsqueda. En el movimiento de una partícula influye su velocidad y las posiciones donde la propia partícula y las partículas de su vecindario encontraron buenas soluciones. En el contexto de PSO, el *vecindario de una partícula* se define como un conjunto de partículas del cúmulo. No debe confundirse con el concepto de vecindario de una solución utilizado previamente en este capítulo. El vecindario de una partícula puede ser *global*, en el cual todas las partículas del cúmulo se consideran vecinas, o *local*, en el que sólo las partículas más cercanas se consideran vecinas.

4.3. Metaheurísticas paralelas

Aunque el uso de metaheurísticas permite reducir significativamente la complejidad temporal del proceso de búsqueda, este tiempo sigue siendo muy elevado en algunos problemas de interés real. El paralelismo puede ayudar no sólo a reducir el tiempo de cómputo, sino a producir también una mejora en la calidad de las soluciones encontradas. Para cada una de las categorías mostradas en la sección anterior se han propuesto diferentes modelos paralelos acorde a sus características. A continuación presentaremos generalidades relacionadas con la paralelización de las metaheurísticas basadas en trayectoria y población.

4.3.1. Modelos paralelos para métodos basados en trayectoria

Los modelos paralelos de metaheurísticas basadas en trayectoria encontrados en la literatura se pueden clasificar, generalmente, dentro de tres posibles esquemas: ejecución en paralelo de varios métodos (*modelo de múltiples ejecuciones*), exploración en paralelo del vecindario (*modelo de movimientos paralelos*), y cálculo en paralelo de la función de *fitness* (*modelo de aceleración del movimiento*). A continuación detallamos cada uno de ellos.

- **Modelo de múltiples ejecuciones:** este modelo consiste en ejecutar en paralelo varios subalgoritmos ya sean homogéneos o heterogéneos [182]. En general, cada subalgoritmo comienza con una solución inicial diferente. Se pueden distinguir diferentes casos dependiendo de si los subalgoritmos colaboran entre sí o no. El caso en el que las ejecuciones son totalmente independientes se usa ampliamente porque es simple de utilizar y muy natural. En este caso, la semántica del modelo es la misma que la de la ejecución secuencial, ya que no existe cooperación. El único beneficio al utilizar este modelo respecto a realizar las ejecuciones en una única máquina, es la reducción del tiempo de ejecución total.

  Por otro lado, en el caso cooperativo (véase el ejemplo de la derecha de la Figura 4.3), los diferentes subalgoritmos intercambian información durante la ejecución. En este caso el comportamiento global del algoritmo paralelo es diferente al secuencial y su rendimiento se ve afectado por cómo esté configurado este intercambio. El usuario debe fijar ciertos parámetros para completar el modelo: qué información se intercambian, cada cuánto se pasan la información y cómo se realiza este intercambio. La información intercambiada suele ser la mejor solución encontrada, los movimientos realizados o algún tipo de información sobre la trayectoria realizada. En cualquier caso, esta

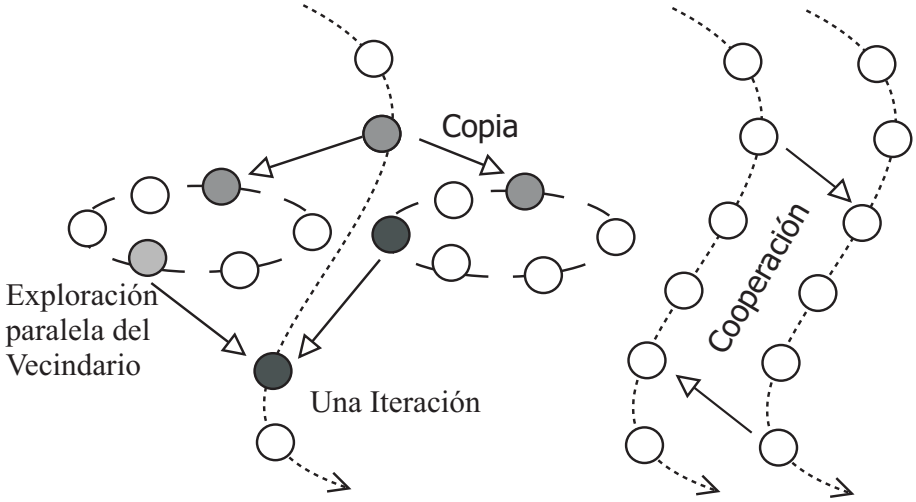


Figura 4.3: Modelos paralelos más usados en los métodos basados en trayectoria. A la izquierda se muestra el modelo de movimientos paralelos, donde se hace una exploración paralela del vecindario. A la derecha, se detalla el modelo de ejecuciones múltiples con cooperación, donde hay varios métodos ejecutándose en paralelo y cooperando entre ellos.

información no debe ser abundante para que el coste de la comunicación no sea excesivo e influya negativamente en la eficiencia. También se debe fijar cada cuántos pasos del algoritmo se intercambia la información. Para elegir este valor hay que tener en cuenta que el intercambio no sea muy frecuente, para que el coste de la comunicación no sea perjudicial; ni muy poco frecuente, para que el intercambio tenga algún efecto en el comportamiento global. Por último, se debe indicar si las comunicaciones se realizarán de forma asíncrona o síncrona. En el caso síncrono, los subalgoritmos, cuando llegan a la fase de comunicación, se detienen hasta que todos ellos llegan a este paso y, sólo entonces, se realiza la comunicación. En el caso asíncrono (el más usado), cada subalgoritmo realiza el intercambio sin esperar al resto cuando llega al paso de comunicación.

- **Modelo de movimientos paralelos:** los métodos basados en trayectoria en cada paso examinan parte de su vecindario y, de él, eligen la siguiente solución a considerar. Este paso suele ser computacionalmente costoso, ya que examinar el vecindario implica múltiples cálculos de la función de *fitness*. El modelo de movimientos paralelos tiene como objetivo acelerar dicho proceso mediante la exploración en paralelo del vecindario (véase el esquema de la izquierda de la Figura 4.3). Siguiendo un modelo maestro-esclavo, el maestro (el que ejecuta el algoritmo) pasa a cada esclavo la solución actual. Cada esclavo explora parte del vecindario de esta solución devolviendo la más prometedora. Entre todas estas soluciones devueltas el maestro elige una para continuar el proceso. Este modelo no cambia la semántica del algoritmo, sino que simplemente acelera su ejecución en caso de ser lanzado en una plataforma paralela. Este modelo es bastante popular debido a su simplicidad.

- **Modelo de aceleración del movimiento:** en muchos casos, el proceso más costoso del algoritmo es el cálculo de la función de *fitness*. Pero este cálculo, en muchos problemas, se puede descomponer en varios cómputos independientes más simples que, una vez llevados a cabo, se pueden combinar

para obtener el valor final de la función de *fitness*. En este modelo, cada uno de esos cálculos más simples se asignan a los diferentes procesadores y se realizan en paralelo, acelerando el cálculo total. Al igual que el anterior, este modelo tampoco modifica la semántica del algoritmo respecto a su ejecución secuencial.

4.3.2. Modelos paralelos para métodos basados en población

El paralelismo surge de manera natural cuando se trabaja con poblaciones, ya que cada individuo puede manejarse de forma independiente. Debido a esto, el rendimiento de los algoritmos basados en población suele mejorar bastante cuando se ejecutan en paralelo. A alto nivel podemos dividir las estrategias de paralelización de este tipo de métodos en dos categorías: (1) paralelización del cómputo, donde las operaciones que se llevan a cabo sobre los individuos son ejecutadas en paralelo; y (2) paralelización de la población, donde se procede a la estructuración de la población.

Uno de los modelos más utilizado que sigue la primera de las estrategias es el denominado *maestro-esclavo* (también conocido como *paralelización global*). En este esquema, un proceso central realiza las operaciones que afectan a toda la población (como, por ejemplo, la selección en los algoritmos evolutivos) mientras que los procesos esclavos se encargan de las operaciones que afectan a los individuos independientemente (como la evaluación de la función de *fitness*, la mutación e incluso, en algunos casos, la recombinación). Con este modelo, la semántica del algoritmo paralelo no cambia respecto al secuencial pero el tiempo global de cómputo es reducido. Este tipo de estrategias son muy utilizadas en las situaciones donde el cálculo de la función de *fitness* es un proceso muy costoso en tiempo. Otra estrategia muy popular es la de acelerar el cómputo mediante la realización de múltiples ejecuciones independientes (sin ninguna interacción entre ellas) usando múltiples máquinas para, finalmente, quedarse con la mejor solución encontrada entre todas las ejecuciones. Al igual que ocurría con el modelo de múltiples ejecuciones sin cooperación de las metaheurísticas paralelas basadas en trayectoria, este esquema no cambia el comportamiento del algoritmo, pero permite reducir de forma importante el tiempo total de cómputo.

Al margen del modelo maestro-esclavo, la mayoría de los algoritmos paralelos basados en población encontrados en la literatura utilizan alguna clase de estructuración de los individuos de la población. Este esquema es ampliamente utilizado especialmente en el campo de los algoritmos evolutivos, en el cual nos centraremos por ser los que usamos como propuesta paralela en uno de los problemas abordados. Entre los esquemas más populares para estructurar la población encontramos el modelo *distribuido* (o de grano grueso) y el modelo *celular* (o de grano fino) [10].

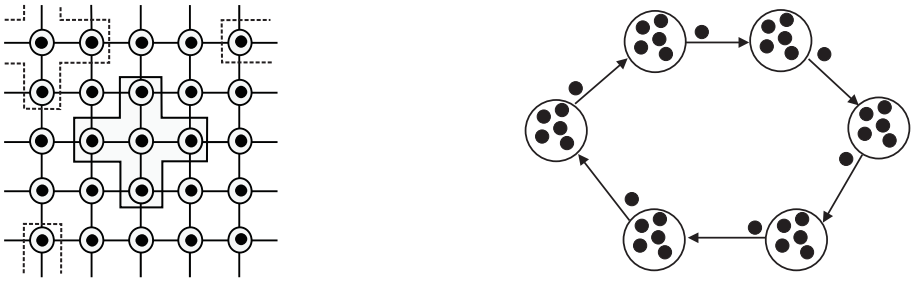


Figura 4.4: Los dos modelos más populares para estructurar la población: a la izquierda el modelo celular y a la derecha el modelo distribuido.

En el caso de los algoritmos distribuidos [14] (véase el esquema de la derecha en la Figura 4.4), la población se divide entre un conjunto de islas que ejecutan una metaheurística secuencial. Las islas cooperan entre sí mediante el intercambio de información (generalmente individuos, aunque nada impide intercambiar otro tipo de información). Esta cooperación permite introducir diversidad en las subpoblaciones, evitando caer así en los óptimos locales. Para terminar de definir este esquema el usuario debe dar una serie de parámetros como: topología, que indica a dónde se envían los individuos de cada isla y de dónde se pueden recibir; periodo de migración, que es el número de iteraciones entre dos intercambios de información; tasa de migración, que es el número de individuos emigrados; criterio de selección de los individuos a migrar y criterio de reemplazo, que indica si se reemplazan algunos individuos de la población actual para introducir a los inmigrantes y determina qué individuos se reemplazarán. Finalmente, se debe decidir si estos intercambios se realizan de forma síncrona o asíncrona.

Por otro lado, las metaheurísticas celulares [89] (véase el esquema de la izquierda en la Figura 4.4) se basan en el concepto de vecindario[1]. Cada individuo tiene a su alrededor un conjunto de individuos vecinos donde se lleva a cabo la explotación de las soluciones. La exploración y la difusión de las soluciones al resto de la población se produce debido a que los vecindarios están solapados, lo que produce que las buenas soluciones se extiendan lentamente por toda la población.

A parte de estos modelos básicos, en la literatura también se han propuesto modelos híbridos donde se implementan esquemas de dos niveles. Por ejemplo, una estrategia bastante común en la literatura es aquélla donde en el nivel más alto tenemos un esquema de grano grueso, mientras que cada subpoblación se organiza siguiendo un esquema celular.

4.4. Metaheurísticas usadas

Tras el rápido repaso por los algoritmos metaheurísticos, en esta sección presentaremos de forma detallada las familias de las que tomamos el esquema de búsqueda para luego aplicarlos a los problemas elegidos de Ingeniería del Software. Estas familias son: algoritmos evolutivos (algoritmos genéticos y estrategias evolutivas), optimización basada en cúmulos de partículas y optimización basada en colonias de hormigas. Dentro de la sección dedicada a los algoritmos evolutivos detallaremos especialmente las particularidades de los algoritmos genéticos y las estrategias evolutivas por ser dos de las técnicas usadas en esta tesis.

4.4.1. Algoritmos evolutivos

Alrededor de los años 60, algunos investigadores visionarios coincidieron (de forma independiente) en la idea de implementar algoritmos basados en el modelo de evolución orgánica como un intento de resolver tareas complejas de optimización en ordenadores. Hoy en día, debido a su robustez, a su amplia aplicabilidad, y también a la disponibilidad de una cada vez mayor potencia computacional, e incluso programas paralelos, el campo de investigación resultante, el de la computación evolutiva, recibe una atención creciente por parte de los investigadores de un gran número de disciplinas.

El marco de la computación evolutiva [28] establece una aproximación para resolver el problema de buscar valores óptimos mediante el uso de modelos computacionales basados en procesos evolutivos (algoritmos evolutivos). Los EA son técnicas de optimización que trabajan sobre poblaciones de soluciones

[1] De nuevo aquí, la palabra *vecindario* se usa en el sentido de definir un conjunto de individuos que serán vecinos de uno dado, como en PSO. No debe confundirse con el concepto de vecindario en el espacio de soluciones que se usa en la búsqueda local o en metaheurísticas como SA o VNS.

y que están diseñadas para buscar valores óptimos en espacios complejos. Están basados en procesos biológicos que se pueden apreciar en la naturaleza, como la selección natural [68] o la herencia genética [203]. Parte de la evolución está determinada por la selección natural de individuos diferentes compitiendo por recursos en su entorno. Algunos individuos son mejores que otros y es deseable que aquellos individuos que son mejores sobrevivan y propaguen su material genético.

La reproducción sexual permite el intercambio del material genético de los cromosomas, produciendo así descendientes que contienen una combinación de la información genética de sus padres. Éste es el *operador de recombinación* utilizado en los EA, también llamado *operador de cruce*. La recombinación ocurre en un entorno en el que la selección de los individuos que tienen que emparejarse depende, principalmente, del valor de la función de *fitness* del individuo, es decir, de cómo de bueno es el individuo comparado con los de su entorno.

Como en el caso biológico, los individuos pueden sufrir mutaciones ocasionalmente (*operador de mutación*). La mutación es una fuente importante de diversidad para los EA. En un EA, se introduce normalmente una gran cantidad de diversidad al comienzo del algoritmo mediante la generación de una población de individuos aleatorios. La importancia de la mutación, que introduce aún más diversidad mientras el algoritmo se ejecuta, es objeto de debate. Algunos se refieren a la mutación como un operador de segundo plano, que simplemente reemplaza parte de la diversidad original que se haya podido perder a lo largo de la evolución, mientras que otros ven la mutación como el operador que juega el papel principal en el proceso evolutivo.

En la Figura 4.5 se muestra el esquema de funcionamiento de un EA típico. Como puede verse, un EA procede de forma iterativa mediante la evolución de los individuos pertenecientes a la población actual. Esta evolución es normalmente consecuencia de la aplicación de operadores estocásticos de variación sobre la población, como la selección, recombinación y mutación, con el fin de calcular una generación completa de nuevos individuos. El criterio de terminación consiste normalmente en alcanzar un número máximo de iteraciones (programado previamente) del algoritmo, o encontrar la solución óptima al problema (o una aproximación a la misma) en caso de que se conozca de antemano.

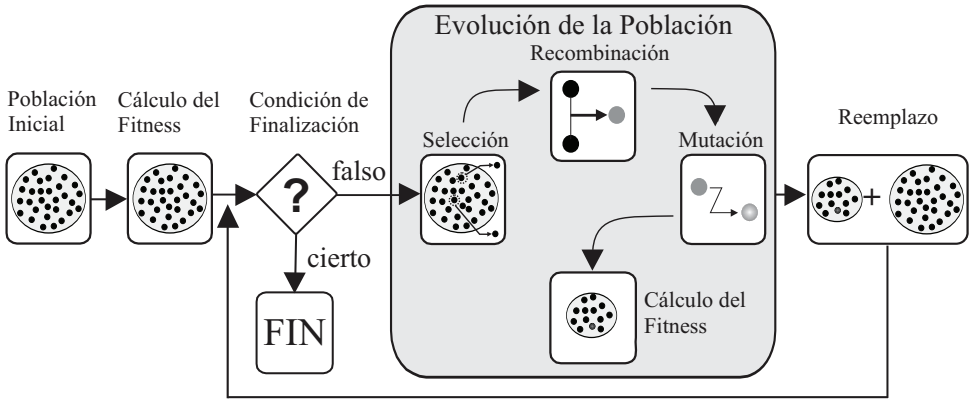


Figura 4.5: Funcionamiento de un EA canónico.

Pasaremos ahora a analizar detalladamente el funcionamiento de un algoritmo evolutivo, cuyo pseudocódigo se muestra en el Algoritmo 1. Como se ha dicho ya con anterioridad, los algoritmos evolutivos

trabajan sobre poblaciones de individuos, que son soluciones tentativas del problema. La población inicial está compuesta usualmente por individuos creados aleatoriamente, aunque también existe cierta tradición en el uso de técnicas de optimización (preferiblemente de poca carga computacional) para crear los individuos que formarán la población inicial, permitiendo así que el EA comience su ejecución sobre un conjunto de soluciones más prometedoras que las generadas aleatoriamente. Tras la generación de la población inicial, se calcula el valor de adecuación (*fitness*) de cada uno de los individuos que la forman y el algoritmo entra en el bucle reproductor. Este bucle consiste en la generación de una nueva población mediante la selección de padres, la recombinación de éstos y la mutación de los descendientes obtenidos. Tras este proceso, los individuos son evaluados. Esta nueva población generada por el bucle reproductor (P') se utilizará, junto con la población actual (P), para obtener la nueva población de individuos de la siguiente generación. Al final, el algoritmo devolverá la mejor solución encontrada durante la ejecución.

Como puede verse, el algoritmo comprende las tres fases principales: selección, reproducción y reemplazo. A continuación detallamos estas tres fases.

- **Selección:** partiendo de la población inicial P de μ individuos, se crea una nueva población temporal (P') de λ individuos. Generalmente los individuos más aptos (aquellos correspondientes a las mejores soluciones contenidas en la población) tienen un mayor número de instancias que aquellos que tienen menos aptitud (selección natural). De acuerdo con los valores de μ y λ podemos definir distintos esquemas de selección (Figura 4.6):

  1. **Selección por estado estacionario.** Cuando $\lambda = 1$ tenemos una selección por estado estacionario (*steady-state*) en la que únicamente se genera un hijo en cada paso de la evolución.
  2. **Selección generacional.** Cuando $\lambda = \mu$ tenemos una selección por generaciones en la que genera una nueva población completa de individuos en cada paso.
  3. **Selección ajustable.** Cuando $1 \leq \lambda \leq \mu$ tenemos una selección intermedia en la que se calcula un número ajustable (*generation gap*) de individuos en cada paso de la evolución. Los anteriores son casos particulares de éste.
  4. **Selección por exceso.** Cuando $\lambda > \mu$ tenemos una selección por exceso típica de los procesos naturales reales.

- **Reproducción:** en esta fase se aplican los operadores reproductivos a los individuos de la población P' para producir una nueva población. Típicamente, esos operadores se corresponden con la recombinación de parejas y con la mutación de los nuevos individuos generados. Estos operadores de

Algoritmo 1 Pseudocódigo de un algoritmo evolutivo

```
1: P = generarPoblaciónInicial();
2: evaluar(P);
3: while not condiciónParada() do
4:    P' = seleccionarPadres(P);
5:    P' = aplicarOperadoresDeVariación(P');
6:    evaluar(P');
7:    P = seleccionarNuevaPoblación(P, P');
8: end while
9: return la mejor solución encontrada
```

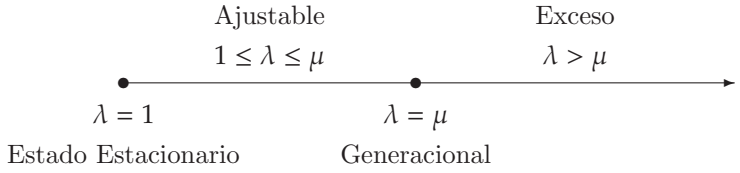


Figura 4.6: Diferencia entre los diversos esquemas de selección.

variación son, en general, no deterministas, es decir, no siempre se tienen que aplicar a todos los individuos y en todas las generaciones del algoritmo, sino que su comportamiento viene determinado por su probabilidad asociada.

- **Reemplazo:** finalmente, los individuos de la población original son sustituidos por los individuos recién creados. Este reemplazo usualmente intenta mantener los mejores individuos eliminando los peores. Dependiendo de si, para realizar el reemplazo, se tienen en cuenta la antigua población P podemos obtener dos tipos de estrategia de reemplazo:

  1. (μ, λ) si el reemplazo se realiza utilizando únicamente los individuos de la nueva población P'. Se debe cumplir que $\mu \leq \lambda$
  2. $(\mu + \lambda)$ si el reemplazo se realiza seleccionando μ individuos de la unión de P y P'.

Estos algoritmos establecen un equilibrio entre la explotación de buenas soluciones (fase de selección) y la exploración de nuevas zonas del espacio de búsqueda (fase de reproducción), basado en el hecho de que la política de reemplazo permite la aceptación de nuevas soluciones que no mejoran necesariamente las existentes.

Fueron cuatro los primeros tipos de algoritmos evolutivos que surgieron [149]. Estas cuatro familias de algoritmos fueron desarrolladas simultáneamente por distintos grupos de investigación. Los *algoritmos genéticos* (GA), fueron inicialmente estudiados por Holland [141, 142], en Ann Arbor (EEUU), Bremermann [44] en Berkeley (EEUU), y Fraser [101] en Sidney (Australia). Las *estrategias evolutivas* fueron propuestas por Rechenberg [232, 233] y Schwefel [248] en Berlin (Alemania), mientras que la *programación evolutiva* se propuso por primera vez por Fogel [100] en San Diego (California). Por último, la cuarta familia de algoritmos, la *programación genética*, surgió dos décadas más tarde, en 1985, como una adaptación, hecha por Cramer [66], de un algoritmo genético que trabajaba con genes en forma de árbol, además de las cadenas de caracteres binarios utilizadas tradicionalmente en GA.

Según la Definición 8, la principal diferencia de los EA con otros algoritmos metaheurísticos es la función Φ_{EA}, que se calcula a partir de otras tres funciones:

$$\Phi_{EA}(x,t,y) = \sum_{z \in \mathcal{T}^\lambda} \sum_{w \in \mathcal{T}^\lambda} \left(\omega_s(x,t,z) \cdot \omega_r(z,t,w) \cdot \prod_{j=1}^{\lambda} \omega_m(w_j,t,y_j) \right) , \tag{4.18}$$

donde:

- $\omega_s : \mathcal{T}^\mu \times \prod_{i=1}^{v} D_i \times \mathcal{T}^\lambda \to [0,1]$, representa el operador de selección de padres. Esta función debe cumplir para todo $x \in \mathcal{T}^\mu$ y para todo $t \in \prod_{i=1}^{v} D_i$

$$\sum_{y \in \mathcal{T}^\lambda} \omega_s(x,t,y) = 1 \ , \tag{4.19}$$

$$\forall y \in \mathcal{T}^\lambda, \omega_s(x,t,y) = 0 \vee \omega_s(x,t,y) > 0 \wedge (\forall i \in \{1,\dots,\lambda\}, \exists j \in \{1,\dots,\mu\}, y_i = x_j) \ . \tag{4.20}$$

- $\omega_r : \mathcal{T}^\lambda \times \prod_{i=1}^{v} D_i \times \mathcal{T}^\lambda \to [0,1]$, representa el operador de recombinación. Esta función debe cumplir para todo $x \in \mathcal{T}^\lambda$ y para todo $t \in \prod_{i=1}^{v} D_i$,

$$\sum_{y \in \mathcal{T}^\lambda} \omega_r(x,t,y) = 1 \ . \tag{4.21}$$

- $\omega_m : \mathcal{T} \times \prod_{i=1}^{v} D_i \times \mathcal{T} \to [0,1]$, representa el operador de mutación. Esta función debe cumplir para todo $x \in \mathcal{T}$ y para todo $t \in \prod_{i=1}^{v} D_i$,

$$\sum_{y \in \mathcal{T}} \omega_m(x,t,y) = 1 \ . \tag{4.22}$$

El siguiente resultado demuestra que la función Φ_{EA} cumple (4.5) y, por tanto, está bien definida.

Proposición 3. *La función Φ_{EA} definida en (4.18) cumple*

$$\sum_{y \in \mathcal{T}^\lambda} \Phi_{EA}(x,t,y) = 1 \ . \tag{4.23}$$

Demostración. La demostración consiste en una sencilla comprobación basada en las propiedades de ω_s, ω_r y ω_m que se muestran en (4.19), (4.21) y (4.22), respectivamente. En primer lugar, tenemos

$$\sum_{y \in \mathcal{T}^\lambda} \Phi_{EA}(x,t,y) = \sum_{y \in \mathcal{T}^\lambda} \sum_{z \in \mathcal{T}^\lambda} \sum_{w \in \mathcal{T}^\lambda} \left(\omega_s(x,t,z) \cdot \omega_r(z,t,w) \cdot \prod_{j=1}^{\lambda} \omega_m(w_j,t,y_j) \right) \ . \tag{4.24}$$

En virtud de las propiedades distributiva y conmutativa, podemos introducir los sumatorios dentro de la expresión entre paréntesis hasta colocarlos delante del término en el que aparece su índice, esto es,

$$\sum_{y \in \mathcal{T}^\lambda} \Phi_{EA}(x,t,y) = \sum_{z \in \mathcal{T}^\lambda} \omega_s(x,t,z) \cdot \left[\sum_{w \in \mathcal{T}^\lambda} \omega_r(z,t,w) \cdot \left(\sum_{y \in \mathcal{T}^\lambda} \prod_{j=1}^{\lambda} \omega_m(w_j,t,y_j) \right) \right] \ . \tag{4.25}$$

Podemos reescribir el término interior de la expresión anterior del siguiente modo

$$\sum_{y\in\mathcal{T}^\lambda}\prod_{j=1}^{\lambda}\omega_m(w_j,t,y_j)=\sum_{y\in\mathcal{T}^\lambda}\omega_m(w_1,t,y_1)\omega_m(w_2,t,y_2)\cdots\omega_m(w_\lambda,t,y_\lambda)=$$
$$\sum_{y_1\in\mathcal{T}}\sum_{y_2\in\mathcal{T}}\cdots\sum_{y_\lambda\in\mathcal{T}}\omega_m(w_1,t,y_1)\omega_m(w_2,t,y_2)\cdots\omega_m(w_\lambda,t,y_\lambda)=$$
$$\left(\sum_{y_1\in\mathcal{T}}\omega_m(w_1,t,y_1)\right)\left(\sum_{y_2\in\mathcal{T}}\omega_m(w_2,t,y_2)\right)\cdots\left(\sum_{y_\lambda\in\mathcal{T}}\omega_m(w_\lambda,t,y_\lambda)\right)=$$
$$\prod_{j=1}^{\lambda}\left(\sum_{y_j\in\mathcal{T}}\omega_m(w_j,t,y_j)\right)\ . \tag{4.26}$$

De esta forma, podemos escribir

$$\sum_{y\in\mathcal{T}^\lambda}\Phi_{EA}(x,t,y)=\sum_{z\in\mathcal{T}^\lambda}\omega_s(x,t,z)\cdot\left\{\sum_{w\in\mathcal{T}^\lambda}\omega_r(z,t,w)\cdot\left[\prod_{j=1}^{\lambda}\left(\sum_{y_j\in\mathcal{T}}\omega_m(w_j,t,y_j)\right)\right]\right\}\ . \tag{4.27}$$

En virtud de (4.22) tenemos $\sum_{y_j\in\mathcal{T}}\omega_m(w_j,t,y_j)=1$ y llegamos a

$$\sum_{y\in\mathcal{T}^\lambda}\Phi_{EA}(x,t,y)=\sum_{z\in\mathcal{T}^\lambda}\omega_s(x,t,z)\cdot\left(\sum_{w\in\mathcal{T}^\lambda}\omega_r(z,t,w)\cdot\right)\ . \tag{4.28}$$

Finalmente, usando las Ecuaciones (4.21) y (4.19) demostramos el enunciado.

■

Algoritmos genéticos

Los GA son un tipo de algoritmo evolutivo pensado inicialmente para trabajar con soluciones representadas mediante cadenas binarias denominadas *cromosomas*. No obstante, a lo largo de los años se han usado otras representaciones no binarias, como permutaciones [9], vectores de enteros [91], reales [136] e incluso estructuras de datos complejas y muy particulares de problemas determinados [264]. Los GA son los más conocidos entre los EA con diferencia, y esto ha provocado la aparición en la literatura de gran cantidad de variantes, de forma que es muy difícil dar una regla clara para caracterizarlos. A pesar de esto, podemos decir que la mayoría de los GA se caracterizan por poseer una representación lineal de las soluciones, no suelen incluir parámetros de autoadaptación en los individuos y dotan de mayor importancia al operador de recombinación que al de mutación.

Algunos de los operadores de recombinación más populares de los GA con representación binaria son el *cruce de un punto* (*Single Point Crossover*, SPX), el de *dos puntos* (*Double Point Crossover*, DPX), y el *cruce uniforme* (*Uniform Crossover*, UX). En SPX se divide el cromosoma de los padres en dos partes seleccionando un punto aleatorio y se intercambian los segmentos formados en la división generando dos nuevos individuos. En DPX se seleccionan dos puntos aleatorios del cromosoma de los padres y se intercambia el segmento delimitado por dichos puntos, dando lugar a dos nuevos individuos. Por último, en UX cada bit del cromosoma hijo se copia de uno de los dos padres. UX posee un parámetro llamado

bias que es la probabilidad de tomar el bit del mejor padre. Como resultado de su operación devuelve un único individuo. Aunque los tres operadores se definieron para cadenas binarias, se pueden aplicar a cualquier tipo de representación lineal, como vectores de enteros o reales.

El operador de mutación tradicional para algoritmos genéticos con representación binaria es la *inversión de bits*. Este operador recorre la cadena de bits invirtiéndolos con una determinada probabilidad, que es un parámetro del operador.

Estrategias evolutivas

En una ES [233] cada individuo está formado por tres componentes: las variables del problema ($\mathbf{x}$), un vector de desviaciones estándar (σ) y, opcionalmente, un vector de ángulos (ω). Estos dos últimos vectores se usan como parámetros para el principal operador de la técnica: la mutación gaussiana. Los vectores de desviaciones y de ángulos evolucionan junto con las variables del problema, permitiendo, de este modo, que el algoritmo se autoadapte conforme avanza la búsqueda.

A diferencia de los GA, en las ES el operador de recombinación tiene un papel secundario. De hecho, en la versión original de ES no existía dicho operador. Fue más tarde cuando se introdujo la recombinación en las ES y se propusieron varias alternativas. Además, cada componente de un individuo puede utilizar un mecanismo diferente de recombinación, dando lugar a una gran cantidad de posibilidades para la recombinación de individuos.

El operador de mutación está gobernado por las siguientes tres ecuaciones:

$$\sigma_i' = \sigma_i \exp(\tau N(0,1) + \eta N_i(0,1)) \ , \tag{4.29}$$

$$\omega_i' = \omega_i + \varphi N_i(0,1) \ , \tag{4.30}$$

$$\mathbf{x}' = \mathbf{x} + \mathbf{N}(\mathbf{0}, C(\sigma', \omega')) \ , \tag{4.31}$$

donde $C(\sigma', \omega')$ es la matriz de covarianza asociada con σ' y ω', $N(0,1)$ es la distribución normal de una variable y $\mathbf{N}(\mathbf{0}, C)$ es la distribución normal multivariable con media $\mathbf{0}$ y matriz de covarianza C. El subíndice i en la distribución normal indica que se genera un nuevo número aleatorio para cada componente del vector. La notación $N(0,1)$ se usa para indicar que se usa el mismo número aleatorio para todos los componentes. Los parámetros τ, η, y φ toman valores $(2n)^{-1/2}$, $(4n)^{-1/4}$ y $5\pi/180$, respectivamente, tal y como se sugiere en [240].

4.4.2. Optimización basada en cúmulos de partículas

El algoritmo PSO mantiene un conjunto de soluciones (vectores de reales), también llamadas *partículas*, que son inicializadas aleatoriamente en el espacio de búsqueda. El movimiento de una partícula $\mathbf{x}^i$ depende de su velocidad $\mathbf{v}^i$ (otro vector de reales) y de las posiciones donde se encontraron buenas soluciones en el pasado, tanto por parte de la partícula considerada como por el vecindario de dicha partícula. En la versión del PSO con vecindario global, la velocidad de una partícula depende de la mejor solución global que ha sido encontrada. Por otro lado, en la versión con vecindario local cada partícula tiene asociado un conjunto de partículas vecinas y la velocidad se actualiza de acuerdo a la mejor solución encontrada por alguna partícula de dicho vecindario. La actualización de la velocidad y la posición de cada partícula viene dada por las siguientes ecuaciones:

$$v_j^i(t+1) = w \cdot v_j^i(t) + c_1 \cdot r_1 \cdot (p_j^i - x_j^i(t)) + c_2 \cdot r_2 \cdot (n_j^i - x_j^i(t)) \ , \tag{4.32}$$

$$x_j^i(t+1) = x_j^i(t) + v_j^i(t+1) \ , \tag{4.33}$$

donde w es la *inercia* y controla la influencia de la velocidad anterior, c_1 y c_2 permiten ajustar la influencia de la *mejor solución personal* ($\mathbf{p}^i$) y la *mejor solución del vecindario* ($\mathbf{n}^i$), y los valores r_1 y r_2 son números reales aleatorios dentro del intervalo $[0,1]$ generados en cada iteración. La mejor solución personal $\mathbf{p}^i$ de una partícula i es la posición con mejor valor de *fitness* encontrada hasta el momento por la partícula i. Por otro lado, la mejor solución del vecindario $\mathbf{n}^i$ de la partícula i es la posición con mejor valor de *fitness* encontrada por las partículas en el vecindario de i (N_i). El valor de w tiene que estar por debajo de 1 para que el algoritmo converja. Un valor alto de w hace que las partículas tiendan a explorar nuevas áreas (diversificación) y un valor bajo hace que busquen cerca de soluciones previas (intensificación). En el Algoritmo 2 podemos ver el pseudocódigo de un PSO.

Algoritmo 2 Optimización basada en cúmulos de partículas

```
1: inicializarPartículas(x,v); {Inicializar las posiciones y velocidades de las partículas}
2: repeat
3:    actualizarMejorPersonal(x,p);
4:    actualizarVecindario(x,p,n);
5:    actualizarPartículas(x,v) {Actualizar la posición y velocidad de las partículas}
6: until condiciónParada()
7: return la mejor solución encontrada
```

Según la Definición 8, esta metaheurística cumple $\mu_{PSO} = \lambda_{PSO}$. El dominio $\mathcal{T}$ está formado por pares de vectores de reales con el mismo número de componentes. Uno de dichos vectores representa la posición y el otro la velocidad de cada partícula. El conjunto Ξ_{PSO} contiene μ_{PSO} variables para representar la mejor solución personal de cada partícula. Por otro lado, la función σ_{PSO} se define del siguiente modo:

$$\sigma_{PSO}(x,y,t,z) = \begin{cases} 1 & \text{si } y = z \\ 0 & \text{en otro caso} \end{cases} \ , \tag{4.34}$$

donde $x,y,z \in \mathcal{T}^{\mu_{PSO}}$ y $t \in \prod_{i=1}^{v} D_i$. La forma en que se modifica la posición y la velocidad de las partículas tiene su reflejo en la definición de $\Phi_{PSO} : \mathcal{T}^{\mu_{PSO}} \times \prod_{i=1}^{v} D_i \times \mathcal{T}^{\mu_{PSO}} \to [0,1]$, que en el caso de usar un vecindario global para las partículas será:

$$\Phi_{PSO}(x,t,y) = \prod_{j=1}^{\mu_{PSO}} \omega_u(x_j,t,y_j) \ , \tag{4.35}$$

donde $x,y \in \mathcal{T}^{\mu_{PSO}}$, $t \in \prod_{i=1}^{v} D_i$ y la función $\omega_u : \mathcal{T} \times \prod_{i=1}^{v} D_i \times \mathcal{T} \to [0,1]$ representa la operación de actualización de una partícula y debe cumplir para todo $x \in \mathcal{T}$ y todo $t \in \prod_{i=1}^{v} D_i$,

$$\sum_{y \in \mathcal{T}} \omega_u(x,t,y) = 1 \ . \tag{4.36}$$

La siguiente proposición demuestra que la función Φ_{PSO} cumple (4.5) y, por tanto, está bien definida.

Proposición 4. *La función Φ_{PSO} definida en (4.35) cumple*

$$\sum_{y\in\mathcal{T}^\lambda} \Phi_{PSO}(x,t,y) = 1 \ . \tag{4.37}$$

Demostración. La demostración consiste en una sencilla comprobación basada en la propiedad de ω_u que se muestra en (4.36). Tendremos en cuenta que $\lambda = \mu$ y usaremos únicamente λ. Aclarado esto, podemos escribir el siguiente desarrollo:

$$\begin{aligned}
&\sum_{y\in\mathcal{T}^\lambda} \Phi_{PSO}(x,t,y) = \sum_{y\in\mathcal{T}^\lambda} \prod_{j=1}^{\lambda} \omega_u(x_j,t,y_j) = \\
&\sum_{y\in\mathcal{T}^\lambda} \omega_u(x_1,t,y_1)\omega_u(x_2,t,y_2)\cdots\omega_u(x_\lambda,t,y_\lambda) = \\
&\sum_{y_1\in\mathcal{T}} \sum_{y_2\in\mathcal{T}} \dots \sum_{y_\lambda\in\mathcal{T}} \omega_u(x_1,t,y_1)\omega_u(x_2,t,y_2)\cdots\omega_u(x_\lambda,t,y_\lambda) = \\
&\left(\sum_{y_1\in\mathcal{T}} \omega_u(x_1,t,y_1)\right)\left(\sum_{y_2\in\mathcal{T}} \omega_u(x_2,t,y_2)\right)\cdots\left(\sum_{y_\lambda\in\mathcal{T}} \omega_u(x_\lambda,t,y_\lambda)\right) = \\
&\prod_{j=1}^{\lambda}\left(\sum_{y_j\in\mathcal{T}} \omega_u(x_j,t,y_j)\right) = 1 \ ,
\end{aligned} \tag{4.38}$$

lo que demuestra el enunciado. ■

4.4.3. Optimización basada en colonias de hormigas

Los algoritmos basados en colonias de hormigas (ACO) son un tipo de metaheurística basada en población cuya filosofía está inspirada en el comportamiento de las hormigas reales cuando buscan comida [37]. La idea principal consiste en usar hormigas artificiales que simulan dicho comportamiento en un escenario también artificial: un grafo. Las hormigas artificiales se colocan en nodos iniciales de dicho grafo y lo recorren saltando de un nodo a otro con la meta de encontrar el camino más corto desde su nodo inicial hasta un nodo final u objetivo. Cada hormiga avanza de forma independiente a las demás, pero la decisión del siguiente nodo a visitar depende de ciertos valores numéricos asociados a los arcos o nodos del grafo. Estos valores modelan los *rastros de feromona* que depositan las hormigas reales cuando caminan. Las hormigas artificiales alteran (al igual que las hormigas reales) los rastros de feromona del camino trazado, de modo que el avance de una puede influir en el camino de otra. De esta forma se incorpora a los ACO un mecanismo de cooperación indirecta entre las distintas hormigas simuladas, que constituye un factor clave en la búsqueda [88].

En un ACO las soluciones candidatas se representan mediante una secuencia de *componentes* escogidos de un conjunto de componentes C. De hecho, una cuestión importante cuando se resuelve un problema de optimización con un ACO es la elección del conjunto de componentes C y el modo en que las soluciones se construyen usando estos componentes. En algunos problemas esta cuestión es trivial porque las soluciones son ya una secuencia de elementos. Este es el caso del problema del viajante de comercio

(*Travelling Salesman Problem*, TSP), donde una solución es una secuencia de ciudades. Para otros problemas, como el entrenamiento de redes neuronales, la representación no es tan directa [257]. Las hormigas artificiales construyen la solución usando un procedimiento constructivo estocástico. Durante esta fase de construcción las hormigas caminan aleatoriamente por un grafo $G = (C, L)$ llamado *grafo de construcción*, donde L es un conjunto de *conexiones* (arcos) entre los componentes (nodos) de C. Cada arco $l_{ij} \in L$ tiene asociado un rastro de feromona τ_{ij} y puede tener asociado también un valor heurístico η_{ij}. Ambos valores se usan para guiar la fase de construcción estocástica que realizan las hormigas, pero hay una diferencia importante entre ellos: los rastros de feromona se modifican a lo largo de la búsqueda, mientras que los heurísticos son valores fijos establecidos por fuentes externas al algoritmo (el diseñador). Los rastros de feromona pueden asociarse también a los nodos del grafo de construcción (componentes de la solución) en lugar de a los arcos (conexiones entre componentes). Esta variación es especialmente adecuada para problemas en los que el orden de los componentes no es relevante (problemas de subconjunto [176]).

El Algoritmo 3 muestra el pseudocódigo de un ACO general. Al principio se inicializan los rastros de feromona (línea 1) y se construye una solución para asignarla a la variable que contendrá la mejor solución encontrada (ant^{bs}). Seguidamente, se ejecuta un bucle que se repite hasta que se cumpla una cierta condición de parada (como, por ejemplo, alcanzar un número prefijado de iteraciones). Dentro del bucle, cada hormiga de la colonia construye una solución, que consiste en un camino en el grafo de construcción (línea 5). La hormiga comienza en un nodo inicial y elige el siguiente nodo de acuerdo con el rastro de feromona y el heurístico asociado con cada arco (o nodo). La hormiga añade el nuevo nodo al camino recorrido y selecciona el siguiente nodo de la misma forma. Este proceso se repite hasta que se construye una solución candidata. Tras la construcción de una solución, en una variante de ACO denominada ACS (Ant Colony System) se lleva a cabo una actualización de los rastros de feromona asociados a los arcos por los que ha pasado la hormiga (línea 6). Hemos incluido dicha actualización en el pseudocódigo para que éste sea lo más general posible. Podemos asumir que, en otras variantes de ACO, el procedimiento *actualizarFeromonasLocal* no hace nada. Cuando todas las hormigas han construido sus caminos se actualizan los rastros de feromona en dos pasos: en primer lugar son evaporados (línea 8) y, tras esto, se incrementa la cantidad de feromona en los arcos de las soluciones más prometedoras encontradas (línea 9). La cantidad de feromona que se deposita en cada arco suele depender de la calidad de las soluciones que pasan por dicho arco. En cada variante de ACO estos dos procedimientos se realizan de manera diferente.

Lo más significativo de la formalización de esta metaheurística siguiendo la Definición 8 es que $\mu_{ACO} = 0$ y existe una variable de estado de la metaheurística, $\mathcal{P} \in first(\Xi_{ACO})$ que representará la matriz de feromona y la heurística. Llamaremos $D_{\mathcal{P}}$ al dominio de esta variable. Este dominio está formado por todos los posibles grafos de construcción con arcos y/o nodos ponderados con valores reales para la representar los rastros de feromonas y, opcionalmente, heurística. En este caso, las función Φ_{ACO} tendrá la siguiente forma:

$$\Phi_{ACO} : D_{\mathcal{P}} \times \prod_{i=2}^{v} D_i \rightarrow \mathcal{T}^{\lambda} \ , \tag{4.39}$$

donde hemos supuesto que la variable $\xi_1 = \mathcal{P}$. También tiene especial importancia en esta metaheurística la función $\mathcal{U}_{ACO}$, que realiza la actualización de los rastros de feromona.

El esquema presentado arriba es muy abstracto. Es suficientemente general para encajar con los distintos modelos de algoritmos ACO que podemos encontrar en la literatura. Estos modelos difieren en el modo en que planifican los procedimientos principales y en cómo actualizan los rastros de feromonas. Algunos ejemplos de estos modelos son los sistemas de hormigas (*Ant Systems*, AS), sistemas elitistas

Algoritmo 3 Optimización basada en colonias de hormigas

```
1: inicializarFeromonas(τ); {Inicializa los rastros de feromona}
2: ant^bs = generarSolución(τ, η); {Se inicializa ant^bs con una solución inicial aleatoria}
3: repeat
4:    for i=1 to μ do
5:       ant^i = generarSolución(τ, η); {La hormiga ant^i construye una solución}
6:       actualizarFeromonasLocal(τ, ant^i); {Actualización local de los rastros de feromona (ACS)}
7:    end for
8:    evaporarFeromonas(τ); {Evaporación de los rastros de feromona}
9:    actualizarFeromonasGlobal(τ,ant,ant^bs); {Actualización global de los rastros de feromona}
10:   for i=1 to μ do
11:      if f(ant^i) > f(ant^bs) then
12:         ant^bs = ant^i; {Actualizar la mejor solución}
13:      end if
14:   end for
15: until condiciónParada()
16: return la mejor solución encontrada
```

de hormigas (*Elitist Ant Systems*, EAS), sistemas de hormigas basados en rango (*Ranked-Based Ant Systems*, Rank-AS), y sistemas de hormigas $\mathcal{MAX}-\mathcal{MIN}$ ($\mathcal{MAX}-\mathcal{MIN}$ *Ant Systems*, $\mathcal{MM}$AS). El lector interesado puede ver el libro de Dorigo y Stützle [88] para una descripción de todas estas variantes de ACO. En esta tesis doctoral no usaremos ninguno de estos modelos, sino que proponemos uno nuevo, llamado ACOhg, que se presentará en la Sección 5.3. ACOhg extiende los modelos de ACO existentes introduciendo nuevas ideas y mecanismos para trabajar con grafos de gran dimensión. Es decir, un algoritmo ACOhg se puede basar en un algoritmo ACO existente tal como AS, $\mathcal{MM}$AS, etc., dando lugar al correspondiente AShg, $\mathcal{MM}$AShg, etc. El algoritmo ACOhg usado en esta tesis se basa en un $\mathcal{MM}$AS al que se le ha añadido el mecanismo de actualización local de feromonas de ACS para aumentar su capacidad de exploración. Aunque describiremos el algoritmo con detalle en la Sección 5.3, en el resto de esta sección describimos la fase de construcción y de actualización de feromona que se usa en ACOhg, por ser común a otros modelos de ACO.

Fase de construcción

Como mencionamos arriba, en la fase de construcción las hormigas seleccionan estocásticamente el siguiente nodo en el grafo para formar la solución. En concreto, cuando la hormiga k está en el nodo i selecciona el nodo j con probabilidad

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{x \in L(i)} [\tau_{ix}]^\alpha [\eta_{ix}]^\beta}, \text{ donde } j \in L(i) \ , \tag{4.40}$$

donde $L(i)$ es el conjunto de nodos sucesores del nodo i, y α y β son dos parámetros del algoritmo que determinan la influencia relativa del rastro de feromona y el valor heurístico en la construcción del camino, respectivamente (véase la Figura 4.7).

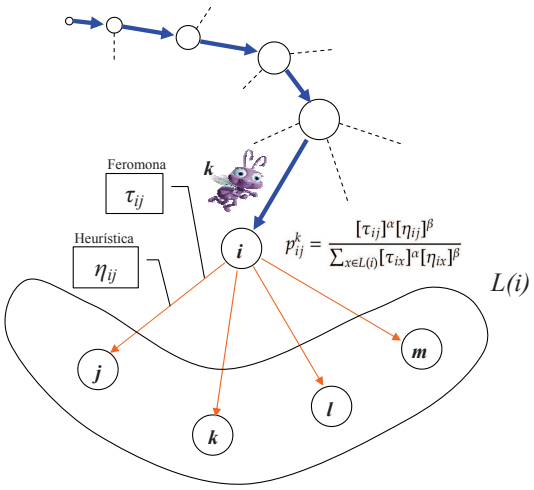


Figura 4.7: Una hormiga durante la fase de construcción.

Actualización de feromona

Durante la fase de construcción, los rastros de feromona asociados con los arcos que las hormigas recorren se actualizan usando la expresión

$$\tau_{ij} \leftarrow (1 - \xi)\tau_{ij} \ , \tag{4.41}$$

donde ξ controla la evaporación de la feromona durante la fase de construcción y cumple $0 < \xi < 1$. Este mecanismo, propio de ACS, aumenta la exploración del algoritmo, ya que reduce la probabilidad de que una hormiga siga el camino de una hormiga anterior.

Tras la fase de construcción, los rastros de feromona se evaporan parcialmente utilizando la expresión

$$\tau_{ij} \leftarrow (1 - \rho)\tau_{ij} \ , \forall (i, j) \in L \ , \tag{4.42}$$

donde ρ es la *tasa de evaporación de feromonas* y debe cumplir $0 < \rho \leq 1$. Finalmente, en los arcos correspondientes a la mejor solución encontrada hasta el momento, se deposita una cantidad de feromona que depende de la calidad de la solución

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{bs}, \ \forall (i, j) \in L \ , \tag{4.43}$$

donde $\Delta\tau_{ij}^{bs}$ es la cantidad de feromona que la hormiga asociada a la mejor solución encontrada deposita en el arco (i, j). En un problema de maximización, este valor es el que toma la función de *fitness* en dicha solución. En un problema de minimización (como el que abordamos en esta tesis) es la inversa del valor de *fitness* de la solución.

En $\mathcal{MM}$AS [88] hay un mecanismo para evitar la convergencia prematura del algoritmo que también usamos aquí. La idea es mantener el valor de los rastros de feromona del algoritmo en un intervalo dado $[\tau_{min}, \tau_{max}]$ para mantener por encima de un determinado umbral la probabilidad de seleccionar un nodo. Los valores de las cotas para los rastros de feromona son

$$\tau_{max} = \frac{Q}{\rho} \ , \tag{4.44}$$

$$\tau_{min} = \frac{\tau_{max}}{a} \ , \tag{4.45}$$

donde Q es el mayor valor de *fitness* encontrado si el problema es de maximización o la inversa del mínimo valor de *fitness* si el problema es de minimización. El parámetro a controla el tamaño del intervalo.

Cuando un rastro de feromona es mayor que τ_{max} se modifica su valor a τ_{max} y, del mismo modo, cuando es menor que τ_{min} se le asigna τ_{min}. Cada vez que se encuentra una solución mejor, los valores τ_{max} y τ_{min} se actualizan consecuentemente y todas los rastros de feromonas se comprueban para mantenerlos dentro del intervalo $[\tau_{min}, \tau_{max}]$.

4.5. Conclusiones

En este capítulo hemos ofrecido una introducción al campo de las metaheurísticas. En primer lugar, hemos dado las definiciones previas y hemos introducido una extensión de la definición formal de metaheurística propuesta por Gabriel Luque en su tesis doctoral [183].

En segundo lugar, hemos incluido un repaso por las técnicas más importantes y populares dentro de este campo. Estas breves descripciones han sido realizadas siguiendo una clasificación de las metaheurísticas que las dividen en dos clases, atendiendo al número de soluciones tentativas con la que trabajan en cada iteración: metaheurísticas basadas en trayectoria y en población. Tras este repaso hemos introducido las metaheurísticas paralelas indicando las distintas formas de paralelismo consideradas en la literatura para cada clase de metaheurística (trayectoria y población).

Por último, hemos descrito con detalle los algoritmos metaheurísticos utilizados en esta tesis. Estas descripciones han sido generales y se han ocultado detalles que dependen de la aplicación concreta a los problemas. Estos detalles serán desvelados en el capítulo de propuestas metodológicas.

Parte II

Resolución de los problemas de Ingeniería del Software seleccionados

Capítulo 5

Propuestas metodológicas

En este capítulo concentramos las aportaciones metodológicas realizadas en esta tesis doctoral. Hemos dividido la descripción de las propuestas en tres secciones que se corresponden con los tres problemas de Ingeniería del Software abordados. En la Sección 5.1 describimos los detalles de la metodología empleada para resolver el problema de planificación de proyectos software. En segundo lugar, la propuesta realizada para la generación automática de casos de prueba se analiza en la Sección 5.2. Por último, los detalles de las técnicas empleadas para resolver el problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes se describen en la Sección 5.3.

5.1. Propuesta para la planificación de proyectos software

El objetivo en el problema de planificación de proyectos software es estudiar las ventajas de aplicar una técnica metaheurística simple, como es un algoritmo genético con representación binaria, a la resolución de diferentes instancias del problema. Nos proponemos mostrar que dicha técnica, aún siendo simple, permite a un gestor de proyectos software ensayar distintas configuraciones para los proyectos.

Con el objetivo de aplicar el algoritmo genético a un gran conjunto de instancias, hemos desarrollado un generador de instancias para este problema. Dicho generador se encuentra descrito en la Sección 5.1.1. Por otro lado, los detalles de la representación, operadores y función de *fitness* utilizados en el algoritmo genético se describen en la Sección 5.1.2.

5.1.1. Generador de instancias

Para realizar un estudio significativo debemos analizar varias instancias del problema de planificación en lugar de centrarnos sólo en una, lo cual podría llevarnos a conclusiones específicas de la instancia y no del problema en general. Para hacer esto, hemos desarrollado un generador de instancias que crea proyectos software ficticios a partir de un conjunto de parámetros tales como el número de tareas, el número de empleados, etc. Un generador de instancias es una aplicación fácilmente configurable que genera instancias con la dificultad deseada. El uso del generador de instancias elimina la posibilidad de ajustar los algoritmos para una instancia particular, permitiendo una comparación más justa entre diferentes algoritmos. Con un generador de instancias los algoritmos se pueden evaluar sobre un gran número

de instancias del problema y, consecuentemente, aumenta el poder predictivo de los resultados para el problema en general. En esta sección describiremos con detalle el generador de instancias desarrollado.

Los componentes de nuestras instancias son: empleados, tareas, habilidades, y el grafo de precedencia de tareas (TPG). Cada uno de estos componentes tiene varios parámetros que debe determinar el generador de instancias. Hay dos tipos de valores a generar: valores numéricos simples y conjuntos. Para los valores numéricos, el usuario especifica una distribución de probabilidad y los valores se generan muestreando dicha distribución. En el caso de conjuntos, el usuario proporciona una distribución de probabilidad para su cardinalidad. Una vez generado el valor para la cardinalidad del conjunto, los elementos de éste se eligen aleatoriamente de su universo hasta formar un conjunto con la cardinalidad indicada.

Todas las distribuciones de probabilidad se especifican en un fichero de configuración. Éste es un fichero de texto plano que contiene pares atributo-valor. Podemos ver un fichero de ejemplo en la Figura 5.1. Cada parámetro de la instancia tiene un nombre clave en el fichero de configuración. Estos nombres se incluyen en la Tabla 5.1. El valor asociado a un nombre clave es una cadena de caracteres que representa el nombre de la distribución de probabilidad que se usará para generar el valor del parámetro. Las distribuciones de probabilidad tienen parámetros que se especifican con pares atributo-valor adicionales de la forma: `<clave>.parameter.<param> = <valor>`. Por ejemplo, la propiedad *employee.skill* en el fichero de ejemplo de la Figura 5.1 indica que el empleado tiene 6 o 7 de las 10 habilidades posibles (propiedad *skill.number*).

Tabla 5.1: Atributos del fichero de configuración y sus parámetros asociados.

Clave	Parámetro
task.number	Número de tareas
task.cost	Esfuerzo de las tareas
task.skill	Número de habilidades requeridas de las tareas
employee.number	Número de empleados
employee.salary	Salario de los empleados
employee.skill	Número de habilidades de los empleados
employee.maxded	Máxima dedicación de los empleados
graph.e-v-rate	Fracción arcos/vértices del TPG
skill.number	Cardinalidad del conjunto de habilidades
random.seed	Semilla aleatoria (opcional)

El generador de instancias lee el fichero de configuración y genera las habilidades, las tareas, el TPG, y los empleados, en ese orden. Para cada tarea, genera el valor de esfuerzo y el conjunto de habilidades requeridas. Para cada empleado genera el salario y el conjunto de habilidades. El pseudocódigo del generador de instancias se muestra en el Algoritmo 4.

Los valores numéricos de una instancia son: el número de tareas, el esfuerzo de las tareas, el número de empleados, el salario de los empleados, la máxima dedicación de los empleados y el número de habilidades. Los conjuntos de una instancia son: las habilidades requeridas de las tareas, las habilidades de los empleados, y el conjunto de arcos del TPG. Para el conjunto de arcos no especificamos una distribución para la cardinalidad directamente, sino para el cociente arcos/vértices, es decir, el valor numérico generado se multiplica por el número de tareas para obtener el número de arcos del TPG.

Utilizaremos el generador de instancias con diferentes configuraciones, es decir, diferente número de tareas, empleados, y habilidades. La dificultad de las instancias depende de estos parámetros. Por ejemplo, esperamos que las instancias con mayor número de tareas sean más difíciles que aquéllas con un conjunto

```
# Fichero de configuración para el generador de instancias

task.number = UniformInt
task.number.parameter.minvalue = 30
task.number.parameter.maxvalue = 30

task.cost = Round
task.cost.parameter.distribution = Normal
task.cost.parameter.distribution.parameter.mu = 10
task.cost.parameter.distribution.parameter.sigma = 5

task.skill = UniformInt
task.skill.parameter.minvalue = 2
task.skill.parameter.maxvalue = 3

graph.e-v-rate = Normal
graph.e-v-rate.parameter.mu = 1.5
graph.e-v-rate.parameter.sigma = 0.5

employee.number = UniformInt
employee.number.parameter.minvalue = 15
employee.number.parameter.maxvalue = 15

employee.salary = Normal
employee.salary.parameter.mu = 10000
employee.salary.parameter.sigma = 1000

employee.skill = UniformInt
employee.skill.parameter.minvalue = 6
employee.skill.parameter.maxvalue = 7

employee.maxded = UniformInt
employee.maxded.parameter.minvalue = 1
employee.maxded.parameter.maxvalue = 1

skill.number = UniformInt
skill.number.parameter.minvalue = 10
skill.number.parameter.maxvalue = 10
```

Figura 5.1: Un ejemplo de fichero de configuración para el generador de instancias.

más pequeño, como en los proyecto reales. Esto es de sentido común, ya que es más difícil hacer más trabajo con el mismo número de empleados (sin trabajar horas extra). Siguiendo este razonamiento, cuando incrementemos el número de empleados mientras mantenemos el número de tareas esperamos que el generador produzca instancias más sencillas. No obstante, estas reglas no se cumplen a veces en proyectos software complejos, ya que existen otros parámetros que tienen una influencia en la dificultad de una instancia. Uno de estos parámetros es el TPG: con el mismo número de tareas, un proyecto puede ser abordado con menos empleados en el mismo tiempo que otro proyecto con diferente TPG.

Por otro lado, si comparamos las instancias con el mismo número de tareas esperamos que, conforme el número de empleados aumente, el proyecto dure menos. No obstante, con un incremento en el número de empleados identificamos dos efectos opuestos asociados al coste: con más empleados trabajando, el coste mensual aumenta; pero al mismo tiempo la duración del proyecto se reduce y, con ello, el coste total del proyecto. Por tanto, no podemos concluir nada *a priori* acerca del coste del proyecto a partir del número de empleados.

Con respecto al número de habilidades del proyecto, esperamos que las instancias que tengan un mayor número de habilidades sean más difíciles de resolver. Con más habilidades, tenemos más empleados

Algoritmo 4 Pseudocódigo del generador de instancias

S = sample (skill.number); {*sample* genera un número aleatorio siguiendo la distribución indicada}
$SK = \{1,\ldots,S\}$;
T = sample (task.number);
for $i = 1$ **to** T **do**
  t_i^{effort} = sample (task.cost);
  $t_i^{skills} = \emptyset$;
  $card$ = sample (task.skill);
  for $j = 1$ **to** $card$ **do**
    s = random (SK/t_i^{skills}); {*random* devuelve un elemento aleatorio del conjunto indicado}
    $t_i^{skills} = t_i^{skills} \cup \{s\}$;
  end for
end for
$evrate$ = sample (graph.e-v-rate);
$A = \emptyset$;
for $i = 1$ **to** $evrate * T$ **do**
  $edge$ = random ($\{(t_a, t_b) | 1 \leq a < b \leq T\}/A$);
  $A = A \cup \{edge\}$;
end for
E = sample (employee.number);
for $i = 1$ **to** E **do**
  e_i^{salary} = sample (employee.salary);
  e_i^{maxded} = sample (employee.maxded);
  $e_i^{skills} = \emptyset$;
  $card$ = sample (employee.skill);
  for $j = 1$ **to** $card$ **do**
    s = random (SK/e_i^{skills});
    $e_i^{skills} = e_i^{skills} \cup \{s\}$;
  end for
end for

especializados y esperamos necesitar más empleados para cubrir las habilidades requeridas de las tareas. Por esto, los empleados trabajan en más tareas y probablemente algunos de ellos pueden exceder su grado de dedicación máximo haciendo que la solución no sea factible. Todas estas cuestiones hacen que sea muy importante para el gestor de proyectos tener una herramienta automática para tomar decisiones.

5.1.2. Detalles del GA

En esta sección discutimos la representación de la solución y la función de *fitness* usada en el algoritmo genético. Como dijimos en la Sección 3.1.1, una solución al problema es una matriz $\mathbf{X}$ cuyos elementos x_{ij} son números reales no negativos y representan el grado de dedicación del empleado e_i a la tarea t_j. En todas las instancias generadas para realizar los experimentos del Capítulo 6 consideramos que ningún empleado trabaja horas extra, así que la dedicación máxima de todos los empleados es 1. Por esta razón, el valor máximo para x_{ij} es 1, con lo que $x_{ij} \in [0, 1]$. Por otro lado, usamos un GA con cromosomas binarios

para representar las soluciones del problema. Por esto, necesitamos discretizar el intervalo $[0,1]$ para codificar el grado de dedicación x_{ij}. Distinguimos ocho valores en este intervalo que están uniformemente distribuidos. Para la representación usamos tres bits por cada elemento de la matriz $\mathbf{X}$ y almacenamos ésta por filas en el cromosoma[1] $\mathbf{x}$. La longitud del cromosoma es el número de empleados por el número de tareas multiplicado por tres, es decir, $3 \cdot E \cdot T$. La Figura 5.2 muestra la representación usada.

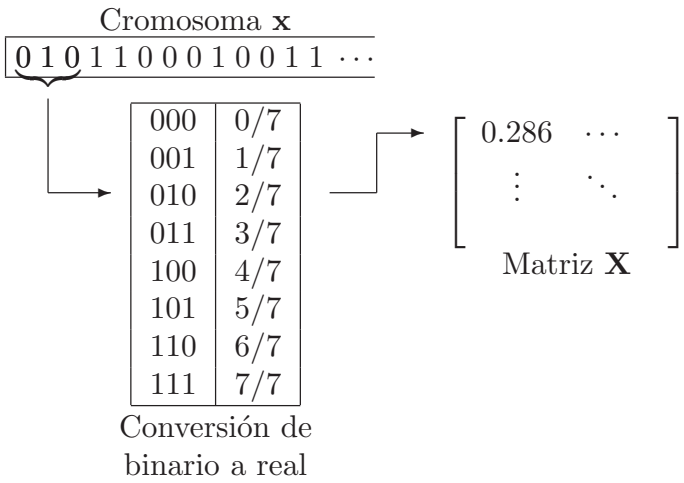


Figura 5.2: Representación de una solución en el algoritmo genético.

Aunque para el algoritmo genético una solución es una cadena de bits, en realidad dicha cadena representa una matriz. Por este motivo, en lugar de utilizar un operador de recombinación tradicional para cadenas de bits, como SPX, DPX o UX, hemos optado por emplear un operador específico para cadenas binarias que representan matrices o tablas: el *operador de recombinación de un punto para tablas* [278]. Este operador selecciona aleatoriamente una fila y una columna (la misma en ambos padres) y posteriormente cambia los elementos del cuadrante superior izquierdo e inferior derecho de ambos individuos (véase la Figura 5.3). En nuestro caso, el operador sólo puede seleccionar columnas múltiplo de tres, para no romper las secuencias de bits que representan cada valor x_{ij} de la solución. Como operador de mutación utilizamos la inversión de bits.

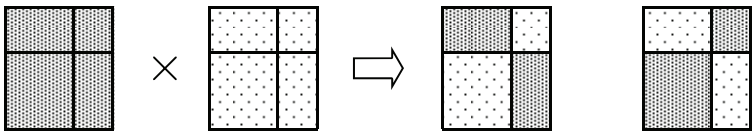


Figura 5.3: Recombinación de un punto para tablas.

Para calcular el *fitness* de un cromosoma $\mathbf{x}$ usamos la siguiente expresión:

$$f(\mathbf{x}) = \begin{cases} 1/q & \text{si la solución es factible} \\ 1/(q+p) & \text{en otro caso ,} \end{cases} \tag{5.1}$$

[1]No se debe confundir el cromosoma $\mathbf{x}$ con la matriz $\mathbf{X}$ a la que representa.

donde

$$q = w_{cost} \cdot p_{cost} + w_{dur} \cdot p_{dur} \ , \tag{5.2}$$

y

$$p = w_{penal} + w_{undt} \cdot undt + w_{reqsk} \cdot reqsk + w_{over} \cdot p_{over} \ . \tag{5.3}$$

La función de *fitness* tiene dos términos: el coste de la solución (q) y la penalización para soluciones no factibles (p). Ambos términos aparecen en el denominador porque el objetivo es minimizarlos, es decir, maximizar $f(\mathbf{x})$. El primer término es la suma ponderada del coste del proyecto (p_{cost}) y su duración (p_{dur}). En este término, w_{cost} y w_{dur} son valores que ponderan la importancia relativa de los dos objetivos. Estos pesos permiten adaptar la función de *fitness* de acuerdo a las necesidades del gestor del proyecto. Por ejemplo, si reducir el coste de un proyecto es una prioridad, el correspondiente peso (w_{cost}) debe ser alto. No obstante, debemos tener en cuenta el orden de magnitud del coste y la duración del proyecto. Esto se puede hacer estableciendo los pesos al valor 1 y ejecutando el GA varias veces. Después, el peso asociado al coste se divide por la media del coste del proyecto y el peso asociado a la duración se divide por la media de la duración del proyecto. De esta forma, los términos ponderados asociados al coste del proyecto y a la duración se encontrarán en el mismo orden de magnitud. A partir de ese momento el gestor del proyecto puede ensayar diferentes valores para los pesos con el objetivo de adaptar las soluciones propuestas por el GA a sus prioridades.

El término de penalización p es la suma ponderada de los parámetros de la solución que la hacen no factible, es decir: el trabajo extra del proyecto (p_{over}), el número de tareas sin empleado asociado ($undt$), y el número de habilidades aún requeridas para realizar todas las tareas del proyecto ($reqsk$). Cada uno de estos parámetros es ponderado y sumado a la constante de penalización w_{penal}. Esta constante se incluye para separar el rango de valores de *fitness* de las soluciones factibles y no factibles. Los pesos asociados con las penalizaciones deben aumentarse hasta obtener un gran número de soluciones factibles. Los valores de los pesos usados en nuestros experimentos se encuentran en la Tabla 5.2. Se han obtenido explorando varias soluciones con el objetivo de mantener todos los términos de la suma en el mismo orden de magnitud.

Tabla 5.2: Pesos de la función de *fitness*.

Peso	Valor
w_{cost}	10^{-6}
w_{dur}	0.1
w_{penal}	100
w_{undt}	10
w_{reqsk}	10
w_{over}	0.1

5.2. Propuesta para la generación de casos de prueba

En esta sección describimos el generador de casos de prueba propuesto y el proceso completo de generación de casos de prueba. En primer lugar, debemos especificar el criterio de adecuación para formalizar el objetivo del generador. Como mencionamos en la Sección 3.2.1, usamos como criterio de adecuación la cobertura de condiciones que, como demostramos en dicha sección, implica a la cobertura de condiciones-decisiones [206] para los programas C.

Nuestro generador descompone el objetivo global (el criterio de cobertura de condiciones) en varios objetivos parciales consistiendo cada uno en hacer que un predicado atómico tome un determinado valor lógico [20]. Por ejemplo, de los fragmentos del grafo de control de flujo de la Figura 5.4 podemos extraer seis objetivos parciales: hacer el predicado atómico 1 cierto, hacerlo falso, etc. Después, cada objetivo parcial es tratado como un problema de optimización en el que la función a minimizar es una distancia, que detallamos en la siguiente sección, entre el caso de prueba actual y un caso de prueba que satisface el objetivo parcial. En la resolución del problema de minimización así planteado es donde incorporamos las técnicas metaheurísticas.

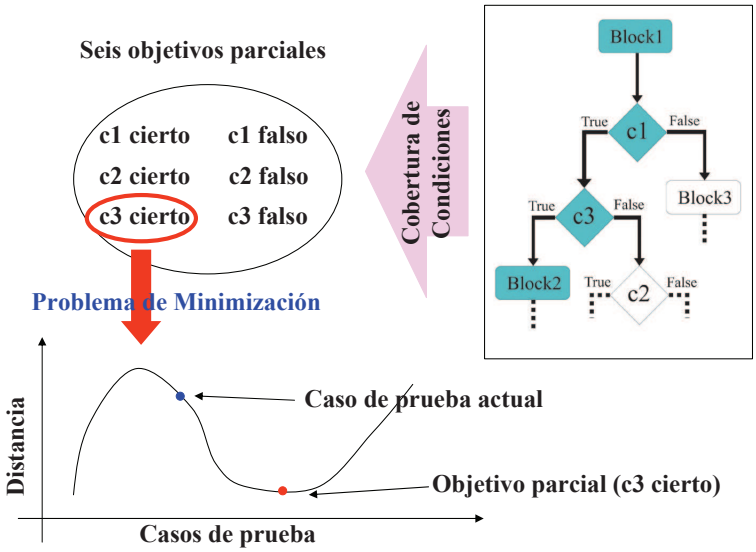


Figura 5.4: Identificamos seis objetivos parciales en este fragmento de diagrama de control de flujo.

5.2.1. Función de distancia

Siguiendo el planteamiento anterior, tenemos que resolver varios problemas de minimización: uno por cada predicado atómico y valor lógico. La función de distancia depende de la expresión del predicado particular asociado al objetivo parcial y de los valores de las variables del programa cuando el predicado es alcanzado. Por tanto, sólo puede ser calculada si el flujo del programa alcanza dicho predicado atómico, en otro caso la distancia toma en nuestra implementación el mayor valor posible para los números reales en una máquina. En la Tabla 5.3 mostramos la función de distancia para cada tipo de predicado atómico y cada valor lógico deseado. Las expresiones de las funciones de distancia han sido diseñadas para que sean diferenciables y su minimización implique la satisfacción del objetivo parcial. Algunas de ellas se pueden encontrar en trabajos previos [206].

Cuando un predicado atómico no se alcanza, su función de distancia asociada toma el valor real más alto que permite la representación IEEE 754 de 64 bits (es decir, $2^{1024} - 2^{971}$). La función de distancia toma también este valor cuando alguno de los argumentos de entrada está fuera del rango válido para ese argumento. Por ejemplo, si el primer argumento de una función o programa objeto debe ser un entero

Tabla 5.3: Funciones de distancia para los distintos tipos de predicados atómicos y valores lógicos. Las variables a y b son numéricas (enteras o reales).

Predicado atómico	Expresión para *cierto*	Expresión para *falso*
$a < b$	$a - b$	$b - a$
$a <= b$	$a - b$	$b - a$
$a == b$	$(b - a)^2$	$(1 + (b - a)^2)^{-1}$
a != b	$(1 + (b - a)^2)^{-1}$	$(b - a)^2$
a	$(1 + a^2)^{-1}$	a^2

comprendido entre 1 y 10, cada vez que se intente evaluar el programa usando como primer argumento un entero menor que 1 o mayor que 10, la función de distancia tomará el máximo valor de los reales.

5.2.2. Instrumentación del programa objeto

Para obtener información sobre el valor de distancia y los predicados atómicos alcanzados durante una ejecución, añadimos ciertas instrucciones al código fuente del programa objeto. La instrumentación debe realizarse con sumo cuidado para evitar cambiar el comportamiento del programa. En nuestro caso lo hacemos de forma automática con una aplicación propia que hemos desarrollado. Esta aplicación analiza el código fuente en C y genera un nuevo programa modificado listo para ejecutar el programa original y devolver, además, información sobre su funcionamiento. La modificación consiste en transformar cada predicado atómico en una expresión cuyo valor es el mismo que el del predicado atómico original, pero además tiene un efecto lateral inofensivo para el comportamiento del programa: informa sobre el valor lógico que toma, y el valor de distancia asociado al predicado atómico. Si `<cond>` es un predicado atómico en el programa original, la expresión usada en su lugar en el programa modificado es:

```
((<cond>)?
    (inform(<ncond>,1),(distance(<ncond>,<true_expr>,<false_expr>),1)):
    (inform(<ncond>,0),(distance(<ncond>,<true_expr>,<false_expr>),0)))
```

donde `<ncond>` es el número del predicado atómico en el programa, `<true_expr>` y `<false_expr>` son las expresiones de *fitness* para los valores *cierto* y *falso* del predicado atómico, `inform` es una función que informa al generador de casos de prueba sobre el predicado atómico alcanzado y su valor, y `distance` es una función que informa sobre el valor de distancia. Esta transformación no modifica el comportamiento funcional del programa a menos que el predicado atómico original tenga un efecto lateral[2]. Para evitar esto último, antes de utilizar la aplicación para instrumentar el programa, lo examinamos en busca de predicados atómicos con efectos laterales y lo modificamos manualmente. Nuestra actual versión de la aplicación de traducción puede transformar todos los predicados atómicos no vacíos que aparecen en las instrucciones `for`, `while`, `do-while`, e `if`. Por ahora, debemos transformar manualmente todas las sentencias `switch` y las expresiones `(a?b:c)` en una de las instrucciones soportadas.

El código modificado se compila y se enlaza con un fichero objeto (`instr.o`) que contiene las funciones que serán invocadas por las nuevas instrucciones añadidas para transmitir la información requerida al generador de casos de prueba. El resultado es un fichero ejecutable que será usado durante el proceso de generación de casos de prueba. El proceso completo de instrumentación se resume en la Figura 5.5.

[2]La razón de esto es que las expresiones de *fitness* se forman combinando los operandos de los predicados atómicos.

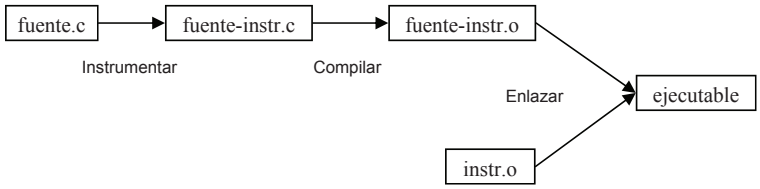


Figura 5.5: Proceso de instrumentación.

El fichero ejecutable resultante se debe lanzar con un programa especial: *launcher*. Este programa actúa como puente entre el programa ejecutable modificado y el generador de casos de prueba (véase la Figura 5.6). Escribe en la salida estándar toda la información sobre los predicados atómicos evaluados en el programa objeto. Durante la evaluación de un caso de prueba, cuando el generador ejecuta el programa objeto modificado, se elabora un informe (predicados atómicos alcanzados y los valores de distancia) que es transmitido al generador. Con esta información el generador mantiene una *tabla de cobertura* donde almacena por cada predicado atómico dos conjuntos de casos de prueba: los que hacen el predicado cierto y los que lo hacen falso. Esta tabla es una importante estructura de datos interna que se consulta durante la generación de casos de prueba para comprobar el grado de cumplimiento del criterio de adecuación. Diremos que un predicado ha sido *alcanzado* si al menos uno de los conjuntos es no vacío. Por otro lado, diremos que un predicado ha sido *cubierto* si los dos conjuntos son no vacíos.

5.2.3. El proceso de generación

Una vez que se han presentado las funciones de distancia y los detalles de la instrumentación podemos centrarnos en la generación de casos de prueba. El bucle principal del generador se muestra en la Figura 5.6. Al comienzo de la generación se crean algunos casos de prueba aleatorios (10 en nuestros experimentos) que alcanzan sólo algunos predicados. Después, comienza el bucle principal del generador donde, en primer lugar, se selecciona un objetivo parcial no cubierto, es decir, un predicado alcanzado pero no cubierto. La elección del objetivo parcial no es aleatoria, siempre se elige un objetivo parcial con una predicado atómico asociado previamente alcanzado. En concreto, se elige el objetivo parcial que, cumpliendo lo anterior, es el siguiente al objetivo recién abordado.

Por ejemplo, supongamos que los bloques sombreados de la Figura 5.7 representan las sentencias ejecutadas del programa para un caso de prueba particular. Entonces, sólo los valores de las funciones de distancia asociadas a los objetivos parciales `c1f` y `c3f` se pueden calcular. La evaluación de las funciones asociadas a `c2f` y `c2c` dará como resultado el mayor valor de los reales porque el flujo no ha alcanzado aún el predicado `c2`.

Cuando el objetivo ha sido elegido, se usa el algoritmo de optimización para buscar casos de prueba que hagan que el predicado tome el valor no cubierto aún. El algoritmo de optimización se inicializa con al menos un caso de prueba que permite alcanzar el predicado elegido. El algoritmo explora diferentes casos de prueba y usa los valores de distancia para guiar la búsqueda. Durante esta búsqueda se pueden encontrar casos de prueba que cubran otros objetivos parciales aún por satisfacer. Estos casos de prueba son usados también para actualizar la tabla de cobertura. De hecho, podemos establecer como condición de parada del algoritmo de optimización cubrir un objetivo parcial no cubierto aún (estudiaremos esta alternativa en el Capítulo 7). Tras ejecutar el algoritmo de optimización e independientemente del éxito de la búsqueda, el cuerpo del bucle principal se ejecuta de nuevo y se elige otro objetivo parcial (el siguiente).

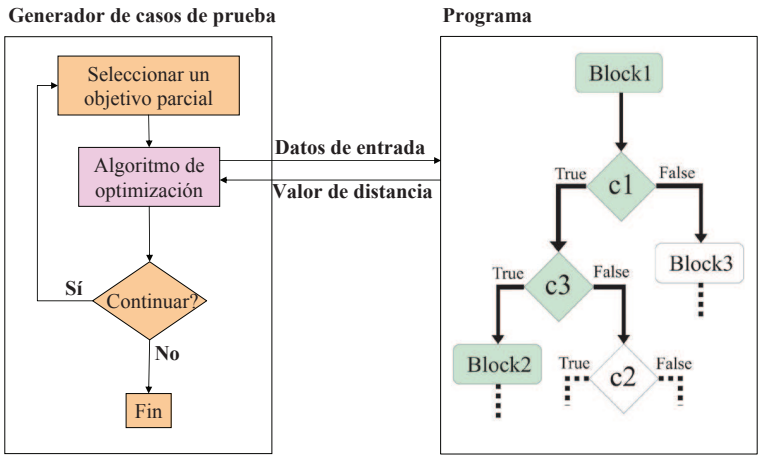


Figura 5.6: Generación de casos de prueba.

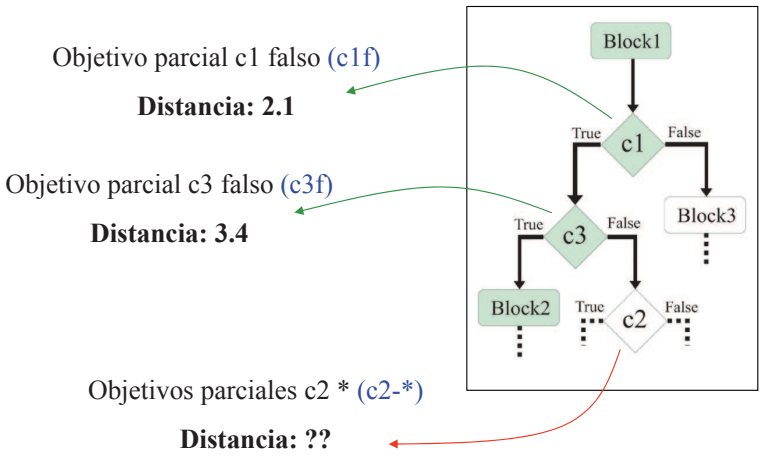


Figura 5.7: En esta situación sólo se pueden calcular las funciones de distancia de los objetivos parciales `c1f` y `c3f`.

Este esquema se repite hasta que se consigue cobertura total o se alcanza un número preestablecido de fracasos del algoritmo de optimización (10 en nuestros experimentos).

Cuando usamos algoritmos de optimización descentralizados, tales como un algoritmo evolutivo distribuido, tenemos varios subalgoritmos trabajando de forma independiente con algunas interconexiones poco frecuentes entre ellos. En este caso podemos asignar un objetivo parcial distinto a cada subalgoritmo. Si todos los objetivos parciales se resuelven aproximadamente al mismo tiempo, la búsqueda debería acelerarse. Esta alternativa será analizada también en el Capítulo 7.

```
while(1)
{
/* El predicado anterior
es siempre cierto */
⋮
}
```

```
char *p;
p = (char *)malloc (4);
if (!p)
{
        fprintf("Error");
        exit(0);
}
```

Figura 5.8: Dos fragmentos de código que impiden a un programa alcanzar 100 % de cobertura de condiciones. El de la izquierda produce una pérdida dependiente del código y el de la derecha una pérdida dependiente del entorno.

5.2.4. Medidas de cobertura

Para terminar con la descripción del generador debemos discutir las medidas de cobertura usadas para presentar los resultados del generador. La medida más simple es el cociente entre los objetivos parciales cubiertos y el número total de objetivos parciales (véase la Sección 3.2.1 para más detalles sobre medidas de cobertura concretas). Este valor expresado en porcentaje se conoce como *porcentaje de cobertura* (porcentaje de cobertura de condiciones, en nuestro caso). Aunque ésta es la forma más simple de medir la eficacia del generador, no es la más apropiada. La razón es que existen programas para los que es imposible conseguir una cobertura total, ya que tienen objetivos parciales inalcanzables. En este caso se produce una pérdida de cobertura que es independiente de la técnica usada para la generación de casos de prueba. Por ejemplo, un bucle infinito tiene un predicado que siempre es verdadero y nunca falso (Figura 5.8 izquierda). Otro ejemplo es el predicado `(sign(x)>2)`, donde la función `sign` puede devolver sólo tres valores: -1, 0, +1. En este caso hablamos de *pérdida de cobertura dependiente del código*. No obstante, hay otro factor que puede producir una pérdida de cobertura inevitable: el entorno en el que el programa se ejecuta. Por ejemplo, si un programa pide una pequeña cantidad de memoria dinámica y después comprueba si la asignación tuvo éxito, lo más probable es que lo tenga en todas las ejecuciones del programa y el predicado que comprueba si hay error sea siempre falso. En este caso decimos que hay una *pérdida de cobertura dependiente del entorno* (Figura 5.8 derecha). Cuando se da una de estas situaciones ningún generador de casos de prueba es capaz de conseguir cobertura total y puede parecer ineficaz cuando, en realidad, no es así. Por ejemplo, podemos obtener un porcentaje de cobertura bajo en un programa, pero esta cobertura podría ser la mayor que se puede conseguir.

Nosotros buscamos una medida de cobertura que tenga en cuenta las pérdidas en la medida de lo posible: una medida que alcance su valor máximo cuando es imposible cubrir más objetivos parciales. Por ello, hemos introducido otra medida que denominamos *cobertura corregida* y que se calcula como el cociente entre el número de objetivos parciales cubiertos y los alcanzables. En esta medida los objetivos parciales inalcanzables no se tienen en cuenta, sin ninguna pérdida de información o desventaja para la generación de casos de prueba. La cobertura corregida es útil para comparar el rendimiento del generador de casos de prueba en diferentes programas. De este modo, podemos ordenar los programas de acuerdo a su dificultad para un generador dado. Si usáramos la medida simple de cobertura podríamos clasificar a un programa como difícil cuando, en realidad, tiene muchos objetivos parciales inalcanzables pero los objetivos alcanzables son fáciles de cubrir. No obstante, el cálculo de la cobertura corregida requiere

conocer los objetivos parciales inalcanzables. En pequeños programas estos objetivos parciales pueden ser fácilmente determinados, pero en grandes programas puede ser una tarea muy difícil (podría ser un problema NP-duro en sí mismo). En estos casos la cobertura corregida no es práctica. En los experimentos de esta tesis, decidimos por observación humana si un objetivo parcial es alcanzable o no y los objetivos parciales inalcanzables se indican al generador de casos de prueba mediante ficheros de configuración. Usamos la cobertura corregida en los experimentos para evitar la pérdida dependiente del código. La pérdida dependiente del entorno es más difícil de evitar y la cobertura corregida no la considera.

Tras la discusión previa sobre las medidas de cobertura y la introducción de la cobertura corregida, necesitamos modificar el modo en que se realiza la selección del objetivo parcial al principio del bucle principal del generador de casos de prueba para contar correctamente el número de evaluaciones requeridas para alcanzar la cobertura registrada. La modificación consiste en no tener en cuenta los objetivos parciales no alcanzables en la selección. Así pues, los predicados asociados a objetivos parciales inalcanzables se marcan y el generador no intenta cubrirlos. Lo más que se conseguirá en estos predicados será alcanzarlos.

5.2.5. Detalles de las metaheurísticas empleadas

La aportación original en este problema ha sido, por un lado, la aplicación de técnicas metaheurísticas nunca antes usadas para la generación automática de casos de prueba y, por otro, un estudio detallado del rendimiento de algoritmos evolutivos distribuidos. En el primer caso, las nuevas metaheurísticas aplicadas son ES y PSO. En ambas las soluciones del problema se representan mediante vectores de números reales, lo cual permite explorar completamente el espacio de búsqueda del problema. Esto contrasta con otros trabajos previos en los que los valores de los argumentos de entrada del programa se restringen a valores enteros y se acotan, reduciendo el espacio de búsqueda [70, 244, 261]. Puesto que GA ha sido la técnica más popular dentro de la generación evolutiva de casos de prueba, incluimos en nuestros estudios un GA que, a diferencia de propuestas previas [261], posee representación real de los argumentos de entrada del programa. De esta forma, podemos comparar la efectividad de ES y PSO frente a la de GA en igualdad de condiciones y no es posible achacar las diferencias encontradas a la representación de los casos de prueba. En cuanto al estudio de los algoritmos distribuidos, los algoritmos escogidos fueron ES y GA junto con sus variantes distribuidas. A continuación describiremos algunos detalles particulares de las técnicas aplicadas a la generación automática de casos de prueba.

Detalles de ES

Los programas utilizados en los experimentos admiten argumentos reales y enteros únicamente. En la ES, cada argumento se representa con un valor real. En el caso de que el argumento correspondiente sea un número entero, se redondea el valor real asociado en el vector solución.

Detalles de PSO

La versión de PSO utilizada representa las soluciones al problema mediante vectores de números reales. Al igual que en la ES, cuando el programa posee un argumento entero, se redondea el valor real asociado al argumento antes de ejecutar el programa.

El algoritmo PSO tiende a converger rápidamente, especialmente en el problema que nos ocupa, cuya función de *fitness* contiene muchas zonas planas (*plateaus*). Por este motivo, decidimos incorporar a la técnica un operador que añade un valor aleatorio a las velocidades de la mitad de las partículas (escogidas aleatoriamente) si la mejor solución del cúmulo no mejora tras una iteración. La magnitud

de la perturbación añadida va creciendo exponencialmente (se multiplica por 10) conforme aumenta el número de iteraciones consecutivas sin mejora.

Detalles de GA

El algoritmo genético usado para este problema usa como cromosoma un vector de valores numéricos, que pueden ser reales o enteros. Los operadores de recombinación más populares para los GA (SPX, DPX y UX) se pueden aplicar a esta representación sin más que cambiar los bits por componentes del vector.

El operador de mutación utilizado para esta representación añade a cada componente del vector un valor aleatoriamente generado siguiendo una distribución normal con media cero. La desviación estándar de esta normal y la probabilidad de aplicar la mutación a un individuo son parámetros del operador.

5.2.6. Representación y función de *fitness*

La función de *fitness* usada en la búsqueda no es exactamente la función de distancia. Queremos evitar valores negativos de *fitness* para poder aplicar operadores de selección que dependen directamente del valor de la función de *fitness*, tales como la selección por ruleta. Por esta razón, transformamos el valor de distancia usando una función arco tangente que lleva el conjunto de los números reales a un intervalo acotado. La función de *fitness* resultante es:

$$f(\mathbf{x}) = \pi/2 - \arctan(\mathit{distance}(\mathbf{x})) + 0.1 \ . \tag{5.4}$$

En la expresión anterior, multiplicamos el `arctan` por -1 debido a que los algoritmos que aplicamos a este problema están diseñados para maximizar la función de *fitness*. Además, necesitamos añadir el valor $\pi/2$ a la expresión para obtener siempre un valor positivo. Finalmente, el valor 0.1 se usa para no obtener valores negativos cuando existe alguna pérdida de precisión en el cálculo.

5.3. Propuesta para la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes

Como mencionamos en la Sección 3.3.3, el problema de buscar violaciones de propiedades de seguridad en sistemas concurrentes puede ser transformado en un problema de búsqueda de un nodo objetivo en un grafo. Para la búsqueda de este nodo podemos hacer uso de información heurística tal y como comentamos en la Sección 3.3.4. De entre las técnicas metaheurísticas, la que utiliza un grafo como escenario de búsqueda es la optimización basada en colonias de hormigas. Por este motivo, nos planteamos aplicar ACO a la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. Tras una corta reflexión surge, no obstante, un inconveniente: la explosión de estados afecta también dramáticamente a los modelos de ACO existentes. La solución a este problema ha sido el desarrollo de un nuevo modelo algorítmico basado en las colonias de hormigas: ACOhg. En lo que sigue justificaremos y detallaremos esta nueva técnica.

5.3.1. Justificación de ACOhg

Los modelos de ACO que podemos encontrar en la literatura se pueden aplicar (y se han aplicado) a problemas con un número de nodos n de varios millares. En estos problemas, el grafo de construcción

tiene un número de arcos del orden de n^2, es decir, varios millones de arcos, y la matriz de feromonas requiere para su almacenamiento varios megabytes de memoria. Sin embargo, el esquema descrito no es adecuado en problemas en los que el grafo de construcción tiene como mínimo del orden de 10^6 nodos (y 10^{12} arcos). Tampoco lo es cuando la cantidad de nodos no se conoce de antemano y los nodos y arcos del grafo de construcción se generan conforme avanza la búsqueda. Nosotros tratamos de resolver aquí este tipo de problemas. En efecto, el número de estados de un sistema concurrente es normalmente muy alto incluso en pequeños modelos. Por ejemplo, el número de estados del modelo del problema de los filósofos presentado en la Sección 8.1 es 3^n, donde n es el número de filósofos. Es decir, el número de estados crece de forma exponencial con respecto al número de filósofos (tamaño del modelo).

Discutamos las cuestiones que impiden a los modelos existentes resolver este tipo de problemas. En primer lugar, en la fase de construcción, las hormigas de un ACO tradicional caminan hasta que construyen una solución completa. En la exploración de un autómata de Büchi una solución completa es un camino que termine en un estado de aceptación. Si permitimos a las hormigas caminar por el grafo sin repetir nodo hasta que encuentren un nodo objetivo, podrían llegar a un nodo sin sucesores no visitados (un nodo sin salida para las hormigas). Incluso si encuentran un nodo objetivo, puede ser necesario mucho tiempo y memoria para construir una solución candidata, ya que los nodos objetivo pueden estar muy lejos del nodo inicial. A esto podemos añadir que ni siquiera tenemos asegurado que exista un estado de aceptación en el grafo, porque el modelo puede cumplir la propiedad que se intenta violar. Por tanto, no es viable, en general, trabajar con soluciones completas como hacen los actuales modelos. Es necesario poder trabajar con soluciones parciales. Queremos destacar que no estamos discutiendo aquí un detalle de implementación, sino que tratamos con cuestiones que deben ser tenidas en cuenta en el diseño del nuevo algoritmo para poder trabajar con problemas que tienen un grafo de construcción de gran dimensión.

Por otro lado, algunos modelos de ACO asumen que el número de nodos del grafo de construcción se conoce de antemano y la cantidad inicial de feromona de cada arco depende de este número de nodos. Este tipo de inicialización no es posible cuando trabajamos con grafos de tamaño desconocido. También debemos tener cuidado con la implementación de los rastros de feromona. En los modelos previos, la matriz de feromonas se almacena en un array, pero esto requiere conocer el número de nodos. En nuestro caso, incluso si conociéramos dicho número, no podríamos almacenar la matriz de feromonas en arrays debido a la gran cantidad de memoria requerida (normalmente no disponible).

Para solventar las dificultades que surgen al trabajar con grafos de gran dimensión, proponemos un nuevo modelo de ACO, llamado ACOhg (*ACO for huge graphs*), que es capaz de abordar problemas con un grafo de construcción subyacente de tamaño desconocido que se construye conforme avanza la búsqueda. Las principales cuestiones que tenemos que resolver están relacionadas con la longitud de los caminos de las hormigas, la función de *fitness*, y la cantidad de memoria usada para almacenar los rastros de feromona. Abordaremos estas cuestiones a continuación.

5.3.2. Longitud de los caminos de las hormigas

Una primera estrategia fundamental para evitar la, generalmente inviable, construcción de soluciones completas consiste en limitar la longitud de los caminos trazados por las hormigas. Es decir, cuando el camino construido por una hormiga alcanza cierta longitud límite λ_{ant} ésta se detiene. De esta forma, la fase de construcción se puede realizar en un tiempo acotado y con una memoria acotada. La limitación de la longitud de las hormigas implica que los caminos trazados por ellas no representan siempre soluciones completas, sino que, en general, serán soluciones parciales. Necesitamos, por tanto, una función de *fitness* que pueda evaluar estas soluciones parciales.

La propuesta anterior resuelve el problema de las "hormigas errantes" pero introduce un nuevo problema: tenemos un nuevo parámetro para el algoritmo (λ_{ant}). No es fácil establecer *a priori* cuál es el mejor valor para λ_{ant}. Si se elige un valor menor que la profundidad[3] de todos los nodos objetivo, el algoritmo no encuentra ninguna solución. Por tanto, debemos escoger un valor mayor que la profundidad de algún nodo objetivo (de hecho, veremos en el capítulo de experimentos que este valor debe ser, en general, varias veces mayor que la profundidad del nodo objetivo para obtener una buena tasa de éxito). Esto no es difícil cuando conocemos la profundidad de un nodo objetivo, pero normalmente se da la situación contraria, en cuyo caso podemos usar dos alternativas que dan lugar a dos variantes del modelo. En la Figura 5.9 ilustramos gráficamente la forma en que trabajan estas dos alternativas presentadas.

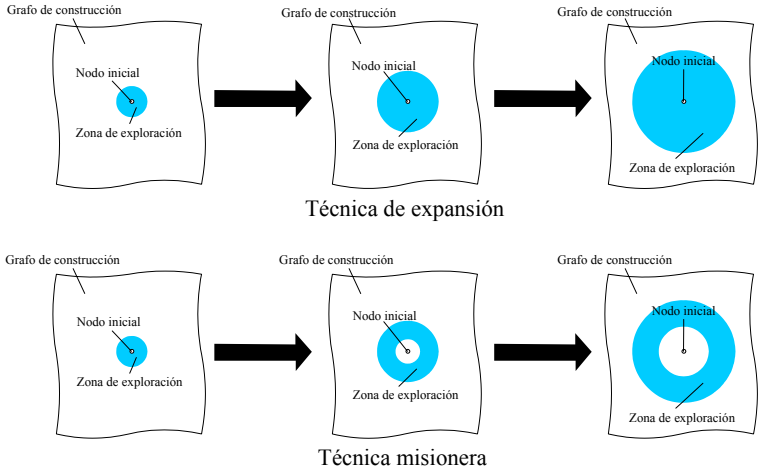


Figura 5.9: Dos alternativas para alcanzar nodos objetivos a profundidad desconocida: técnica de expansión (arriba) y técnica misionera (abajo). Mostramos instantáneas de diferentes momentos de la búsqueda.

La primera consiste en incrementar λ_{ant} durante la búsqueda si no se encuentra ningún nodo objetivo. Al principio se asigna un valor bajo a λ_{ant} y se incrementa en una cantidad dada δ_l cada cierto número de pasos σ_i del algoritmo. De esta forma, en algún momento la longitud máxima de los caminos será suficientemente larga como para alcanzar algún nodo objetivo. Esta estrategia, llamada *técnica de expansión*, es similar a la que usa IDA* y puede ser útil cuando la profundidad de los nodos objetivos no es muy alta. En caso contrario, la longitud de los caminos de las hormigas crecerá mucho y lo mismo sucederá con el tiempo y la memoria requeridos para construir estos caminos, convirtiéndose el ACOhg en un ACO tradicional de forma incremental.

La segunda alternativa consiste en comenzar la construcción del camino de las hormigas en nodos diferentes durante la búsqueda. Al principio las hormigas se colocan en el nodo inicial del grafo y el algoritmo se ejecuta durante un número dado de pasos σ_s (llamado *etapa*). Si no se encuentra ningún nodo objetivo, los últimos nodos de los caminos construidos por las hormigas se usan como nodos iniciales para las siguientes hormigas. En los siguientes pasos del algoritmo (segunda etapa), las nuevas hormigas comienzan su exploración al final de los caminos de las hormigas de la primera etapa, intentando ir más

[3]Definimos la *profundidad* de un nodo en el grafo de construcción como la distancia del camino más corto que llega a él partiendo del nodo inicial.

allá en el grafo. Esta estrategia se denomina *técnica misionera*. La longitud de estos caminos, λ_{ant}, se mantiene constante durante la búsqueda y los rastros de feromona pueden olvidarse de una etapa a otra para mantener casi constante la cantidad de recursos computacionales (memoria y CPU) en todas las etapas. La elección del nodo inicial para las hormigas se realiza en dos fases. Primero, necesitamos elegir los caminos de la etapa anterior cuyos últimos nodos se usan como nodos de comienzo en la nueva. Para esto, guardamos los mejores caminos (de acuerdo a su *fitness*) de la etapa anterior. Llamamos s al número de caminos guardados. Una vez que tenemos el conjunto de nodos de comienzo, tenemos que asignar las hormigas a esos nodos. Para cada hormiga seleccionamos su nodo inicial usando selección por ruleta; es decir, la probabilidad de elegir un camino es proporcional al valor de *fitness* de la solución asociada con él. El número de caminos guardados entre dos etapas, s, es un parámetro del algoritmo establecido por el usuario. Debe cumplirse $1 \leq s \leq \sigma_s \cdot colsize$[4], es decir, debe haber al menos un camino guardado (las hormigas de las siguientes etapas necesitan al menos un nodo de partida) y la cota máxima es el número máximo de caminos que el algoritmo es capaz de construir en σ_s pasos: $\sigma_s \cdot colsize$.

5.3.3. Función de *fitness*

El objetivo de ACOhg es encontrar un camino de bajo coste entre un nodo inicial y un nodo objetivo. Para el problema de búsqueda de violaciones de propiedades de seguridad, el coste de una solución es su longitud, pero, en general, coste y longitud (número de componentes) de una solución pueden ser diferentes, y por este motivo es más adecuado hablar de coste. Si consideramos un problema de minimización, la función de *fitness* de una solución completa puede ser el coste de la solución. No obstante, como dijimos en la Sección 5.3.1, la función de *fitness* debe ser capaz de evaluar soluciones parciales. En este caso, el coste de una solución parcial no es un valor adecuado de *fitness*, porque una solución parcial tendrá en general menor coste que una solución completa y puede ser considerada mejor que ésta. Esto significa que las soluciones parciales de bajo coste son premiadas y la función de *fitness* no representará adecuadamente la calidad de las soluciones. Para evitar este problema penalizamos las soluciones parciales añadiendo una cantidad fija p_p al coste de tales soluciones.

En nuestro caso, para resolver el problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes, calculamos la función de *fitness* del siguiente modo. En primer lugar, dada una solución (parcial) calculamos una cota inferior de la longitud de una solución completa que es una extensión de la solución. Esta cota inferior se calcula es la suma de la longitud de la solución parcial y el valor heurístico del último estado del camino. En segundo lugar, añadimos un término de penalización cuando la solución no es completa. Así conseguimos, gracias a la penalización, que las soluciones parciales tengan un valor de *fitness* más alto que las completas. Además, las soluciones parciales más prometedoras tendrán un valor de *fitness* menor que el resto.

En ocasiones, la formación de ciclos en el grafo de construcción no es deseable. Esto ocurre, en particular, en nuestro problema. Un camino que posea un ciclo puede ser sustituido por otro sin ciclos con menor longitud que el primero. Es, por tanto, deseable para algunos problemas evitar la posibilidad de formar ciclos en el grafo de construcción. A continuación describimos la forma en que evitamos tales ciclos cuando aplicamos ACOhg al problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes.

Durante la fase de construcción de una hormiga, cuando ésta recorre su camino, puede parar debido a tres razones: se alcanza la longitud máxima λ_{ant}, el último nodo del camino de la hormiga es un nodo objetivo, o todos los nodos sucesores están en el camino de la hormiga (nodos visitados). Esta última

[4]Denotamos con *colsize* el número de hormigas de la colonia.

condición, que es la que evita la construcción de caminos con ciclos, tiene un efecto lateral poco deseable: premia los caminos que tienden a formar ciclos. En efecto, la detención prematura de la hormiga durante la fase de construcción produce una solución parcial más corta de lo normal. Puesto que tratamos de resolver un problema de minimización de la longitud de los caminos, estas soluciones parciales que dan lugar a ciclos serán considerados de mayor calidad, por ser más cortos, que los caminos que no dan lugar a la formación de ciclos. Para evitar esta situación, penalizamos aquellas soluciones parciales cuya longitud es menor que λ_{ant}.

La expresión total para el término de penalización es

$$p = p_p + p_c \frac{\lambda_{ant} - l}{\lambda_{ant} - 1} \ , \tag{5.5}$$

donde p_p es la penalización debida a la parcialidad de las soluciones, p_c es una constante de penalización asociada a la formación de ciclos, y l es la longitud del camino. El segundo término en (5.5) hace que la penalización sea mayor en caminos que contengan ciclos más cortos. La idea intuitiva detrás de esto es que los caminos conteniendo ciclos más largos están más cerca de convertirse en caminos sin ciclos. Por esta razón, añadimos a p_p la penalización máxima de ciclo (p_c) cuando la longitud del camino es el mínimo ($l = 1$) y no hay penalización de ciclo cuando no hay ciclo ($l = \lambda_{ant}$). En conclusión, la función de *fitness* final que usamos en nuestros experimentos para evaluar soluciones parciales en el problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes es:

$$f(x) = l + H(j) + p_p + p_c \frac{\lambda_{ant} - l}{\lambda_{ant} - 1} \ , \tag{5.6}$$

donde j es el último estado del camino y $H(j)$ es el valor heurístico en el estado j. El valor de *fitness* para una solución completa es exactamente la longitud del camino de la hormiga l.

5.3.4. Rastros de feromona

En ACOhg la matriz de feromonas es sustituida por una matriz dispersa donde sólo se almacenan los valores de feromona de los arcos por los que pasan las hormigas. Esto es así tanto para la técnica de expansión como para la técnica misionera. Conforme la búsqueda progresa, la memoria requerida para dicha matriz puede crecer hasta cantidades inadmisibles. Podemos evitar esto eliminando de la matriz de feromona los rastros con poca influencia en la fase de construcción. Esto se hace en la técnica misionera cuando comienza una nueva etapa. En cada etapa, la región del grafo explorada por las hormigas es diferente de las exploradas en etapas previas[5]. Esto significa que los rastros de feromona usados en las etapas anteriores no son útiles en la actual y pueden ser descartados sin una influencia negativa en los resultados (como se puede comprobar en la Sección A.2.3).

5.3.5. Grafo de construcción

Una última cuestión que hay que tener en cuenta en el nuevo modelo es la generación del grafo de construcción. A diferencia de otros modelos de ACO, en el nuestro el grafo se genera conforme se realiza la exploración. Esto significa que conforme avance la búsqueda será necesaria más memoria para almacenar los nodos del grafo. Es necesario cuidar la implementación para ahorrar memoria. Por ejemplo, si el

[5]Es posible encontrar algún solapamiento entre regiones exploradas en diferentes etapas, pero este solapamiento debe ser muy pequeño si la función de *fitness* y el heurístico están bien definidos.

tamaño de los nodos en memoria supera el tamaño de un puntero, se puede mantener una sola copia del nodo y hacer referencia a ella allá donde se necesite. En este caso hay que evitar duplicados de un nodo en memoria. No obstante, si el tamaño de los nodos es igual o inferior al de un puntero, el mecanismo anterior no es útil y convierte la implementación en ineficiente. Por otro lado, si se eliminan los rastros de feromona tras cada etapa en la técnica misionera, los nodos extremos de los arcos suprimidos podrían ser también eliminados si no son referenciados por ningún otro arco u hormiga.

Las alternativas arriba mencionadas son mecanismos de ahorro de memoria que se han tenido en cuenta en la implementación del algoritmo para maximimizar la región del grafo que puede almacenarse en memoria, ya que es en esta región donde el algoritmo puede trabajar con mayor eficacia para maximizar la calidad de las soluciones parciales.

5.3.6. Pseudocódigo de ACOhg

En esta sección presentamos el pseudocódigo del algoritmo ACOhg. Antes de explicarlo debemos introducir algo de notación. Tal y como hicimos en la Sección 4.4.3, denotaremos el grafo de construcción mediante $G = (C, L)$, donde C es el conjunto de nodos del grafo y $L \in C \times C$ es el conjunto de arcos. El nodo inicial del grafo es $q \in C$ y el conjunto de nodos finales (nodos objetivo) es $F \subseteq C$. Denotaremos con $L(s)$ el conjunto de nodos sucesores de s. Las hormigas artificiales (ant^i) son caminos finitos en el grafo, es decir, son secuencias de la forma $\text{ant}^i = s_1 s_2 \dots s_n$ donde $s_j \in C$ para $j = 1, 2, \dots, n$. Denotaremos con ant^i_j el estado j-ésimo del camino de la hormiga ant^i y con $|\text{ant}^i|$ la longitud del camino asociado a dicha hormiga. Por último, usaremos ant^i_* para referirnos al último estado del camino de la hormiga ant^i, es decir, $\text{ant}^i_* = \text{ant}^i_{|ant^i|}$. El pseudocódigo de ACOhg se encuentra en el Algoritmo 5.

La variable `init` contiene el conjunto de caminos cuyos últimos nodos se usarán como nodos iniciales de las hormigas. Por otro lado, `next_init` contiene el conjunto de los mejores caminos encontrados en una etapa. Estas dos variables sólo se actualizan cuando se usa la técnica misionera, ya que en la técnica de expansión todas las hormigas construyen su camino partiendo del nodo inicial del grafo de construcción. Al comienzo, `init` sólo contiene el nodo inicial de grafo de construcción, q (línea 1 en el Algoritmo 5). La variable `stage` no tiene ningún papel relevante en el pseudocódigo; la incluimos para indicar claramente cuándo se produce el cambio de etapa en la técnica misionera.

Tras la inicialización de las variables (líneas 1-6), el algoritmo entra en un bucle del que sólo sale cuando se cumpla una determinada condición de parada (líneas 7-38). Dentro de dicho bucle, las hormigas de la colonia construyen soluciones parciales (caminos en el grafo) usando la misma regla que un ACO tradicional para escoger el siguiente nodo del camino (líneas 8-16). La construcción de un camino se interrumpe cuando la hormiga ha alcanzado la longitud máxima permitida (λ_{ant}), ha alcanzado un nodo ya visitado durante la fase de construcción actual, o ha llegado a un nodo objetivo (línea 11). Si se usa la técnica de expansión, el valor de λ_{ant} se incrementa en δ_l cada σ_i pasos (líneas 20-22). En el caso de que se use la técnica misionera, los últimos nodos de los mejores caminos de la etapa actual se usarán como nodos iniciales para que las hormigas construyan sus caminos en la próxima etapa (líneas 24-30). Además, los rastros de feromona son inicializados de nuevo, con el objetivo de reducir el consumo de memoria (línea 28). El resto de líneas de código del Algoritmo 5 contienen instrucciones ya presentes en los ACO tradicionales (véase el Algoritmo 3). En este caso estamos considerando un problema de minimización de la función objetivo, como puede apreciarse en el signo “menor que” de la línea 33.

Algoritmo 5 Ant colony optimization for huge graphs (ACOhg)

```
1: init = {q};
2: next_init = ∅;
3: step = 0;
4: stage = 0;
5: inicializarFeromonas(τ); {Inicializa los rastros de feromona}
6: ant^bs = generarSolución(τ, η); {Se inicializa ant^bs con una solución inicial aleatoria}
7: repeat
8:    for i=1 to colsize do
9:       ant^i = ∅;
10:      ant^i_1 = elegirNodoInicial(init);
11:      while |ant^i| ≤ λ_ant ∧ L(ant^i_*)−ant^i ≠ ∅∧ ant^i_* ∉ F do
12:         nodo = elegirNodoSucesor(L(ant^i_*), τ, η);
13:         ant^i = ant^i + nodo; {Añade un nodo al camino actual}
14:         actualizarFeromonasLocal(τ, ant^i); {Actualización local de los rastros de feromona}
15:      end while
16:   end for
17:   evaporarFeromonas(τ); {Evaporación de los rastros de feromona}
18:   actualizarFeromonasGlobal(τ,ant,ant^bs); {Actualización global de los rastros de feromona}
19:   if expansión then
20:      if step ≡ 0 mod σ_i then
21:         λ_ant = λ_ant + δ_l; {Incrementar λ_ant}
22:      end if
23:   else if misionera then
24:      next_init = elegirMejoresCaminos(init, next_init, ant);
25:      if step ≡ 0 mod σ_s then
26:         init = next_init; {Cambio de etapa}
27:         next_init = ∅;
28:         inicializarFeromonas(τ);
29:         stage = stage + 1;
30:      end if
31:   end if
32:   for i=1 to colsize do
33:      if f(ant^i) < f(ant^bs) then
34:         ant^bs = ant^i; {Actualizar la mejor solución}
35:      end if
36:   end for
37:   step = step + 1;
38: until condiciónParada()
39: return la mejor solución encontrada
```

5.3.7. Integración de ACOhg y HSF-SPIN

La implementación de nuestro modelo se ha realizado dentro de la biblioteca MALLBA [7], usando como base una implementación previa y general de los modelos ACO existentes realizada por Guillermo Ordóñez [8]. La incorporación del modelo dentro de MALLBA tiene la ventaja, entre otras, de que puede paralelizarse sin mucho esfuerzo y de forma transparente a los usuarios finales.

Por otro lado, existe un model checker desarrollado por Stefan Edelkamp y Alberto Lluch-Lafuente llamado HSF-SPIN que integra una biblioteca de algoritmos de exploración de grafos (HSF) y SPIN permitiendo la aplicación de métodos de búsqueda heurística a la verificación de sistemas modelados en Promela [92]. Nosotros hemos incorporado MALLBA dentro de HSF-SPIN para poder usar nuestra implementación de ACOhg. De esta forma, podemos despreocuparnos de los detalles relacionados con la representación de modelos (interpretación del lenguaje Promela, fórmulas LTL, etc.) y disponemos además de una gran cantidad de funciones heurísticas listas para usar (ya implementadas en HSF-SPIN).

Para la implementación de algoritmos generales de búsqueda, HSF-SPIN proporciona una función denominada `expand` que toma como argumento un estado del sistema concurrente y devuelve una lista con todos los posibles próximos estados. Cuando se da la circunstancia de que sólo existe un próximo estado y a dicho estado le sigue, a su vez, un único estado, la función `expand` avanza dos pasos y devuelve el último estado. No obstante, el estado intermedio es almacenado como parte del camino y HSF-SPIN lo contabiliza como un estado más. Nuestra implementación de ACOhg en MALLBA, sin embargo, no cuenta el estado intermedio y, durante la fase de construcción, una hormiga que realice dicha transición avanza directamente al segundo estado. Esta forma de proceder implica que tenemos dos medidas para la longitud de los caminos: el número de estados tal y como los cuenta HSF-SPIN y el número de nodos tal y como los cuenta ACOhg (la longitud del camino de la hormiga). En general, estas dos medidas difieren (la segunda es menor) y debemos elegir una de ellas a la hora de mostrar los resultados. En esta tesis hemos decidido usar la longitud de los caminos de las hormigas, es decir, no tenemos en cuenta los estados intermedios. Esta medida es más adecuada para relacionar los parámetros de ACOhg (λ_{ant}, σ_s, σ_i y δ_l) con los resultados obtenidos. Pero esto también significa que los valores de longitud mostrados en las tablas y gráficas de resultados no serán los que se obtendrían usando una versión estándar de HSF-SPIN.

5.4. Conclusiones

En este capítulo hemos presentado las propuestas metodológicas para cada uno de los problemas abordados. En todos los casos hemos usado metaheurísticas basadas en población; en concreto, algoritmos genéticos, estrategias evolutivas, optimización basada en cúmulos de partículas y optimización basada en colonias de hormigas. En todos los casos hemos tenido que adaptar los algoritmos a las particularidades del problema a resolver. En el caso de la planificación de proyectos, se ha utilizado un operador de cruce inusual en el dominio de las metaheurísticas: el operador de recombinación de un punto para tablas. En la generación de casos de prueba, el algoritmo de optimización forma parte de un sistema mayor que orquesta todo el proceso de generación de casos de prueba. Fue necesario emplear representación real en el algoritmo genético y, debido a ello, un operador de mutación para dicha codificación. También hemos incorporado al algoritmo PSO un mecanismo de perturbación para evitar la convergencia prematura. Por último, para la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes, hemos desarrollado un nuevo modelo de ACO para poder abordar el problema, ya que los ACO existentes no pueden aplicarse debido a la dimensión del grafo de construcción del problema.

Capítulo 6

Aplicación de GA a la planificación de proyectos software

En este capítulo abordamos el problema de planificación de proyectos software usando algoritmos genéticos (GA). Gracias a nuestro generador de instancias podemos realizar estudios estructurados sobre la influencia que los atributos más importantes del problema tienen en las soluciones. Para el estudio experimental generamos un total de 48 instancias diferentes con el generador de instancias (Sección 5.1.1) y las resolvemos con el algoritmo genético descrito en la Sección 5.1.2. Hemos clasificado dichas instancias en cinco grupos dependiendo de sus características. En los tres primeros cambiamos sólo un parámetro del problema. Con estos estudios queremos analizar la influencia aislada de estos parámetros en los resultados. En los últimos dos grupos cambiamos varios parámetros a la vez. De este modo estudiamos si los resultados cambian del modo sugerido por los estudios de los primeros tres grupos o, por el contrario, existen dependencias entre los parámetros.

El capítulo se organiza como sigue. La siguiente sección presenta los parámetros del algoritmo genético utilizado en los experimentos. Las Secciones 6.2 a 6.6 analizan los resultados obtenidos tras la aplicación del GA a las 48 instancias del problema ordenadas por grupos. La Sección 6.7 muestra más detalles sobre la dinámica del algoritmo genético cuando se resuelven las instancias. Por último, la Sección 6.8 presenta las conclusiones finales sobre los resultados obtenidos.

6.1. Configuración del algoritmo

Para resolver las instancias usamos un GA con una población de 64 individuos, selección por torneo binario, recombinación de un punto para tablas, mutación por inversión de bits, y reemplazo elitista del peor (algoritmo genético de estado estacionario). El criterio de parada es alcanzar 5000 generaciones del algoritmo (un total de 10064 evaluaciones). Hemos elegido el operador de recombinación de un punto para tablas por ser la estructura de las soluciones una matriz. Debido a la representación binaria de las soluciones hemos usado el operador de mutación por inversión de bits. La elección de un GA estacionario frente a un GA generacional se debe a que existen trabajos que muestran una mayor eficacia del primero [276]. Realizamos 100 ejecuciones independientes para cada instancia y presentamos medias y desviaciones típicas (en subíndice) en las tablas de resultados. Las máquinas usadas para los experimentos

son Pentium 4 a 2.8 GHz con 512 MB de RAM y sistema operativo Linux (versión del *kernel* 2.4.19-4GB). En las tablas de resultados no mostramos el tiempo de ejecución del GA porque su análisis no forma parte del objetivo de estos experimentos. Sin embargo, podemos indicar que dicho tiempo de ejecución se encuentra en todos los casos comprendido entre unos 2.5 segundos en las instancias más pequeñas y 24 segundos en las más grandes aquí resueltas. Este tiempo de ejecución es razonablemente bajo para que el uso de GA en este problema sea viable en la práctica. En la Tabla 6.1 resumimos los parámetros del GA. El resto de detalles del algoritmo se pueden consultar en la Sección 5.1.2.

Tabla 6.1: Parámetros del GA.

Parámetro	Valor
Población	64
Selección	Torneo binario
Recombinación	SPX para tablas
Mutación	Inversión de bits (p_m=1/*longitud*)
Reemplazo	Elitista
Parada	5000 generaciones

6.2. Primer grupo de instancias: variación en el número de empleados

Con este primer grupo de instancias, vamos a estudiar la influencia que tiene el número de empleados en las soluciones. Esto permite a un gestor de proyectos analizar las implicaciones de contratar más personal para un proyecto determinado. Usamos cuatro instancias diferentes del problema con el mismo proyecto software, es decir, tienen las mismas tareas (diez) y el mismo TPG. La única diferencia entre las instancias es el número de empleados, que toma valores 5, 10, 15 y 20. La dedicación máxima y el salario de los empleados es también el mismo. Además, la restricción **R2** (asociada a las habilidades) no se tiene en cuenta. Es decir, todos los empleados tienen las habilidades necesarias para realizar cualquier tarea dada. Esta situación se ha modelado introduciendo solamente una habilidad requerida en el proyecto y proporcionando a todos los empleados dicha habilidad. El trabajo total que deben realizar los empleados es siempre el mismo en las cuatro instancias. Esperamos, por tanto, que la duración del proyecto en las soluciones propuestas por el algoritmo genético disminuya cuando el número de empleados aumente. Más precisamente, la duración del proyecto y el número de empleados deben tener una relación inversa y su producto debe ser constante. En la Tabla 6.2 mostramos los resultados obtenidos en las cuatro instancias. Para cada caso presentamos la tasa de éxito (porcentaje de las ejecuciones que consiguieron una solución factible), la duración de las soluciones factibles propuestas y el producto del número de empleados y por la duración del proyecto.

Observamos en los resultados que la tasa de éxito disminuye cuando el número de empleados aumenta, es decir, el problema se hace más difícil para el algoritmo genético cuando aumentamos el número de empleados. Cabría esperar que con más empleados fuese más fácil encontrar una solución al problema. No obstante, en esta situación la restricción **R3** (que exige que no haya trabajo extra) es más difícil de satisfacer porque hay más empleados para incumplirla. Además, el espacio de búsqueda es mayor y esto perjudica al proceso de búsqueda. Como predijimos antes, la duración del proyecto disminuye cuando el número de empleados aumenta. De hecho, el producto del número de empleados y la duración media es

Tabla 6.2: Resultados obtenidos cuando el número de empleados cambia.

Empleados	Tasa de éxito (%)	Duración (meses)	$E \times p_{dur}$ (personas-mes)
5	87	$21.88_{0.91}$	$109.40_{4.54}$
10	65	$11.27_{0.32}$	$112.74_{3.17}$
15	49	$7.73_{0.20}$	$115.90_{2.95}$
20	51	$5.88_{0.14}$	$117.56_{2.74}$

muy similar para las cuatro instancias (cuarta columna). No obstante, aumenta ligeramente (casi todas las diferencias son estadísticamente significativas, véase el Apéndice B) con el número de empleados por la misma razón que la tasa de éxito se reduce: las instancias son más difíciles para el GA. El coste total del proyecto software (p_{cost}) es exactamente el mismo en todas las soluciones porque todos los empleados tienen el mismo salario.

6.3. Segundo grupo de instancias: cambio en el número de tareas

Ahora estudiamos la influencia del número de tareas en las soluciones. Esto se corresponde con la posibilidad de abordar proyectos de distinta envergadura usando la misma plantilla. Este tipo de experimentos permite al gestor de proyectos analizar el rendimiento esperado en cada uno de los proyectos sin cambiar el personal. Resolvemos tres instancias donde mantenemos los empleados y cambiamos el proyecto software. En particular, los tres proyectos software tienen diferente número de tareas: 10, 20 y 30. Como en el grupo anterior, todos los empleados tienen el mismo salario y dedicación máxima. Por este motivo, todas las soluciones para el mismo proyecto tienen el mismo coste. Ya que usamos la misma distribución de probabilidad para generar el coste de las tareas en los tres proyectos, esperamos un aumento en el coste del proyecto con el aumento en el número de tareas. Además, no consideramos la restricción **R2**, así que esperamos una relación proporcional entre la duración y el coste de los proyectos. Más aún, si todos los empleados trabajan todo el tiempo para el proyecto, la razón entre el coste y la duración debe ser exactamente la suma del salario de todos los empleados. En las instancias hay cinco empleados con un salario mensual de 10000 €, así que la razón coste-duración debe ser cercana a 50000 €. Presentamos los resultados de las tres instancias en la Tabla 6.3, donde mostramos la tasa de éxito, el coste del proyecto en euros, la duración en meses de las soluciones factibles propuestas, y el coste por mes de los proyectos en euros por mes.

Tabla 6.3: Resultados obtenidos cuando el número de tareas cambia.

Tareas	Tasa de éxito (%)	Coste (€)	Duración (meses)	p_{cost}/p_{dur} (€/mes)
10	73	$980000_{0.00}$	$21.84_{0.87}$	$44944.34_{1720.76}$
20	33	$2600000_{0.00}$	$58.29_{3.76}$	$44748.12_{2265.24}$
30	0	-	-	-

En los resultados podemos observar que el problema se hace más duro cuando el número de tareas aumenta. De hecho, el algoritmo genético no es capaz de obtener ninguna solución factible para el proyecto con 30 tareas. La razón para este comportamiento es la misma que en el grupo previo: cuando el número de tareas aumenta, es más difícil para el GA conseguir una solución que satisfaga la restricción **R3** (trabajo extra). También observamos que tanto el coste de los proyectos (tercera columna) como la duración del

proyecto (cuarta columna) aumentan con el número de tareas. El coste por mes del proyecto (quinta columna) es cercano a 50000 € en las dos instancias que el GA consigue resolver, tal y como predijimos. Este parámetro no puede ser mayor que 50000 € porque esto implicaría una violación de la restricción de trabajo extra. Cuando el valor de este parámetro está cerca del óptimo (50000 € en nuestro caso) indica una asignación eficiente de empleados a tareas. Concluimos, a partir de los resultados, que la asignación conseguida para la instancia de diez tareas es más eficiente que la obtenida para la de 20 tareas. Esto se debe a un aumento del tamaño del espacio de búsqueda cuando cambiamos de diez a veinte tareas.

6.4. Tercer grupo de instancias: cambio en la experiencia de los empleados

En esta sección estudiamos cómo el número de habilidades por empleado, es decir, la experiencia de los empleados, influye en los resultados. Esto permite a un gestor de proyectos analizar cuantitativamente las ventajas de contratar empleados con más experiencia u organizar cursos de formación para personal. Resolvemos cinco instancias con el mismo proyecto software y el mismo número de empleados. Los empleados tienen todos el mismo salario mensual y la misma dedicación máxima. Las instancias difieren en las habilidades de los empleados. Analizamos cinco valores diferentes para el número de habilidades de los empleados: 2, 4, 6, 8 y 10. Estas habilidades se eligen de forma aleatoria del conjunto de diez habilidades del proyecto. Todas las tareas requieren cinco habilidades distintas. Presentamos la tasa de éxito, la duración de los proyectos y el coste por mes en la Tabla 6.4.

Tabla 6.4: Resultados obtenidos cuando cambia el número de habilidades por empleado.

Habilidades	Tasa de éxito (%)	Duración (meses)	p_{cost}/p_{dur} (€/mes)
2	39	$21.71_{0.97}$	$45230.22_{1957.89}$
4	53	$21.77_{0.75}$	$45068.66_{1535.53}$
6	77	$21.98_{0.84}$	$44651.29_{1593.47}$
8	66	$22.00_{0.87}$	$44617.01_{1717.67}$
10	75	$22.11_{1.15}$	$44426.93_{2051.03}$

Observamos que el problema es más duro de resolver con un número bajo de habilidades por empleado, es decir, si la experiencia de los empleados es baja es más difícil asignarlos a las tareas sin violar la restricción **R2**. También podemos apreciar que la duración del proyecto obtenida en las diferentes instancias permanece casi constante con un ligero aumento (no significativo estadísticamente) para los valores más altos de experiencia de los empleados. Esta tendencia indica que el GA es capaz de asignar los empleados a las tareas de un modo más eficiente cuando el nivel de experiencia de los empleados es menor. Una posible razón para esto es que la región factible del espacio de búsqueda aumenta cuando los empleados tienen más habilidades, y por esto la calidad media de las soluciones incluidas en la región factible disminuye.

6.5. Cuarto grupo de instancias: especialización del conocimiento constante

Incluimos en este grupo 18 instancias distintas del problema que difieren en todos los parámetros previamente estudiados. En particular, asignamos diferentes valores al número de empleados, el número de tareas (y, por tanto, el TPG), y el número de habilidades de los empleados. El número total de habilidades del proyecto es 10 en todas las instancias. El número de empleados varía entre 5, 10 y 15 y el número de tareas entre 10, 20 y 30. Se consideran dos rangos de valores separados para el número de habilidades de los empleados: de 4 a 5, y de 6 a 7. Con este estudio queremos comprobar si los parámetros analizados anteriormente de forma aislada influyen del mismo modo cuando interaccionan entre ellos. Mostramos en la Tabla 6.5 la tasa de éxito para todas las instancias.

Tabla 6.5: Tasa de éxito (%) para el cuarto grupo de instancias.

	4-5 habilidades			6-7 habilidades		
	Empleados			Empleados		
Tareas	5	10	15	5	10	15
10	94	97	97	84	100	97
20	0	6	43	0	76	0
30	0	0	0	0	0	0

A partir de estos resultados podemos concluir que las instancias con un mayor número de tareas son más difíciles de resolver que aquéllas con un menor número, como concluimos en la Sección 6.3. En la segunda fila de resultados, podemos observar que cuantos menos empleados hay más difícil resulta la instancia. Esto contrasta con los resultados del primer grupo de instancias (Sección 6.2). La principal diferencia entre los dos grupos reside en las habilidades. En el primer grupo la restricción **R2** (de las habilidades) no se consideró, mientras que en el cuarto grupo sí. Cuando el número de empleados aumenta, es más difícil cumplir la restricción **R3** (del trabajo extra) pero más fácil cumplir la restricción **R2** porque la plantilla está altamente cualificada. Estas dos tendencias están en conflicto, pero en los resultados de la Tabla 6.5 la segunda parece ser predominante (con la excepción de la instancia con 20 tareas, 15 empleados y de 6 a 7 habilidades por empleado).

Para ilustrar mejor el significado de estos resultados, dibujamos las soluciones obtenidas en una gráfica mostrando su coste frente a su duración (Figuras 6.1 y 6.2). Coste y duración son claros criterios de compromiso en cualquier proyecto. A diferencia de las gráficas usadas en optimización multiobjetivo, en éstas representamos todas las soluciones factibles y no sólo la no dominadas. Este es el tipo de gráficas que a un gestor de proyectos software le gustaría ver antes de tomar una decisión sobre el proyecto. Hemos puesto una etiqueta `<tasks>-<employees>` cerca de las soluciones de la misma instancia.

En las figuras, las soluciones de las instancias pueden verse como conjuntos de puntos. Su forma alargada depende de la escala de los ejes (elegidos para mantener las soluciones de todas las instancias en las misma gráfica), no obstante podemos apreciar una ligera inclinación de los conjuntos mostrando el compromiso mencionado entre coste y duración: cuando el coste de una planificación es menor, su duración es mayor.

Como podríamos esperar, cuando el número de empleados disminuye para un número de tareas fijo, la duración del proyecto se alarga. Esta observación se mantiene a pesar de que cada conjunto de puntos representa una instancia diferente con diferente TPG. En las figuras, podemos observar que un número

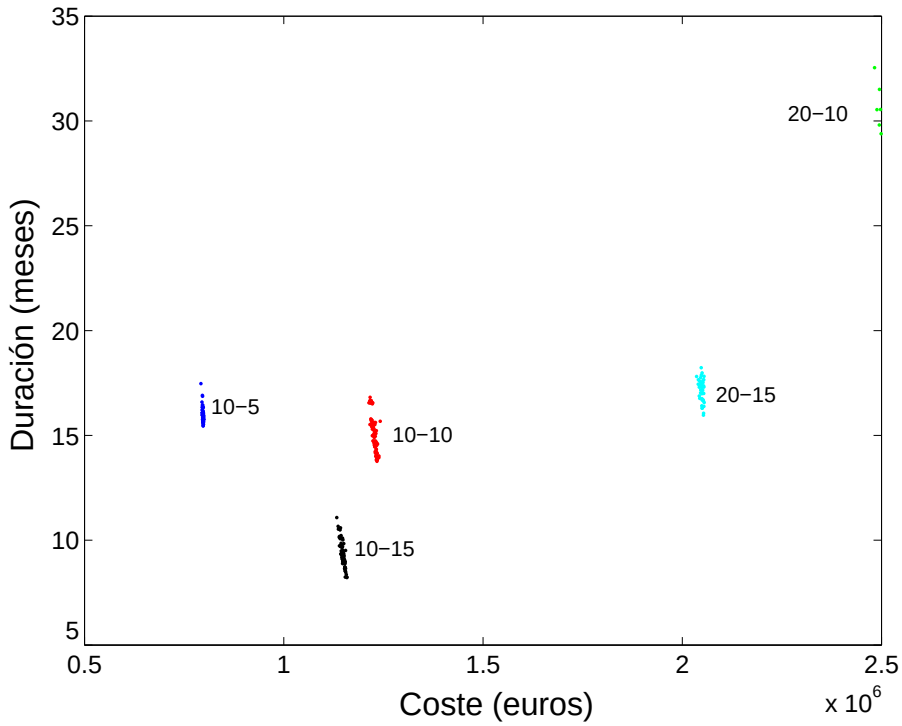


Figura 6.1: Resultados con 4-5 habilidades por empleado. Las etiquetas muestran el número de tareas y empleados de las instancias.

mayor de empleados no significa necesariamente un proyecto más caro en todos los casos. No obstante, no podemos llegar a ninguna conclusión fundamental sobre este hecho porque las instancias pertenecen a proyectos software muy diferentes.

6.6. Quinto grupo de instancias: experiencia de los empleados constante

En este último grupo de 18 instancias estudiamos la influencia del número total de habilidades de un proyecto. Un estudio de este tipo puede ayudar a las grandes compañías, donde un conjunto de personas de experiencia variada tienen que ser asignados óptimamente a proyectos software. En este caso, el rango del número de habilidades por tarea y empleado varía de 2 a 3. El número de tareas puede ser 10, 20 o 30 y el número de empleados toma valores 5, 10 y 15 como en el grupo de instancias anterior. El número de habilidades totales toma valores 5 y 10. La Tabla 6.6 muestra la tasa de éxito obtenida en los resultados.

Como en la sección previa, podemos ver que un aumento en el número de tareas significa un aumento en la dificultad del problema. Por otro lado, la participación de más empleados normalmente implica una disminución en la dificultad de la instancia (es más fácil llevar a cabo el proyecto). No obstante, podemos

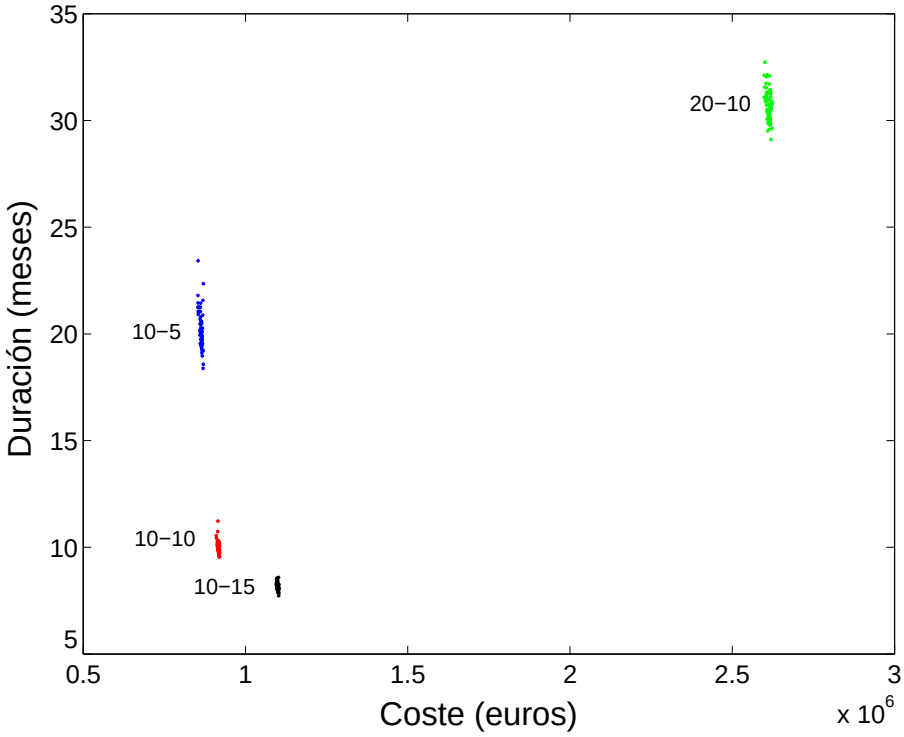


Figura 6.2: Resultados con 6-7 habilidades por empleado. Las etiquetas muestran el número de tareas y empleados de las instancias.

Tabla 6.6: Tasa de éxito (%) para el quinto grupo de instancias.

	5 habilidades			10 habilidades		
	Empleados			Empleados		
Tareas	5	10	15	5	10	15
10	98	99	100	61	85	85
20	6	9	12	8	1	6
30	0	0	0	0	0	0

concluir ahora un hecho adicional: confirmamos, como se esperaba, que un número mayor de habilidades demandadas hace la instancia más difícil, en general, de resolver.

A partir de las Figuras 6.3 y 6.4 concluimos que el coste del proyecto aumenta con el número de tareas, y la duración del proyecto disminuye con el aumento en el número de empleados. Esto también se observó en los grupos de instancias anteriores. No obstante, con más empleados, el coste total del proyecto se reduce en todos los casos, un hecho que no fue observado anteriormente (solamente similar a 10-15 y 20-15 en la Figura 6.1). Anteriormente argumentamos que las diferentes instancias usan diferentes proyectos y, por este motivo, no pudimos obtener ninguna conclusión definitiva. Aquí, estamos en la

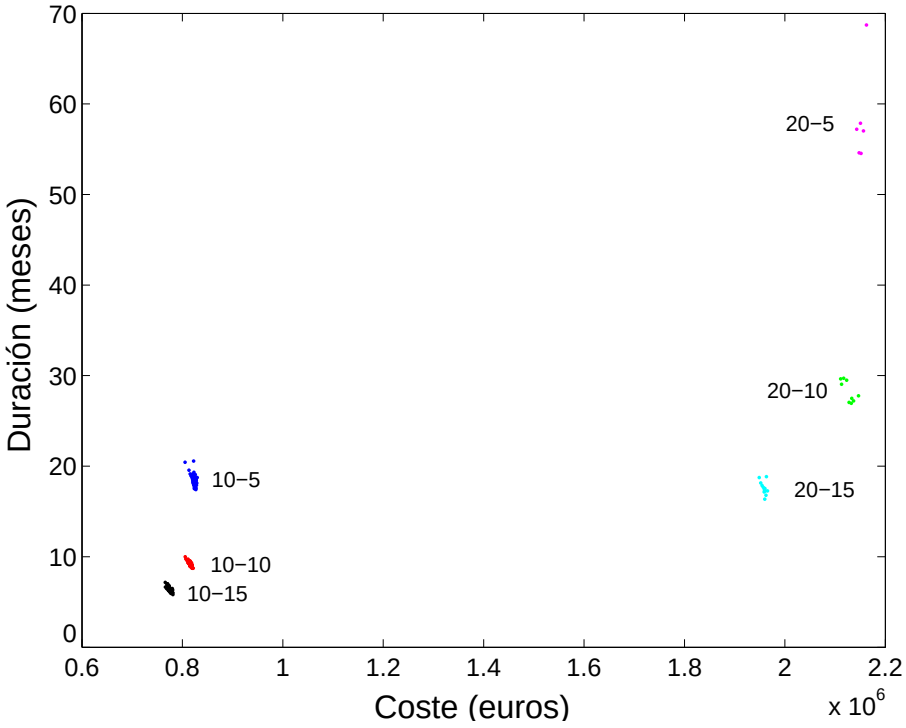


Figura 6.3: Resultados con 5 habilidades requeridas en el proyecto. Las etiquetas muestran el número de tareas y empleados de las instancias.

misma situación, pero analizando las soluciones particulares de las instancias observamos que cuando hay un número mayor de empleados todos ellos trabajan en todas las tareas con un bajo grado de dedicación. De este modo, las tareas se realizan más rápidamente y el coste global del proyecto es bajo.

6.7. Análisis detallado de la dinámica del algoritmo

Para finalizar nuestra presentación de resultados, mostramos la evolución del *fitness* de la mejor solución para las instancias de los dos últimos grupos promediada en las 100 ejecuciones. Nuestro objetivo es ofrecer una traza de la búsqueda realizada por el GA. Hemos agrupado las instancias relacionadas en la misma gráfica para comparar las trazas. Se han seguido tres criterios para agrupar las instancias que han dado lugar a las tres figuras que se presentan en las siguientes páginas (Figuras 6.5, 6.6 y 6.7).

El primer criterio consiste en mantener en la misma gráfica todas aquéllas instancias que tienen el mismo número de habilidades de proyecto, habilidades por empleado, y número de tareas. Como tenemos cuatro posibles configuraciones de habilidades de proyecto y tres de tareas, obtenemos 12 gráficas que se muestran en la Figura 6.5.

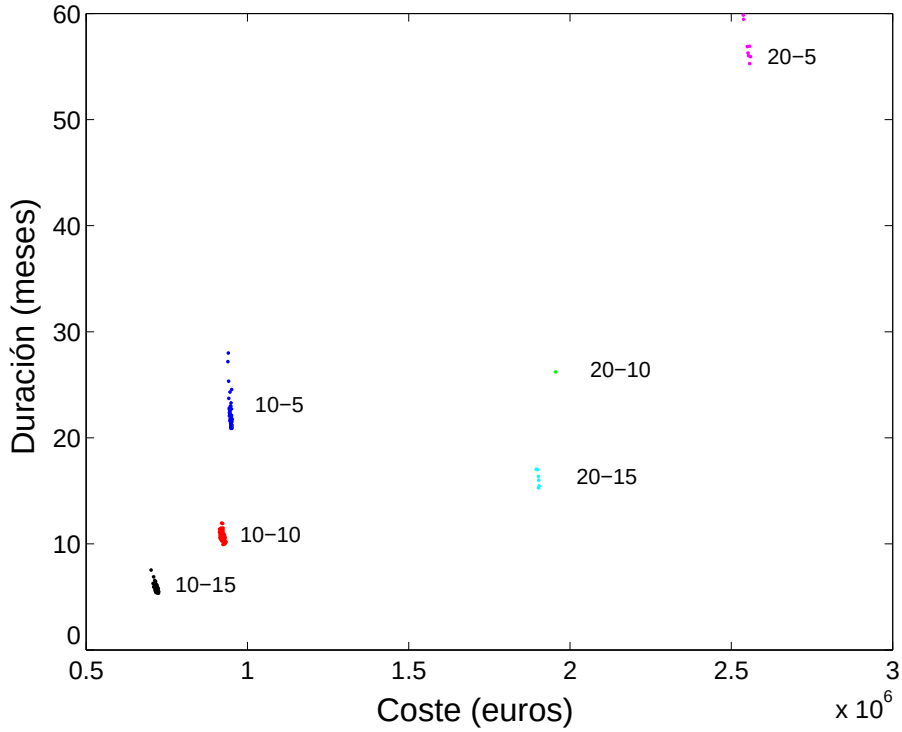


Figura 6.4: Resultados con 10 habilidades requeridas en el proyecto. Las etiquetas muestran el número de tareas y empleados de las instancias.

Observamos evoluciones de *fitness* escalonadas en las gráficas de la fila central (instancias de 20 tareas). La parte plana de la evolución del *fitness* se corresponde con las generaciones en las que el algoritmo genético no encuentra ninguna solución factible. La entrada en la región factible del espacio de búsqueda se produce siempre tras 2000 generaciones aproximadamente. Para estas instancias con 20 tareas observamos, además, una evolución del *fitness* que indica que el algoritmo genético aún no ha convergido, por lo que con un número mayor de evaluaciones se habrían descubierto mejores soluciones. En contraste con las gráficas de la segunda fila, se observa gran suavidad en las curvas de la tercera fila, todas ellas pertenecientes a instancias de 30 tareas donde el GA no consigue soluciones factibles. En este caso, el algoritmo posiblemente requiera más generaciones para entrar en la región factible.

El segundo criterio nos lleva a presentar juntas, en la misma gráfica, las trazas pertenecientes a instancias que tienen el mismo número de empleados y la misma configuración de habilidades (Figura 6.6). De nuevo, tenemos 12 gráficas con tres trazas por gráfica (una por cada número de tareas). La primera observación es que sólo las curvas de las instancias de 10 tareas se mantienen siempre por encima del valor de *fitness* de soluciones factibles (0.01). El punto en el que la curva comienza su ascenso depende del número de empleados. Con un número alto de empleados el ascenso se retrasa, quizás debido al gran tamaño del espacio de búsqueda. En algunas gráficas (como la de las instancias de 5 empleados y 10 habilidades) podemos ver un ascenso modesto de las curvas pertenecientes a las instancias de 20 tareas.

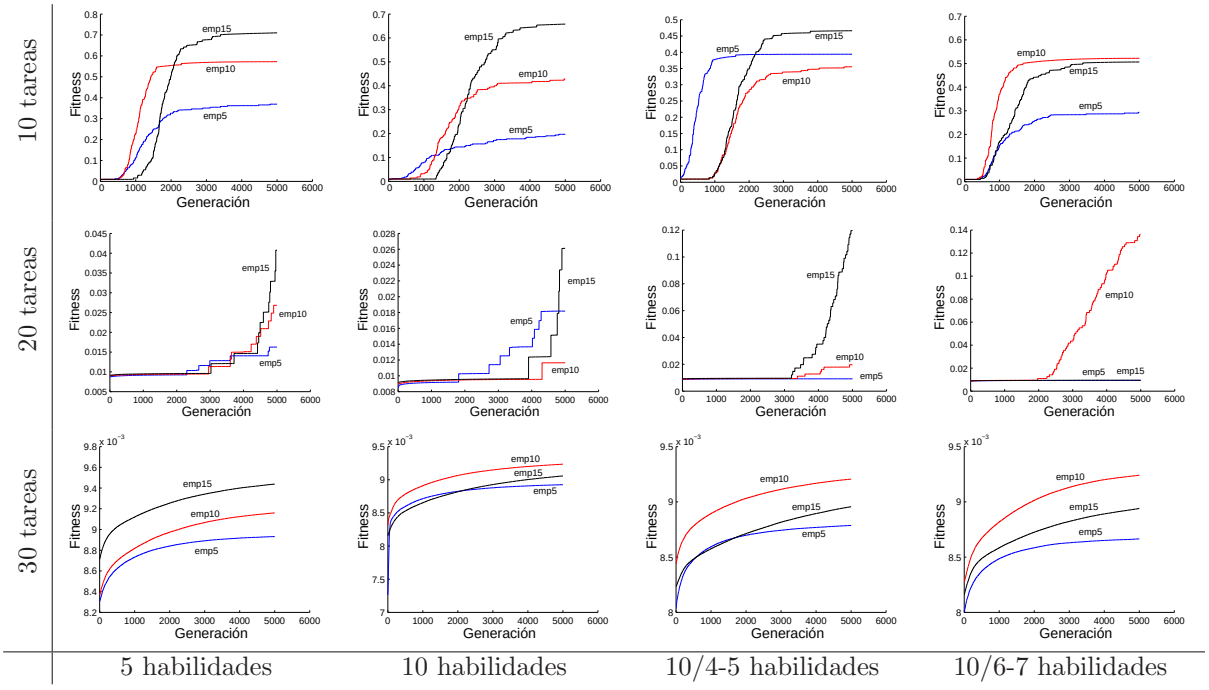


Figura 6.5: Tareas y habilidades fijas.

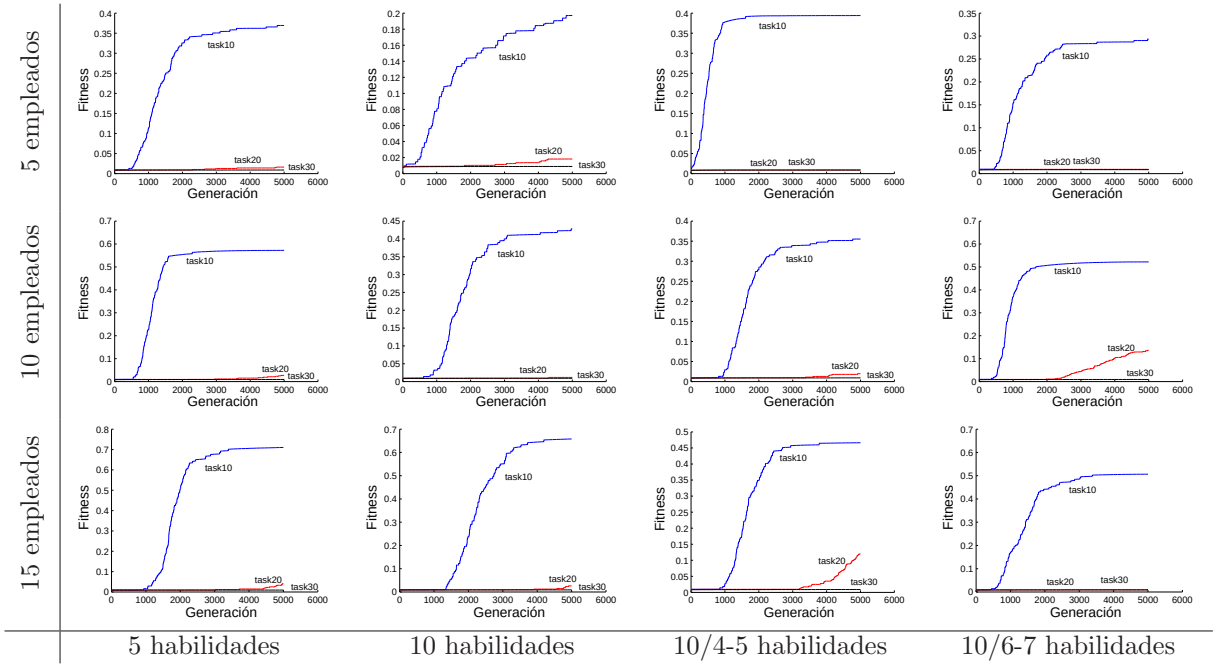


Figura 6.6: Empleados y habilidades fijas.

Por último, el tercer criterio consiste en agrupar las instancias que tienen el mismo número de tareas y empleados, obteniendo así las nueve gráficas de la Figura 6.7. En la primera columna (instancias de 10 tareas) podemos ver que el mejor *fitness* final de las instancias de 5 habilidades está por encima del de las instancias de 10 habilidades. Esto ya fue discutido en la Sección 6.6 cuando observamos que los proyectos con 10 habilidades eran más difíciles de resolver que los de 5. Por otro lado, el punto en el que las curvas comienzan su ascenso se retrasa con el aumento en el número de empleados (esto también se observó en la Figura 6.6). La segunda columna nos ayuda a concluir que un número mayor de empleados hace la búsqueda más sencilla.

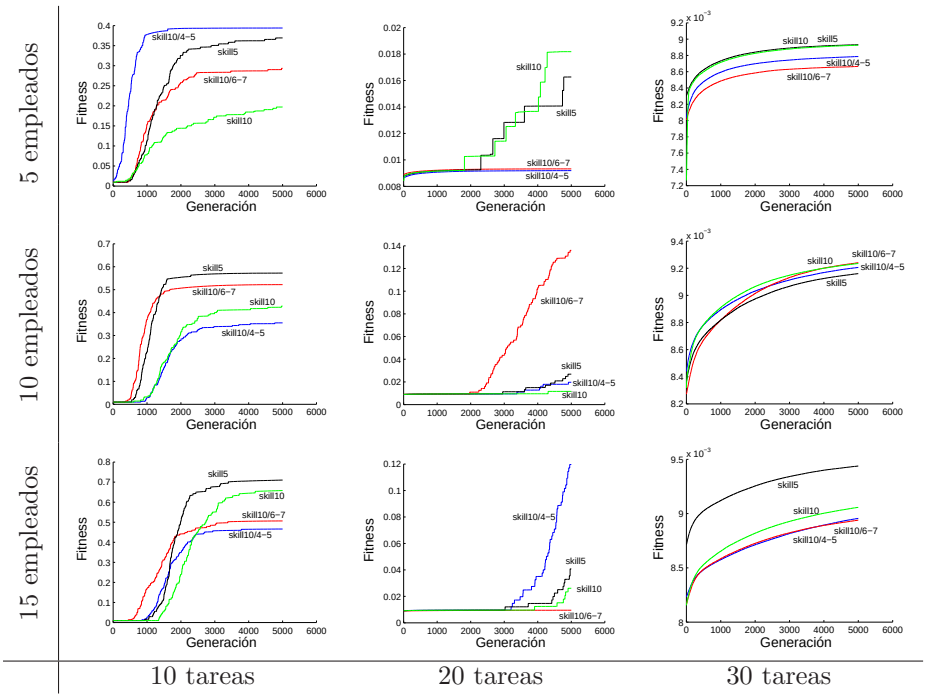


Figura 6.7: Empleados y tareas fijas.

6.8. Conclusiones

En este capítulo hemos abordado el problema general de planificación de proyectos con algoritmos genéticos. Este problema es esencial para la industria de Ingeniería del Software hoy en día: encontrar automáticamente "buenas" soluciones al problema puede ahorrar a las compañías de software tiempo y dinero. Un gestor de proyectos puede estudiar diferentes escenarios usando un GA simple y tomar decisiones sobre los mejores proyectos para su compañía. Más aún, en nuestro enfoque, el gestor puede ajustar los pesos utilizados en la función de *fitness* para representar mejor las prioridades particulares en cada caso. La planificación de proyectos es un problema de optimización combinatoria y una búsqueda exhaustiva puede requerir mucho tiempo para conseguir una solución. Aquí, así como en otros trabajos [54], se confirma claramente la utilidad de las técnicas metaheurísticas. En nuestro caso, además,

abordamos el problema como clase, no como instancias aisladas. Con el análisis que hemos presentado en este capítulo, queremos mostrar que los GA pueden ser una herramienta muy útil para la gestión en Ingeniería del Software. Éstos pueden asignar empleados a tareas en un proyecto de un modo casi óptimo y permiten ensayar distintas configuraciones cambiando la importancia relativa del coste y la duración del proyecto, sin ningún riesgo para el proyecto real por realizarse dentro de un ordenador. Aunque el modelo del proyecto es muy simple, puede servir como un primer paso en la aplicación de algoritmos evolutivos a los experimentos *in silico* en Ingeniería del Software empírica.

Hemos usado algoritmos genéticos, y hemos realizado un análisis en profundidad con un generador de instancias. Resolvimos 48 escenarios distintos y realizamos 100 ejecuciones independientes para cada prueba con el objetivo de obtener alta confianza estadística. Como se esperaba, los resultados muestran que las instancias con más tareas son más difíciles de resolver y sus soluciones son más caras. En la misma línea, los proyectos con mayor número de empleados son más fáciles de abordar y pueden ser llevados a buen término en menos tiempo. Además, el aumento en el número de habilidades requeridas en el proyecto hacen la instancia más difícil, en general, para el GA. Por último, hemos observado que la relación entre empleados y coste no es tan simple: en algunos casos es directa y en otros es inversa.

Capítulo 7

Aplicación de metaheurísticas a la generación de casos de prueba

En este capítulo analizamos la aplicación de varios algoritmos metaheurísticos a la generación automática de casos de prueba. Usamos, concretamente, algoritmos genéticos, estrategias evolutivas, versiones paralelas descentralizadas de estos dos y optimización basada en cúmulos de partículas. En la siguiente sección presentamos los programas objeto para los que pretendemos generar casos de prueba. En la Sección 7.2 mostramos los parámetros de los algoritmos empleados en los experimentos. Tras esto, comparamos los algoritmos paralelos descentralizados con los secuenciales centralizados en la Sección 7.3 y mostramos los resultados de un generador aleatorio en la Sección 7.4. Para profundizar en el enfoque distribuido, se realiza un estudio pormenorizado de distintas alternativas en la Sección 7.5. En la Sección 7.6, comparamos los resultados de ES, GA y el generador aleatorio con PSO. Por último, mostramos algunos resultados obtenidos previamente en la literatura en la Sección 7.7 y concluimos en la Sección 7.8.

7.1. Casos de estudio

Los experimentos se han realizado sobre un conjunto de doce programas en C que abarcan aspectos prácticos y fundamentales de la Informática. Los programas van desde computación numérica (funciones de Bessel) hasta métodos de optimización global (enfriamiento simulado). La mayoría de ellos se han extraído del libro "C Recipes" [229]. La lista de programas se encuentra en la Tabla 7.1, donde presentamos información sobre el número de predicados atómicos, líneas código (LdC), número de argumentos de entrada, una breve descripción de su objetivo y la forma de conseguirlos.

El primer programa, `triangle`, recibe tres números enteros como argumentos y decide si se corresponden con los lados de un triángulo. En caso afirmativo, indica el tipo de triángulo: equilátero, isósceles o escaleno. El siguiente programa, `gcd`, calcula el máximo común divisor de dos argumentos enteros. El programa `calday` calcula el día de la semana de una fecha especificada con tres argumentos enteros. En `crc` se calcula el código de redundancia cíclica (CRC) de trece números enteros dados como argumentos. Los siguientes cuatro programas (`insertion`, `shell`, `quicksort` y `heapsort`) ordenan una lista de entrada con 20 números reales usando algoritmos de ordenación. El programa `select` devuelve el k-ésimo elemento de una lista desordenada de números reales. El primer argumento es k y el resto es la lista

Tabla 7.1: Programas objeto usados en los experimentos. La última columna indica el nombre de la función en C-Recipes.

Programa	Pred.	LdC	Arg.	Descripción	Fuente
`triangle`	21	53	3	Clasifica triángulos	Ref. [206]
`gcd`	5	38	2	Calcula el máximo común denominador	F. Chicano
`calday`	11	72	3	Calcula el día de la semana	`julday`
`crc`	9	82	13	Calcula del código redundante cíclico	`icrc`
`insertion`	5	47	20	Ordenación usando inserción	`piksrt`
`shell`	7	58	20	Ordenación usando shell	`shell`
`quicksort`	18	143	20	Ordenación usando quicksort	`sort`
`heapsort`	10	72	20	Ordenación usando heapsort	`hpsort`
`select`	28	200	21	k-ésimo elemento de una lista desordenada	`selip`
`bessel`	21	245	2	Funciones Bessel J_n y Y_n	`bessj,bessy`
`sa`	30	332	23	Enfriamiento simulado	`anneal`
`netflow`	55	112	66	Optimización de una red	Wegener [282]

desordenada. El siguiente programa (`bessel`) calcula las funciones de Bessel dado el orden n (entero) y un valor real. El programa `sa` resuelve una instancia del problema del viajante de comercio (TSP) con diez ciudades usando enfriamiento simulado. Los tres primeros argumentos son las semillas para los generadores de números aleatorios y los otros 20 números reales representan la posición (en un plano) de cada ciudad. Finalmente, el programa `netflow` optimiza una arquitectura de red para conseguir flujo de datos máximo. La entrada es la descripción de la red a optimizar. Su tamaño está limitado a 6 nodos y 10 conexiones. Este último programa es el único con restricciones en sus argumentos. Los primeros veinte argumentos deben ser números enteros comprendidos entre 1 y 6, ambos inclusive. Cuando se intente ejecutar el programa con un valor inadecuado en alguno de esos argumentos, la función de distancia devolverá el mayor número real representable en una máquina, tal y como se describe en la Sección 5.2.1.

En primer lugar, hemos usado dos programas sin bucles: `triangle`, que posee predicados atómicos comprobando la igualdad de enteros, y `calday`, con objetivos parciales dispersos en el espacio de búsqueda. De entre los programas con bucles, hemos escogido `insertion`, `shell`, `quicksort`, `heapsort` y `select` por su relevancia dentro de la Informática. Los programas `gcd` y `crc` representan a los algoritmos de computación entera, mientras que `bessel` representa a los de computación de punto flotante. Por último, `sa` y `netflow` son programas complejos con aplicación real.

7.2. Parámetros de los algoritmos

En esta sección presentamos los parámetros de los algoritmos usados en los experimentos de este capítulo. Los parámetros de las versiones centralizadas y descentralizadas de GA y ES se presentan en las Tablas 7.2 y 7.3. Los parámetros del PSO se encuentran en la Tabla 7.4.

Los parámetros seleccionados para ES son el resultado de un estudio que puede consultarse en la Sección A.1.1. En dicho estudio se llega a la conclusión de que la mejor configuración es $\mu = 5$ y $\lambda = 1$. El hecho de usar en $\mu = 25$ y $\lambda = 5$ en dES es para conseguir que cada subalgoritmo trabaje con una subpoblación de cinco individuos y genere uno en cada iteración (como sugiere el estudio de la Sección A.1.1). Con objeto de hacer una comparación lo más justa posible entre algoritmos evolutivos,

Tabla 7.2: Parámetros de los algoritmos distribuidos dES y dGA.

Parámetros	dES	dGA
Tamaño de población	25	25
Selección	Aleatoria	Aleatoria
Recombinación	-	DPX ($p_c = 1.0$)
Mutación	Gaussiana	Perturbación $N(0,1)$ ($p_m = 0.6$)
Descendencia	1 por isla	1 por isla
Reemplazo	$(\mu + \lambda)$	$(\mu + \lambda)$
Criterio de parada	Objetivo o 500 evaluaciones	Objetivo o 500 evaluaciones
Islas	5	
Topología	Anillo unidireccional	
Tipo de migración	Asíncrono	
Periodo de migración	10	
Selección de emigrantes	Torneo binario	
Tasa de migración	1	
Reemplazo de migración	Reemplazo del peor si el inmigrante es mejor	

Tabla 7.3: Parámetros de los algoritmos centralizados ES y GA.

Parámetros	ES	GA
Tamaño de población	25	25
Selección	Aleatoria	Aleatoria
Recombinación	-	DPX ($p_c = 1.0$)
Mutación	Gaussiana	Perturbación $N(0,1)$ ($p_m = 0.6$)
Descendencia	5	5
Reemplazo	$(\mu + \lambda)$	$(\mu + \lambda)$
Criterio de parada	Objetivo o 500 evaluaciones	Objetivo o 500 evaluaciones

Tabla 7.4: Parámetros de PSO.

Parámetro	Valor
Partículas	10
w	0.729
$c1$	1.494
$c2$	1.494
Criterio de parada	Objetivo o 1000 evaluaciones

usamos los mismos parámetros μ y λ en todos ellos. El criterio de parada consiste en encontrar un caso de prueba que cumpla el objetivo parcial o alcanzar un máximo de 500 evaluaciones (en los algoritmos distribuidos cada una de las cinco islas realiza un máximo de 100 evaluaciones).

La configuración de los operadores de GA son el resultado de un estudio previo para determinar su mejor configuración. Dicho estudio se encuentra en la Sección A.1.2. El operador de recombinación usado en estos algoritmos es el cruce de dos puntos (DPX) con probabilidad 1.0. No obstante, de los dos hijos creados por el operador sólo se elige uno. El operador de mutación en GA (y dGA) consiste en sumar al

valor de un argumento de entrada un número aleatorio siguiendo una distribución normal de media 0 y desviación estándar 1 (como se mencionó en la Sección 5.2.5) con probabilidad 0.6.

Los algoritmos distribuidos usan cinco islas conectadas siguiendo una topología de anillo unidireccional. Para la migración, se elige un individuo usando selección por torneo binario y se envía a la isla vecina cada diez pasos del bucle del subalgoritmo. El individuo emigrado se inserta en la subpoblación destino si es mejor que el peor individuo de esa subpoblación. Al principio de la búsqueda, el generador de casos de prueba inserta en la población un individuo (caso de prueba) que alcanza el predicado asociado con el objetivo parcial perseguido, tal y como mencionamos en la Sección 5.2.3.

Las máquinas usadas para los experimentos son Pentium 4 a 2.8 GHz con 512 MB de RAM y sistema operativo Linux (versión del *kernel* 2.4.19-4GB). Realizamos 30 ejecuciones independientes de los generadores de casos de prueba para cada programa objeto. En las tablas de resultados presentamos las medias y las desviaciones típicas de esas 30 ejecuciones.

7.3. Algoritmos descentralizados frente a centralizados

En esta sección comparamos los resultados obtenidos por los generadores de casos de prueba que usan como motor de búsqueda algoritmos evolutivos descentralizados y centralizados. En concreto, comparamos la estrategia evolutiva distribuida (dES) con la panmíctica (ES) y el algoritmo genético distribuido (dGA) con el panmíctico (GA). En los algoritmos descentralizados, cada isla (subalgoritmo) se ejecuta en máquinas diferentes (ejecución paralela). Con esta comparativa queremos estudiar la influencia del diseño descentralizado en la búsqueda. Por este motivo, fijamos el tamaño de la población y el número máximo de evaluaciones en todos los algoritmos al mismo valor. Debemos recordar aquí que los algoritmos evolutivos pueden ser ejecutados muchas veces durante una ejecución del generador de casos de prueba (véase la Sección 5.2.3). Por esto, el número total de evaluaciones puede ser mayor que el número total de evaluaciones de una ejecución (500).

En la Tabla 7.5 presentamos el porcentaje de cobertura corregida de condiciones, el número de evaluaciones (ejecuciones del programa objeto) requeridas para conseguir la cobertura alcanzada y el tiempo del proceso de generación completo (en segundos) para los algoritmos dES y ES. Señalamos con negrita los mejores resultados (los valores mayores en el caso de cobertura y los menores en el caso del número de evaluaciones y el tiempo).

Tabla 7.5: Resultados obtenidos con dES y ES para todos los programas.

	dES			ES		
Programa	Cobertura (%)	Evaluaciones	Tiempo (s)	Cobertura (%)	Evaluaciones	Tiempo (s)
`triangle`	99.92 0.44	1898.43 1583.28	**5.50** 4.10	**100.00** 0.00	**1207.30** 766.99	7.80 4.04
`gcd`	**100.00** 0.00	39.93 10.02	0.80 0.40	**100.00** 0.00	**15.67** 2.10	**0.53** 0.50
`calday`	**97.88** 3.66	**2188.17** 2036.97	**7.47** 3.64	97.73 3.47	2586.30 2156.01	23.97 9.55
`crc`	**100.00** 0.00	39.80 66.44	1.43 1.41	**100.00** 0.00	**12.83** 10.67	**0.80** 0.79
`insertion`	**100.00** 0.00	**10.00** 0.00	0.60 0.49	**100.00** 0.00	**10.00** 0.00	**0.17** 0.37
`shell`	**100.00** 0.00	**10.00** 0.00	0.53 0.50	**100.00** 0.00	**10.00** 0.00	**0.30** 0.46
`quicksort`	**94.12** 0.00	**10.00** 0.00	**13.57** 0.50	**94.12** 0.00	**10.00** 0.00	39.43 0.67
`heapsort`	**100.00** 0.00	**10.00** 0.00	0.47 0.50	**100.00** 0.00	**10.00** 0.00	**0.43** 0.50
`select`	**83.33** 0.00	34.17 12.25	**14.40** 0.49	**83.33** 0.00	**13.30** 1.75	41.03 0.60
`bessel`	**97.56** 0.00	481.33 851.80	**11.53** 1.82	**97.56** 0.00	**198.63** 396.11	32.20 2.56
`sa`	99.55 0.75	**1451.13** 1493.89	**4330.20** 2763.58	**99.77** 0.58	1730.43 1757.05	10302.73 8471.02
`netflow`	**98.17** 0.00	579.83 218.59	**1440.50** 225.02	**98.17** 0.00	**362.40** 246.16	3242.50 543.44

De los resultados de la Tabla 7.5 concluimos que no hay diferencia entre los dos algoritmos con respecto a la cobertura. De hecho, en aquellos programas donde las medias difieren (`triangle`, `calday` y

`sa`), un test estadístico (los detalles están en la Sección B.2) revela que la diferencia no es significativa. Con respecto al esfuerzo, observamos un número de evaluaciones ligeramente más elevado en el caso de dES en programas en los que se consigue la misma cobertura con ambos algoritmos (`gcd`, `crc`, `select`, `bessel` y `netflow`). Esto no se esperaba, ya que nuestro objetivo era conseguir mayor cobertura y reducir el coste computacional usando estrategias evolutivas distribuidas. Para la generación de casos de prueba, nuestra primera conclusión es que esto no se cumple, al menos para el modelo de islas que hemos usado, y la única mejora obtenida al usar dES es una reducción en el tiempo de ejecución. Este resultado no es generalizable, ya que muchos otros trabajos muestran que las versiones distribuidas superan claramente a la centralizadas [4, 12, 115]. Esto indica que los algoritmos distribuidos en la generación de casos de prueba deben cooperar de distinto modo para aventajar en eficacia a los algoritmos centralizados.

Es importante mencionar que para `netflow` los algoritmos encuentran varios casos de prueba para los que no termina la ejecución, lo cual es un claro ejemplo de que la metodología seguida funciona y tiene utilidad práctica. No obstante, para la realización de los experimentos tuvimos que incorporar un mecanismo que detiene el programa objeto cuando lleva cierto tiempo en ejecución. En concreto, si la ejecución del programa tarda más de un minuto, el generador lo detiene para continuar con la búsqueda. Este hecho explica los altos tiempos observados en la Tabla 7.5 (y, en menor medida, en los siguientes resultados) para dicho programa (cuando el programa termina sin ser abortado su ejecución no dura más de un segundo).

Con respecto al tiempo de ejecución obtuvimos el resultado esperado. La ejecución de dES es más rápida que la de ES en programas más lentos porque cada subalgoritmo se ejecuta en paralelo. La ganancia es sublineal (hasta 3.21 veces más rápido con 5 procesadores), ya que los algoritmos no hacen exactamente lo mismo y existe una importante parte secuencial en el algoritmo distribuido. El proceso de arranque de dES requiere más tiempo debido a la ejecución de las islas en diferentes máquinas (conexión a cada una de ellas y arranque de los procesos correspondientes). Por esta razón, podemos observar una ventaja de ES para los programas más rápidos.

Una conclusión de estos resultados es que `insertion`, `shell`, y `heapsort` son los programas más sencillos del banco de pruebas. El generador de casos de prueba es capaz de conseguir cobertura total sólo con los diez primeros casos de prueba generados aleatoriamente, es decir, no es necesario usar ningún algoritmo de optimización. A estos tres programas les siguen otros dos: `gcd` y `crc`.

Ahora, presentamos en la Tabla 7.6 los resultados obtenidos por el generador de casos de prueba usando como motor de búsqueda el algoritmo genético paralelo distribuido y el secuencial centralizado.

Tabla 7.6: Resultados obtenidos con dGA y GA para todos los programas.

	dGA			GA		
Programa	Cobertura (%)	Evaluaciones	Tiempo (s)	Cobertura (%)	Evaluaciones	Tiempo (s)
`triangle`	**99.84** 0.61	**3004.43** 2820.07	**7.53** 5.75	99.67 1.04	3209.47 1985.48	20.30 10.97
`gcd`	**100.00** 0.00	445.60 155.66	**1.57** 0.56	**100.00** 0.00	**257.20** 198.82	2.00 1.41
`calday`	**90.91** 0.00	304.17 107.04	**10.43** 0.50	**90.91** 0.00	**75.03** 42.81	28.53 0.96
`crc`	**100.00** 0.00	14.83 21.85	0.83 0.90	**100.00** 0.00	**10.37** 1.97	**0.73** 0.63
`insertion`	**100.00** 0.00	**10.00** 0.00	0.47 0.50	**100.00** 0.00	**10.00** 0.00	**0.40** 0.49
`shell`	**100.00** 0.00	**10.00** 0.00	**0.37** 0.48	**100.00** 0.00	**10.00** 0.00	0.43 0.50
`quicksort`	**94.12** 0.00	**10.00** 0.00	**13.20** 0.60	**94.12** 0.00	**10.00** 0.00	36.43 0.50
`heapsort`	**100.00** 0.00	**10.00** 0.00	0.50 0.81	**100.00** 0.00	**10.00** 0.00	**0.33** 0.47
`select`	**83.33** 0.00	322.07 167.71	**14.48** 0.86	**83.33** 0.00	**83.20** 70.96	36.07 0.77
`bessel`	**97.56** 0.00	550.67 259.75	**13.57** 0.76	**97.56** 0.00	**533.03** 418.76	38.63 3.02
`sa`	**96.78** 0.51	335.33 1041.23	**3865.30** 696.46	96.72 0.42	**176.63** 697.55	6529.63 862.35
`netflow`	96.36 0.16	937.33 653.39	**203.33** 365.22	**96.42** 0.36	**917.90** 974.56	240.13 134.52

De estos resultados concluimos, como en la comparación anterior, que no hay diferencia entre los dos algoritmos en la cobertura obtenida (no hay ninguna diferencia estadísticamente significativa). El número de evaluaciones es ligeramente superior en dGA, y el tiempo de ejecución de dGA es menor que el de GA. De aquí concluimos que el modo en que se explora el espacio de búsqueda en las versiones distribuidas de los EA no parece mejorar los resultados con respecto a las versiones panmícticas. Este hecho es una contribución inesperada de nuestro estudio, ya que, generalmente, se esperan ventajas de la búsqueda descentralizada.

A pesar de lo anterior, debido a la ejecución paralela, las versiones distribuidas pueden ser adecuadas para programas computacionalmente costosos, como `sa`. Podemos también paralelizar la versión panmíctica usando, por ejemplo, un esquema maestro-esclavo (véase el modelo paralelo maestro-esclavo en la Sección 4.3.2). Pero, en este último caso, el algoritmo necesita sincronizarse con todas las máquinas una vez en cada iteración. Esa sincronización puede ser muy dañina para el tiempo de ejecución del proceso completo, especialmente cuando el tiempo de ejecución del programa objeto depende del caso de prueba. Debido a esta diferencia en el tiempo de ejecución, muchos procesadores podrían estar ociosos durante largos periodos de tiempo. En el EA distribuido, la sincronización entre las máquinas se realiza cuando todos los subalgoritmos se detienen y, por esto, el número de puntos de sincronización es muy bajo (es exactamente el número de ejecuciones del dEA durante una ejecución del generador de casos de prueba), reduciendo el tiempo total de la generación. En conclusión, los algoritmos paralelos distribuidos como los que hemos utilizado en esta sección, a pesar de no superar en eficacia a los secuenciales, pueden ser útiles para reducir el tiempo requerido por la generación de casos de prueba.

7.4. Generación aleatoria de casos de prueba

Para confirmar que `insertion`, `shell`, `heapsort`, `gcd`, y `crc` son los programas más simples del banco de pruebas, hemos usado un generador de casos de prueba aleatorio que genera un máximo de 20000 casos de prueba y mostramos los resultados en la Tabla 7.7.

Tabla 7.7: Resultados obtenidos con generación aleatoria para todos los programas.

	Generador de casos de prueba aleatorio		
Programa	Cobertura (%)	Evaluaciones	Tiempo (s)
`triangle`	$51.22_{0.00}$	$141.67_{105.05}$	$92.30_{1.70}$
`gcd`	$80.00_{0.00}$	$3.33_{1.62}$	$94.77_{2.08}$
`calday`	$95.45_{0.00}$	$74.97_{35.63}$	$95.10_{3.41}$
`crc`	$100.00_{0.00}$	$5.97_{3.28}$	$0.33_{0.47}$
`insertion`	$100.00_{0.00}$	$1.07_{0.25}$	$0.10_{0.30}$
`shell`	$100.00_{0.00}$	$1.00_{0.00}$	$0.07_{0.25}$
`quicksort`	$94.12_{0.00}$	$2.07_{1.03}$	$123.23_{0.62}$
`heapsort`	$100.00_{0.00}$	$1.00_{0.00}$	$0.10_{0.30}$
`select`	$11.11_{0.00}$	$1.00_{0.00}$	$99.87_{0.76}$
`bessel`	$70.73_{0.00}$	$158.10_{107.44}$	$97.70_{0.69}$
`sa`	$96.67_{0.30}$	$639.03_{3429.60}$	$5861.73_{12.51}$
`netflow`	$95.38_{0.65}$	$9225.77_{6658.85}$	$133.40_{0.88}$

Como esperábamos, `crc`, `insertion`, `shell` y `heapsort` se cubren con menos de diez casos de prueba en media. No obstante, en contra de nuestras predicciones, `gcd` no se puede cubrir con generación aleatoria,

sólo el 80 % de los objetivos parciales se cubren tras 20000 casos de prueba aleatorios. Además, este 80 % se cubre con los primeros tres o cuatro casos generados. La razón es que hay dos igualdades dentro del programa, las cuales son muy difíciles de cubrir con casos aleatorios. En los resultados de la generación aleatoria, descubrimos que la cobertura obtenida por `quicksort` es la misma que la obtenida con los generadores de casos de prueba basados en EA; es decir, que los generadores de casos de prueba que usan EA para la búsqueda no mejoran los resultados de una búsqueda aleatoria. Esto revela que es muy fácil conseguir el 94.12 % de cobertura, pero muy difícil conseguir la cobertura total. La razón es que hay un predicado que comprueba un fallo de una asignación de memoria dinámica y otro que comprueba un desbordamiento de pila. Es decir, hay una pérdida dependiente del entorno que hace que 94.12 % sea el máximo absoluto para este programa en el entorno de ejecución dado. En conclusión, no necesitamos algoritmos metaheurísticos para generar casos de prueba en el caso de los programas `crc`, `insertion`, `shell`, `quicksort` y `heapsort`; los objetivos parciales alcanzables se pueden cubrir con generación aleatoria. Por esta razón, descartamos estos programas para la realización de los experimentos de las siguientes secciones.

7.5. Análisis del enfoque distribuido

En esta sección vamos a estudiar la influencia de algunos parámetros de los algoritmos distribuidos para, no sólo caracterizarlos mejor, sino también asegurarnos de que los resultados previos no son el producto de una configuración inicial poco acertada. Los programas `netflow` y `sa` requieren mucho tiempo de ejecución, mientras que para `triangle`, `gcd`, `select` y `bessel`, tanto dGA como dES obtienen la misma cobertura en todas las ejecuciones independientes (posible pérdida de cobertura dependiente del entorno) o una cobertura demasiado cercana al 100 % como para apreciar diferencias significativas en los resultados. Por este motivo, usamos el algoritmo dES y el programa `calday` en los experimentos cuyos resultados mostramos en esta sección.

7.5.1. Análisis del modo de búsqueda

Inicialmente, estudiaremos el comportamiento del algoritmo cuando cada isla busca un objetivo diferente (modo de búsqueda *diff*), tal y como se propuso en la Sección 5.2.3. En los experimentos previos todos los subalgoritmos buscaban el mismo objetivo parcial (modo de búsqueda *same*).

Pero antes de mostrar los resultados debemos discutir un pequeño detalle. Los algoritmos de las secciones previas se detienen cuando una de las islas encuentra el objetivo, ya que todas ellas buscan el mismo. Ahora, usando el modo de búsqueda *diff* vamos a cambiar la condición de parada del algoritmo. Puesto que todas las islas buscan un objetivo potencialmente distinto al mismo tiempo, parece razonable detener el algoritmo tras un número de evaluaciones predefinidas, con el propósito de encontrar más objetivos parciales en una ejecución del algoritmo.

Este número máximo de evaluaciones es 500 (100 en cada isla, como en los experimentos previos). En la Tabla 7.8 mostramos el porcentaje de cobertura, el número de evaluaciones, y el tiempo de los dos modos de búsqueda.

Como podemos observar, los mejores resultados en cobertura y tiempo son los obtenidos cuando todas las islas buscan el mismo objetivo parcial (modo de búsqueda *same*). Además, las diferencias en todos los valores son estadísticamente significativas (véase la Tabla B.3 en la Sección B.2). Es decir, la colaboración entre las islas es fructífera cuando todos buscan el mismo objetivo, y no cuando buscan objetivos diferentes.

Tabla 7.8: Comparación de dos versiones de dES con distinto modo de búsqueda.

Modo de búsqueda	Cobertura (%)	Evaluaciones	Tiempo (s)
same	**97.88** 3.66	2188.17 2036.97	**7.47** 3.64
diff	92.12 3.09	**693.83** 1288.95	10.07 1.71

7.5.2. Análisis del criterio de parada

Ahora, vamos a estudiar el criterio de parada usado en dES. Como mencionamos en la Sección 5.2.3, todos los casos de prueba que cubren un objetivo parcial no cubierto se almacenan también en la tabla de cobertura, incluso si el objetivo cubierto no es el que se busca actualmente. No obstante, el algoritmo se detiene sólo cuando se cubre el objetivo parcial buscado o se alcanza el número máximo de evaluaciones. Aquí estudiamos la alternativa de detener el algoritmo cuando un caso de prueba cubre un objetivo parcial no cubierto (no necesariamente el que se busca). En la Tabla 7.9 comparamos esta alternativa (condición de parada *any*) con la anteriormente usada (condición de parada *obj*).

Tabla 7.9: Comparación de dos versiones de dES con diferente criterio de parada.

Criterio de parada	Cobertura (%)	Evaluaciones	Tiempo (s)
obj	**97.88** 3.66	**2188.17** 2036.97	**7.47** 3.64
any	97.73 3.27	2640.47 2090.55	9.17 4.47

Podemos observar en los resultados que la condición de parada *obj* tiene una ligera ventaja sobre *any*. No obstante, las diferencias no son estadísticamente significativas, así que no podemos afirmar que la condición de parada usada en las secciones anteriores (*obj*) sea mejor que la introducida en esta sección.

7.5.3. Análisis del número de semillas

El siguiente estudio concierne al número de individuos usados para inicializar el algoritmo de optimización. El generador de casos de prueba usado en los experimentos previos inicializa cada isla de dES con un individuo (caso de prueba) que alcanza el predicado atómico asociado al objetivo parcial. Si hay suficientes casos de prueba que alcancen el predicado atómico en la tabla de cobertura, todas las islas recibirán un caso de prueba diferente. En caso contrario, varias islas recibirán el mismo caso de prueba. Examinamos en estos experimentos tres valores distintos para el número de semillas utilizadas en la inicialización de las islas. El objetivo de este estudio es comprobar si el número de "buenas" soluciones en la población inicial de dES mejora la precisión del generador de casos de prueba. En la Tabla 7.10 mostramos los resultados obtenidos cuando se comparan tres generadores usando 1, 2 y 3 semillas, respectivamente.

Tabla 7.10: Comparación de tres versiones del generador de casos de prueba con diferente número de semillas en la población inicial de dES.

Semillas	Cobertura (%)	Evaluaciones	Tiempo (s)
1	**97.88** 3.66	2188.17 2036.97	**7.47** 3.64
2	94.85 4.35	**1300.90** 1760.61	9.17 2.93
3	95.91 4.45	1745.30 2292.11	8.53 4.04

El mayor porcentaje de cobertura se obtiene cuando se usa una semilla (lo cual es contraintuitivo). No encontramos diferencias estadísticamente significativas para el número de evaluaciones ni para el tiempo (véase la Sección B.2). Una posible razón de este resultado inesperado podría ser que cada isla de dES tiene que dividir su búsqueda entre dos buenas soluciones de la población (las dos semillas) y, por esto, los descendientes de ambas semillas alcanzan el objetivo más tarde. No obstante, esto no se observa cuando se usan tres semillas. En este caso, alguna de las semillas tiene que estar necesariamente repetida en la población (ya que sólo se almacenan diez semillas en la tabla de cobertura para cada objetivo parcial) y esto puede beneficiar la evolución de los individuos redundantes. No obstante, son necesarios más experimentos para confirmar (o refutar) esta hipótesis.

7.5.4. Análisis del periodo de migración

Por último, vamos a estudiar la influencia del periodo de migración en los resultados. El periodo de migración es el número de pasos entre dos migraciones consecutivas y es una medida del acoplamiento entre las islas en el EA distribuido. En los experimentos previos el periodo de migración era 10. En esta sección estudiaremos cuatro valores más: 30, 50, 70 y 90. Puesto que el número máximo de pasos de cada subalgoritmo es 100, un periodo de migración de 90 significa que no hay casi comunicación entre las islas. Queremos comprobar con este experimento si la colaboración entre las islas beneficia la búsqueda o no. En la Tabla 7.11 mostramos los resultados.

Tabla 7.11: Comparación de cinco versiones del generador de casos de prueba con diferente periodo de migración en dES.

Periodo de migración	Cobertura (%)	Evaluaciones	Tiempo (s)
10	97.88 $_{3.66}$	2188.17 $_{2036.97}$	7.47 $_{3.64}$
30	98.18 $_{3.02}$	**1861.13** $_{1609.42}$	7.27 $_{3.48}$
50	98.33 $_{2.99}$	2085.97 $_{1555.87}$	7.20 $_{3.27}$
70	98.94 $_{1.92}$	2073.27 $_{1540.50}$	7.13 $_{3.69}$
90	**99.09** $_{2.16}$	2005.50 $_{1113.87}$	**6.23** $_{2.94}$

Podemos observar que los resultados mejoran cuando el periodo de migración aumenta. Esto no es sorprendente ya que, como vimos en la Sección 7.3, el enfoque distribuido no es mejor que el centralizado. No obstante, los valores de la Tabla 7.11 no son estadísticamente significativos.

7.6. Resultados de PSO

En esta sección presentamos los resultados obtenidos por los generadores de casos de prueba con PSO como motor de búsqueda y los comparamos con otros tres generadores: dos que usan ES y GA, y un generador aleatorio (RND). En la Tabla 7.12 presentamos la cobertura corregida de condiciones y el número de evaluaciones necesarias para obtener la cobertura alcanzada. Los programas utilizados para estos experimentos son: `triangle`, `calday`, `select`, `bessel`, `sa` y `netflow`.

En primer lugar, podemos observar que GA y RND obtienen de forma general la cobertura más baja. En segundo lugar, comparando PSO y ES observamos que su eficacia es aproximadamente la misma. En `calday`, `select`, y `sa` la cobertura media que obtiene PSO es mayor que la de ES, pero ES supera a PSO en `triangle` y `netflow`. Además, la validación estadística (Sección B.2) demuestra que las únicas

Tabla 7.12: Resultados obtenidos con PSO comparados con los de ES, GA y RND.

Programa	PSO		ES		GA		RND	
	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.
`triangle`	93.98 3.43	11295.77 6294.94	**99.84** 0.85	**2370.03** 2200.28	99.67 1.04	3209.47 1985.48	51.22 0.00	141.67 105.05
`calday`	**100.00** 0.00	179.33 155.71	98.18 3.44	3166.47 3336.17	90.91 0.00	75.03 42.81	95.45 0.00	**74.97** 35.63
`select`	**88.89** 0.00	380.13 22.10	83.33 0.00	13.27 2.72	83.33 0.00	83.20 70.96	11.11 0.00	**1.00** 0.00
`bessel`	**97.56** 0.00	**116.90** 76.69	**97.56** 0.00	350.63 712.41	**97.56** 0.00	533.03 418.76	70.73 0.00	158.10 107.44
`sa`	**100.00** 0.00	**165.67** 13.83	99.94 0.30	2337.30 1726.49	96.72 0.42	176.63 697.55	96.67 0.30	639.03 3429.60
`netflow`	97.77 0.56	4681.70 4835.78	**98.17** 0.00	**307.77** 200.01	96.42 0.36	917.90 974.56	95.38 0.65	9225.77 6658.85

diferencias significativas son las de `triangle` y `select`. Hemos observado que en `triangle`, PSO obtiene menor cobertura que ES debido a la perturbación que añadimos a la técnica (los experimentos previos que realizamos para decidir los parámetros de los algoritmos así lo demuestran). La perturbación después de cada paso del algoritmo aumenta de forma drástica la capacidad de exploración del mismo. Eso ayuda a programas como `calday` o `select`, que poseen objetivos parciales en lugares muy alejados de la región donde se encuentran las partículas inicialmente. Sin embargo, la densidad de objetivos parciales en `triangle` es mayor en la región inicial y la exploración perjudica la búsqueda en ese caso. Por otro lado, ES tiene una mayor componente de explotación y menor de exploración, lo cual explica que no alcance la cobertura que obtiene PSO en `calday` y `select`.

El uso de la cobertura corregida en la presentación de nuestros experimentos ha permitido eliminar la pérdida debido al código. Sin embargo, podemos observar que los mejores resultados de cobertura en algunos problemas se mantienen por debajo del 100 %. Esto se debe en parte a la pérdida debida al entorno (`select`, por ejemplo, usa memoria dinámica) y a la ineficacia del generador (en `bessel` existe un predicado que comprueba la igualdad entre números reales, que es muy difícil de satisfacer con técnicas de generación dinámica).

Para comparar el número de evaluaciones debemos tener en cuenta que la comparación es justa solamente cuando los algoritmos obtienen la misma cobertura. En otro caso, podemos concluir que el algoritmo que requiere menos evaluaciones es mejor si la cobertura obtenida es mayor. Este es el caso del PSO en los programas `bessel` y `sa` y de ES en los programas `triangle` y `netflow`.

7.7. Resultados previos en la literatura

La tarea de comparar nuestros resultados con los de trabajos previos no es sencilla. Primero, necesitamos encontrar trabajos que aborden los mismos programas. Hay un programa muy popular en el dominio de la generación automática de casos de prueba: el clasificador de triángulos. No obstante, hay diferentes implementaciones en la literatura y el código fuente no se publica normalmente. En este capítulo, usamos la implementación del clasificador de triángulos publicada en [206]. Tenemos otros dos programas en común con [206]: el cálculo del máximo común divisor y la ordenación por inserción. No obstante, usamos implementaciones distintas de estos algoritmos. Para facilitar futuras comparaciones hemos indicado en la Tabla 7.1 cómo conseguir el código fuente de los programas objetivo (disponibles online en `http://tracer.lcc.uma.es/problems/testing/` con la excepción de `netflow`).

Un segundo obstáculo cuando se comparan diferentes técnicas es el de las medidas. Aquí usamos la cobertura de condiciones corregida para medir la calidad de las soluciones. En [151, 244, 246], los autores usan la cobertura de ramas. Por otro lado, la medida de cobertura usada en [206] se obtiene mediante un software propietario: `DeepCover`. No podemos sacar grandes conclusiones de una comparación cuantitativa de todos estos resultados porque se usan medidas distintas. Otro aspecto que puede afectar a los resultados

es el número de ejecuciones independientes. Un número bajo de ejecuciones independientes de algoritmos estocásticos no es suficiente para obtener una idea clara sobre el comportamiento de la técnica usada.

A pesar de todas las consideraciones previas, incluimos en la Tabla 7.13 los mejores resultados de cobertura media obtenidos para el clasificador de triángulos en [151, 206, 244, 246] y el número necesario de evaluaciones (ejecuciones del programa objeto). Mostramos en la misma tabla los resultados de ES, el mejor algoritmo con respecto al porcentaje de cobertura en este capítulo.

Tabla 7.13: Resultados previos de cobertura y número de evaluaciones para `triangle`.

`triangle`	Ref. [206]	Ref. [151]	Ref. [244]	Ref. [246]	ES (aquí)
Cobertura (%)	94.29[b]	100.00[a]	100.00[a]	100.00[a]	100.00[c]
Evaluaciones	≈ 8000.00	18000.00	608.00	3439.00	1207.30

[a]Cobertura de ramas.
[b]Cobertura con `DeepCover`.
[c]Cobertura de condiciones corregida.

Si nos centramos en la cobertura de la Tabla 7.13, nuestra ES obtiene cobertura total junto con los algoritmos de [151, 244, 246]. Comparando el número de evaluaciones, nuestro trabajo se encuentra en segunda posición, detrás del trabajo de Sagarna y Lozano [244]. No obstante, debemos recordar que la cobertura de condiciones (usada en este capítulo) es un criterio de adecuación más severo que la cobertura de ramas usado por los autores en [244] (véase la Sección 3.2), lo cual puede explicar esta diferencia en el número de evaluaciones.

Tenemos también un programa en común con el trabajo de Wegener *et al.* [282]: `netflow`. La comparación se muestra en la Tabla 7.14. Analizando el código descubrimos que nuestro 98.17 % de cobertura corregida de condiciones obtenida con 362.40 evaluaciones se corresponde con el 99 % de cobertura de ramas que ellos obtienen con 40503. Además, esta cobertura de ramas es la máxima cobertura alcanzable, como se indica en [282]. La conclusión es que obtenemos la misma cobertura con, aproximadamente, una centésima parte de las evaluaciones, a pesar del hecho de que usamos una función de *fitness* menos elaborada que la suya.

Tabla 7.14: Resultados previos de cobertura y número de evaluaciones para `netflow`.

`netflow`	Ref. [282]	ES (aquí)
Cobertura (%)	99.00[a]	98.17[b]
Evaluaciones	40503.00	362.40

[a]Cobertura de ramas.
[b]Cobertura de condiciones corregida.

7.8. Conclusiones

En este capítulo hemos analizado, en primer lugar, la aplicación de algoritmos evolutivos con población centralizada y descentralizada al problema de generación de casos de prueba. En concreto, hemos comparado una ES distribuida y un GA distribuido con sus versiones centralizadas. Los resultados muestran

que las versiones descentralizadas no tienen ventaja estadística sobre las centralizadas, ni en términos de cobertura ni en esfuerzo. Esta es una observación inesperada, ya que existen muchos trabajos mostrando un alto grado de precisión del enfoque descentralizado. Para comprobar que las conclusiones obtenidas con respecto al enfoque distribuido no son el resultado de una configuración poco acertada, hemos estudiado la influencia en los resultados de varios parámetros del generador de casos de prueba basado en dES. Estos parámetros han sido el modo de búsqueda, la condición de parada, el número de semillas usadas en las islas y el periodo de migración. Los experimentos muestran que los resultados mejoran si todas las islas persiguen el mismo objetivo parcial con respecto a las versiones que persiguen distintos objetivos. Esto puede ser debido a que los distintos objetivos parciales que cada isla trata conseguir son muy diferentes entre sí y la colaboración entre las islas no ayuda en la búsqueda. Por otro lado, la condición de parada que consiste en detener el algoritmo cuando un nuevo objetivo parcial se cubre (*any*) no tiene una clara ventaja sobre aquélla en la que sólo la cobertura del objetivo parcial perseguido se usa para detener el algoritmo (*obj*). Esto puede indicar que, en realidad, no se cubren con frecuencia objetivos parciales que no sean el perseguido en un instante determinado. Analizando el número de semillas usadas en la población inicial de dES, descubrimos que los mejores resultados se obtienen con una única semilla. La presencia de varias semillas puede repartir la búsqueda que se realiza en cada isla entre regiones distintas del espacio, requiriéndose un mayor número de evaluaciones para conseguir el objetivo parcial. Finalmente, un alto periodo de migración parece beneficiar la búsqueda, lo que confirma que el enfoque distribuido utilizado no beneficia la búsqueda en el problema de generación de casos de prueba para las instancias abordadas en este capítulo.

En una segunda etapa, hemos mostrado los resultados obtenidos con PSO y los hemos comparado con ES, GA y RND. Los resultados muestran que PSO y ES poseen una eficacia similar en general. Mientras que en algunos programas PSO aventaja a ES, en otros, ES obtiene los mejores resultados. De cualquier forma, concluimos que ambos algoritmos resultan mejores que RND y GA, el cual es la base de numerosos trabajos en el dominio. Esto abre una prometedora línea de investigación relativa a la aplicación de PSO y ES en la generación de casos de prueba con algoritmos metaheurísticos, donde GA ha sido hasta ahora un estándar *de facto*. Más aún, el número de parámetros a ajustar en la ES es más bajo que el de los otros algoritmos y, por este motivo, creemos que es más adecuado para las herramientas automáticas que deben ser usadas, en general, por personas sin conocimiento sobre algoritmos evolutivos.

Capítulo 8

Aplicación de ACOhg a la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes

En este capítulo abordamos el problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. Para resolver el problema usamos la metaheurística desarrollada en la Sección 5.3: ACOhg. La organización del capítulo es como sigue. En la siguiente sección describimos los modelos Promela usados en los experimentos. Tras esto, presentamos la configuración utilizada en los experimentos en la Sección 8.2. Posteriormente, se muestran y discuten los resultados obtenidos al estudiar la influencia aislada de λ_{ant} en los resultados (Sección 8.3) y analizar en profundidad las técnicas misionera y de expansión de ACOhg (Sección 8.4). Seguidamente, en la Sección 8.5, comparamos los resultados que ACOhg obtiene con los de algoritmos exactos bien conocidos en el dominio del problema. La Sección 8.6 presenta los resultados de la combinación de ACOhg con reducción de orden parcial (introducida en la Sección 3.3). Por último, comparamos ACOhg con un algoritmo genético aparecido en un trabajo previo en la Sección 8.7 y concluimos el capítulo en la Sección 8.8.

8.1. Conjunto de sistemas concurrentes

Para la realización de los experimentos hemos seleccionado 10 modelos Promela pertenecientes a sistemas concurrentes que violan propiedades de seguridad. Todos estos modelos pueden encontrarse documentados en la literatura [94]. En la Tabla 8.1 presentamos los modelos indicando su longitud en líneas de código (LdC), el número de estados que poseen, el número de procesos, la longitud de una traza de error óptima (más corta) y la propiedad de seguridad que violan. Estos modelos se pueden encontrar junto con el código fuente de HSF-SPIN en `www.albertolluch.com`.

Dentro de estos modelos hay cuatro de ellos escalables: `giop`, `leader`, `marriers` y `phi`. Estos modelos tienen parámetros que permiten generar instancias del modelo tan grandes como se desee. El modelo `giop`ij es una implementación del protocolo Inter-ORB de CORBA para i clientes y j servidores (sólo puede ser 1 o 2) [156]. El siguiente modelo, `leader`i, es un algoritmo de elección de un líder para un anillo unidireccional con i procesos candidatos [67]. El modelo `marriers`i es un protocolo que resuelve el

Tabla 8.1: Modelos Promela usados en los experimentos.

Modelo	LdC	Estados	Procesos	Long. traza óptima	Propiedad violada
`garp`	272	Desconocido	8	16	Ausencia de interbloqueo
`giop`ij	717	Desconocido	$i+3(j+1)$	-	Ausencia de interbloqueo
`leader`i	172	Desconocido	$i+1$	-	Aserto
`lynch`	47	146	4	27	Aserto
`marriers`i	142	Desconocido	$i+1$	-	Ausencia de interbloqueo
`needham`	260	18242	4	5	Fórmula LTL
`phi`j	34	3^j *	$j+1$	$j+1$	Ausencia de interbloqueo
`pots`	453	Desconocido	8	5	Ausencia de interbloqueo
`relay`	70	1062	3	12	Invariante
`x509`	246	4528	7	47	Invariante

* Resultado teórico.

problema de las parejas estables para i parejas [202]. El último de los modelos escalables, `phi`j, implementa el conocido problema de los filósofos de Dijkstra con j filósofos.

El resto de los modelos no son escalables. El modelo `garp` implementa el protocolo de registro de atributos genéricos (*Generic Attribute Registration Protocol*) [218]. `lynch` es una implementación del protocolo de Lynch [184]. El protocolo de autenticación Needham-Schroeder [219] se implementa en `needham`. `pots` modela un sistema de telefonía [155]. El modelo `relay` implementa un circuito electromecánico [275]. El último modelo, `x509`, es el protocolo de autenticación X.509 [154].

Los modelos más pequeños son `lynch`, `needham`, `relay`, y `x509`. Los otros dos modelos no escalables (`garp` y `pots`) tienen asociado un autómata de Büchi que no cabe en la memoria principal de la máquina usada para los experimentos (512 MB). En cuanto a los modelos escalables, las versiones más grandes que caben en la máquina utilizada son `marriers3`, `leader5`, `phi10` y `giop12`. Como ilustración del crecimiento de estos modelos escalables diremos que para el modelo `phi16` se necesitan 11 GB de memoria para almacenar los estados y para el `phi20` son necesarios 1039 GB.

8.2. Parámetros de ACOhg

En los experimentos usamos el algoritmo ACOhg con la configuración mostrada en la Tabla 8.2. Salvo que se diga lo contrario, en las secciones siguientes se usa la técnica misionera. Estos parámetros no se han elegido de forma arbitraria; son el resultado de un estudio previo dirigido a encontrar la mejor configuración para abordar los modelos. Es decir, realizamos un diseño factorial de experimentos usando un conjunto de valores para cada parámetro y seleccionamos la configuración para la que los algoritmos obtenían el mejor compromiso entre eficacia y calidad de la solución. La parte más relevante de este estudio se encuentra en la Sección A.2.

Con respecto a la información heurística, η_{ij}, usamos la expresión $\eta_{ij} = 1/(1 + H_\varphi(j))$ cuando el objetivo es encontrar un contraejemplo de una fórmula LTL (`needham`), un aserto (`leader`i y `lynch`) o un invariante (`relay` y `x509`). En la expresión anterior, $H_\varphi(j)$ es la heurística basada en fórmula evaluada en el estado j. Para el caso de la detección de interbloqueos, usamos $\eta_{ij} = 1/(1+H_{ap}(j))$ donde $H_{ap}(j)$ es la heurística de los procesos activos evaluada en el estado j (véase la Sección 3.3.4). No obstante, también mostramos en las secciones siguientes una versión de ACOhg que no usa información heurística. Cuando

Tabla 8.2: Parámetros de ACOhg.

Parámetro	Valor
Iteraciones	100
colsize	10
λ_{ant}	10
σ_s	2
s	10
ξ	0.5
a	5
ρ	0.2
α	1.0
β	2.0
p_p	1000
p_c	1000

exista posibilidad de confusión llamaremos ACOhg-h a la versión que utiliza información heurística y ACOhg-b a la que no la usa. El criterio de parada de nuestros algoritmos consiste en encontrar una traza de error o alcanzar el número máximo de iteraciones permitidas (100). Este no es el único criterio de parada interesante; el algoritmo podría seguir la búsqueda después de encontrar una traza de error para optimizar su longitud. No obstante, estamos interesados aquí en observar el esfuerzo requerido por el algoritmo para obtener una traza de error, que es un objetivo prioritario en las primeras fases de diseño de software crítico.

Ya que ACOhg es un algoritmo estocástico, necesitamos realizar varias ejecuciones independientes para tener una idea de su comportamiento. Por esto, realizamos 100 ejecuciones independientes para conseguir muy alta confianza estadística. En las tablas de resultados mostramos la media y la desviación estándar de esas 100 ejecuciones. La máquina usada en los experimentos es un Pentium 4 a 2.8 GHz con 512 MB de RAM y sistema operativo Linux (versión del *kernel* 2.4.19-4GB). La cantidad máxima de memoria asignada a los algoritmos es 512 MB (como la cantidad de memoria física de la máquina): cuando un proceso excede esta memoria se detiene automáticamente. Hacemos esto para evitar un gran trasiego de datos desde/hasta la memoria secundaria, que podría afectar significativamente al tiempo requerido en la búsqueda.

8.3. Influencia de λ_{ant}

En esta sección pretendemos estudiar la influencia de λ_{ant} en los resultados. Este es un primer paso para comprender la forma en que trabaja el algoritmo. Hemos usado cuatro instancias del modelo `phi` con 8, 12, 16 y 20 filósofos. Para cada instancia hemos usado distintos valores de λ_{ant} en distintas ejecuciones. Concretamente, se han usado valores que van desde la profundidad del nodo objetivo más cercano d_{opt} (solución óptima) hasta $6d_{opt}$ aumentando en intervalos de $0.5d_{opt}$. Por comodidad en las discusiones siguientes llamaremos κ al *coeficiente de aumento de longitud* calculado como el cociente λ_{ant}/d_{opt}.

En estos experimentos no se ha usado ninguna información heurística para guiar la búsqueda. De esta forma queremos estudiar el comportamiento de ACOhg en ausencia de información (ACOhg-b). El número máximo de iteraciones que realiza el algoritmo se ha reducido de 100 a 10. Se ha tomado este

bajo número de iteraciones porque de esta forma se puede apreciar la influencia de λ_{ant} en la tasa de éxito[1]. Experimentos previos demostraron que usando un número más alto de evaluaciones, la tasa de éxito ascendía al 100 % en todos los casos. No hemos usado la técnica de expansión o misionera. En las Tablas 8.3 y 8.4 se muestra la tasa de éxito (número de ejecuciones que encontraron una trazas de error) y la longitud de las soluciones encontradas, respectivamente.

Tabla 8.3: Tasa de éxito (%) en función de κ.

Número de	Coeficiente de aumento de longitud (κ)										
Filósofos	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0
8	100	100	100	100	100	100	100	100	100	100	100
12	90	100	100	100	100	100	100	100	100	100	100
16	43	95	100	100	100	100	100	100	100	100	100
20	7	52	95	100	100	100	100	100	100	100	100

Tabla 8.4: Longitud de la trazas de error encontrada en función de κ.

Núm.	Coeficiente de aumento de longitud (κ)										
Filós.	1.0	1.5	2.0	2.5	3.0	3.5	4.0	4.5	5.0	5.5	6.0
8	$10.00_{0.00}$	$10.00_{0.00}$	$11.68_{3.26}$	$16.80_{1.83}$	$14.08_{3.71}$	$16.32_{2.89}$	$16.80_{2.86}$	$12.28_{2.90}$	$17.20_{2.40}$	$13.88_{3.17}$	$14.48_{3.50}$
12	$14.00_{0.00}$	$16.04_{2.00}$	$17.52_{1.30}$	$18.00_{0.00}$	$18.00_{0.00}$	$17.72_{1.02}$	$17.76_{0.95}$	$18.00_{0.00}$	$18.00_{0.00}$	$18.00_{0.00}$	$18.00_{0.00}$
16	$18.00_{0.00}$	$21.54_{2.95}$	$25.40_{4.54}$	$25.64_{5.40}$	$32.68_{8.06}$	$34.68_{9.75}$	$30.08_{11.03}$	$38.84_{16.24}$	$47.72_{20.69}$	$51.64_{19.28}$	$60.08_{25.13}$
20	$22.00_{0.00}$	$27.69_{2.64}$	$31.14_{6.71}$	$38.68_{7.27}$	$45.44_{9.91}$	$49.68_{12.26}$	$56.60_{16.93}$	$47.52_{16.98}$	$74.36_{18.81}$	$72.64_{24.35}$	$76.92_{30.94}$

Podemos observar en las últimas filas de la Tabla 8.3 que la tasa de éxito aumenta al permitir a las hormigas recorrer caminos más largos. Esto es debido a que la porción del grafo explorado aumenta con λ_{ant} y así las hormigas encuentran diferentes caminos para llegar al objetivo. Estos caminos serán, en general, más largos que el camino óptimo. Una prueba de ello se muestra en la Tabla 8.4 donde se puede apreciar que la longitud media de las soluciones aumenta, en general, con λ_{ant}. Si nos fijamos en el coeficiente de aumento de longitud necesario para conseguir el 100 % de éxito, observaremos que aumenta de una forma lineal (1.0, 1.5, 2.0 y 2.5) con respecto al número de filósofos. Esto es un resultado importante, ya que el número de estados del sistema concurrente aumenta de forma exponencial. No obstante, son necesarios más experimentos para corroborar dicha hipótesis.

El aumento de la tasa de éxito con κ depende en gran parte del grafo explorado. Para que el aumento en la longitud de las hormigas ayude a la búsqueda es preciso que existan varios caminos que lleguen al nodo (o nodos) objetivo. Si no fuera así, el aumento de λ_{ant} sería inútil. Este es el caso cuando se explora un árbol con un sólo nodo objetivo. En dicho escenario sólo hay un camino desde el nodo inicial al nodo objetivo. Una hormiga que no haya encontrado el nodo objetivo cuando ha pasado por su profundidad nunca más podrá volver a él. Por lo tanto, la forma en que influye λ_{ant} en la tasa de éxito depende del grafo a explorar.

El hecho de que el algoritmo encuentre más fácilmente la traza de error cuando λ_{ant} es mayor, se deja notar también en la memoria y el tiempo empleado para encontrar dicha traza. En las Figuras 8.1 y 8.2 se muestra gráficamente la cantidad de memoria usada (en Kilobytes) y el tiempo medio empleado (en milisegundos). Debemos aclarar que la memoria usada es la cantidad máxima de memoria que requiere

[1]En este problema, una ejecución tiene *éxito* cuando encuentra una traza que lleva a un estado de aceptación, refutando la propiedad de seguridad de ausencia de interbloqueos.

el algoritmo durante su ejecución. Puesto que realizamos 100 ejecuciones independientes, tomamos esos valores máximos de memoria y mostramos la media. Como se puede apreciar, tanto la memoria como el tiempo disminuyen al aumentar λ_{ant}. Esto es debido a que la probabilidad de que una hormiga encuentre un nodo objetivo es más alta y no son necesarias tantas iteraciones del algoritmo. De esta forma, además de reducir el tiempo de cómputo se reduce la cantidad de rastros de feromona almacenados en memoria.

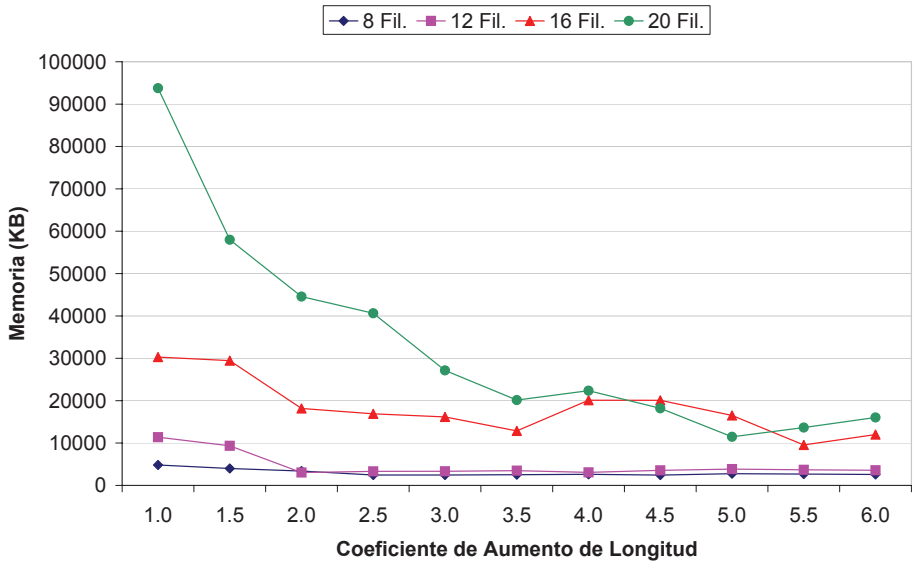


Figura 8.1: Memoria máxima usada por el algoritmo.

8.4. Análisis de las técnicas misionera y de expansión

A continuación nos proponemos investigar cuál de las dos técnicas descritas en la Sección 5.3.2 para poder alcanzar nodos alejados del nodo inicial es más adecuada para el problema de encontrar errores en sistemas concurrentes. Para esto, estudiaremos ambas técnicas por separados en las siguientes secciones y las compararemos en la Sección 8.4.3. Para estos experimentos usamos el modelo `phi20` y utilizamos información heurística (algoritmo ACOhg-h).

8.4.1. Técnica misionera

Para la realización de este experimento fijamos el valor de λ_{ant} a 21, que es la longitud de un camino óptimo en `phi20`. Reducimos el número de pasos del algoritmo de 100 a 20. Los valores de σ_s varían entre 1 y 20. El objetivo de este experimento es observar la influencia en los resultados del parámetro σ_s. Los restantes parámetros son los de la Tabla 8.2. En la Tabla 8.5 podemos ver la tasa de éxito, la longitud de las trazas de error, la cantidad de memoria requerida, el número de estados expandidos y el tiempo.

A partir de los resultados, concluimos que la tasa de éxito es mayor cuando el número de pasos por etapa σ_s es bajo, es decir, cuando se realizan más etapas. Esto se esperaba, ya que de este modo las

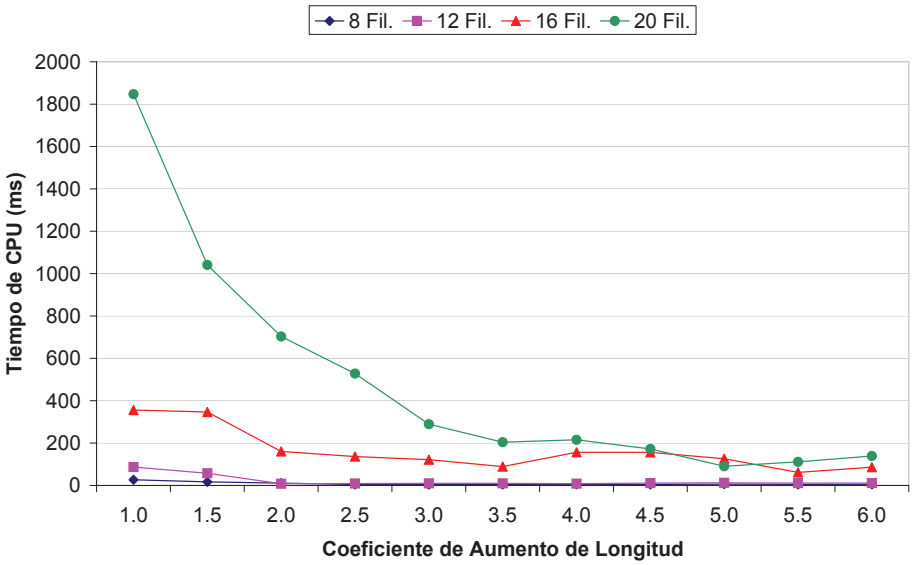


Figura 8.2: Tiempo requerido por el algoritmo.

Tabla 8.5: Análisis de la técnica misionera.

σ_s	Tasa de éxito (%)	Longitud (estados)	Memoria (KB)	Estados exp.	Tiempo (ms)
1	100.00	45.92 14.62	15626.24 449.16	652.50 189.23	383.00 57.37
2	100.00	36.16 8.42	25374.72 1098.88	867.94 214.74	702.80 89.41
3	100.00	32.40 6.68	35215.36 3989.38	1029.48 292.24	1114.50 231.71
4	100.00	31.92 6.09	45936.64 4674.69	1255.85 316.13	1694.10 315.52
5	100.00	30.44 5.56	56166.40 6488.27	1430.79 285.89	2384.50 417.26
6	100.00	29.60 5.32	66242.56 7579.95	1611.84 344.92	3142.60 679.88
7	100.00	29.28 5.89	73749.25 14313.94	1761.05 445.76	3853.80 1152.69
8	100.00	29.12 5.95	82042.88 17677.74	1908.51 490.15	4777.30 1589.97
9	100.00	28.16 5.38	90163.69 22919.01	2042.31 573.35	5930.50 2144.56
10	100.00	27.96 5.59	100341.76 22539.86	2259.97 563.21	6924.90 2493.97
11	100.00	26.72 5.04	108922.88 25936.39	2443.72 633.48	8228.50 3047.22
12	100.00	26.56 4.86	113469.44 35408.40	2526.26 838.53	9354.60 4051.15
13	99.00	26.98 5.43	124834.91 33336.51	2773.32 792.54	11202.32 4687.86
14	100.00	27.24 5.32	133345.28 37115.04	2955.43 872.13	13232.80 5706.19
15	100.00	26.84 5.41	141302.45 40493.07	3122.49 942.98	15046.60 6888.43
16	100.00	27.08 4.98	151788.85 40731.56	3365.07 956.01	18021.80 7774.61
17	99.00	25.93 4.82	149257.38 50491.50	3298.78 1189.23	18431.92 10623.06
18	99.00	26.33 4.65	162816.00 52206.25	3599.38 1215.47	22315.45 11582.47
19	95.00	26.05 4.86	177077.23 45198.72	3902.56 1059.14	26734.53 12853.33
20	27.00	21.00 0.00	114384.59 60693.26	2390.67 1321.36	6434.44 9159.06

hormigas pueden alcanzar nodos más profundos en el grafo y pueden encontrar más caminos que llegan al nodo objetivo. Cuando σ_s va de 1 a 10 (lo que significa que el algoritmo realiza entre 2 y 20 etapas) la tasa de éxito es del 100 %. No obstante, cuando σ_s está en el rango de 11 a 19 no siempre se consigue 100 % de tasa de éxito. En estos casos se realizan dos etapas, pero la segunda es más corta que la primera (menos pasos) y, aunque la tasa de éxito se mantiene muy alta (por encima del 95 %), a veces ACOhg no puede encontrar el nodo objetivo. El gran descenso de la tasa de éxito sucede cuando sólo se realiza una etapa ($\sigma_s = 20$), es decir, cuando la técnica misionera no se usa. Esto significa que el uso de la técnica misionera aumenta la tasa de éxito.

Con respecto a la longitud de las trazas de error, observamos que aumenta cuando se realiza un gran número de etapas (σ_s bajo). Esto es debido a que el uso de un valor bajo en σ_s produce cambios más rápidos de etapas, con lo que el algoritmo explora nodos más profundos en el grafo de construcción y encuentra, con mayor probabilidad, caminos más largos al nodo objetivo. En general, valores bajos de σ_s implican mayor tasa de éxito pero trazas de error más largas. Tenemos que encontrar un compromiso entre eficacia y calidad de la solución.

Si observamos la cantidad de recursos requeridos por el algoritmo, apreciamos que la memoria, los estados expandidos, y el tiempo de CPU aumentan con σ_s (los tests estadísticos apoyan esta observación, véase la Sección B.3). Es decir, como en el caso de la tasa de éxito, se prefieren valores bajos de σ_s para encontrar errores con pocos recursos. En conclusión, desde un punto de vista práctico, podemos afirmar lo siguiente: si se requiere una traza de error rápidamente y/o usando poca cantidad de memoria, entonces se debe usar un valor bajo para σ_s; pero si se prefiere una traza de error corta, se debe asignar a σ_s un valor alto o, incluso, se podría no usar la técnica misionera. Esto es una gran ventaja de ACOhg: ajustando sus parámetros podemos conseguir el comportamiento deseado. Cuando se usan otras técnicas (como los algoritmos exhaustivos cuyos resultados mostraremos en la Sección 8.5) es necesario elegir entre distintas técnicas dependiendo de los requisitos del usuario (el ingeniero a cargo de la calidad del software). En el caso de ACOhg, basta un cambio de parámetros para ajustarse a los nuevos requisitos.

8.4.2. Técnica de expansión

A continuación analizamos la técnica de expansión usando de nuevo el modelo `phi20`. Al parámetro δ_l le asignamos la longitud de una traza óptima, es decir, 21 y asignamos a σ_i valores de 1 a 20. El valor inicial de λ_{ant} es también 21. De este modo, podremos hacer una comparación justa entre las técnicas misionera y de expansión en la siguiente sección, porque la profundidad máxima que los algoritmos alcanzan durante la búsqueda en cada paso es la misma cuando $\sigma_s = \sigma_i$. Como en la sección anterior, el máximo número de pasos que realizan los algoritmos es 20; el resto de parámetros son los de la Tabla 8.2. En la Tabla 8.6 presentamos los resultados en el mismo formato que el usado para la técnica misionera.

A partir de la Tabla 8.6, obtenemos las mismas conclusiones que para la técnica misionera. La tasa de éxito aumenta cuando σ_i es bajo, es decir, cuando λ_{ant} aumenta frecuentemente. La memoria requerida, los estados expandidos y el tiempo de CPU toman los valores más bajos también cuando σ_i es bajo. No obstante, un valor bajo de σ_i significa que se exploran caminos más largos muy pronto en la búsqueda y, por tanto, se encuentran trazas de error de mayor longitud (lejos del óptimo). La influencia de σ_i en los resultados es similar a la influencia de σ_s usando la técnica misionera, y podemos encontrar explicaciones similares para esta influencia.

Tabla 8.6: Análisis de la técnica de expansión.

σ_i	Tasa de éxito (%)	Longitud (estados)	Memoria (KB)	Estados exp.	Tiempo (ms)
1	100.00	36.32 9.14	46571.52 16796.40	1131.39 487.05	919.90 492.38
2	100.00	33.36 7.70	63610.88 21670.26	1490.61 583.73	1782.70 907.38
3	100.00	31.48 7.75	70338.56 27169.29	1605.40 711.17	2309.60 1376.49
4	100.00	31.84 7.16	88576.00 31982.10	2018.49 835.09	3423.10 2041.52
5	100.00	31.84 7.00	89640.96 31630.30	1996.46 800.46	3521.10 2016.99
6	100.00	29.44 6.74	103761.92 43615.20	2327.61 1102.30	4765.70 3220.69
7	100.00	31.08 6.94	110695.33 40902.88	2458.04 1012.00	5149.80 2956.62
8	100.00	30.04 6.99	116736.00 46757.91	2590.29 1174.00	5937.00 4122.49
9	100.00	28.84 6.76	123444.37 43519.17	2715.92 1068.11	6350.80 3585.08
10	99.00	28.80 6.99	142449.78 51958.73	3173.07 1289.45	8207.47 5036.16
11	97.00	28.01 6.62	137511.59 47092.28	3007.11 1109.13	7665.05 3757.01
12	99.00	28.72 6.78	150641.78 41769.62	3297.68 1005.78	8863.94 3852.30
13	97.00	29.08 7.36	153821.69 51457.41	3362.97 1206.19	9406.39 4447.12
14	94.00	28.15 6.60	166193.02 54359.66	3656.00 1290.20	10874.68 5110.09
15	94.00	27.77 6.75	158077.28 58818.82	3437.20 1360.49	10197.66 5383.95
16	89.00	26.89 6.38	162850.52 60625.07	3532.03 1385.09	10774.38 5412.35
17	83.00	28.66 7.48	173315.08 60555.18	3765.87 1376.22	11957.59 5473.64
18	61.00	26.11 6.72	155698.36 66906.18	3343.66 1500.63	10628.36 6347.18
19	50.00	27.08 7.21	169226.24 56057.92	3617.12 1257.65	11526.40 5598.17
20	31.00	21.00 0.00	126315.35 51566.37	2641.42 1130.68	7030.97 4640.26

8.4.3. Comparación de ambas técnicas

Ahora, comparamos las dos técnicas. Por motivos de claridad, presentamos en las Figuras 8.3, 8.4, 8.5, 8.6 y 8.7 la tasa de éxito, la longitud de las trazas de error, la memoria usada, el número de estados expandidos y el tiempo de CPU requerido por las dos técnicas. La información de estas figuras es la mostrada en las Tablas 8.5 y 8.6.

En la Figura 8.3 podemos observar que ambas técnicas consiguen 100 % de éxito cuando $\sigma_i = \sigma_s$ está por debajo de 10. No obstante, para valores mayores, cuando sólo hay dos etapas en la técnica misionera, y un cambio de λ_{ant} en la técnica de expansión, la eficacia de la primera es mayor. Es decir, la tasa de éxito de la técnica misionera se mantiene por encima del 95 %, mientras que la técnica de expansión alcanza el 50 % de tasa de éxito cuando $\sigma_i = 19$. Esto se puede deber al hecho de que la técnica misionera se centra en una nueva región del grafo de construcción y no explora de nuevo la región de la primera etapa (como hace la técnica de expansión), que podría llevar al algoritmo a tomar un mayor número de caminos equivocados.

Analizando la longitud media de las trazas de error encontradas (Figura 8.4), observamos que las dos técnicas obtienen resultados similares. Se aprecia sólo una ventaja leve de la técnica de expansión para valores bajos de σ_i. En la técnica de expansión, las hormigas parten siempre del nodo inicial y, por lo tanto, la probabilidad de encontrar una traza de error óptima es más alta. En la técnica misionera, las diferentes etapas exploran una región de tamaño similar que se encuentra a diferentes profundidades (las hormigas comienzan la construcción en diferentes nodos) y es más probable encontrar trazas de error más largas. No obstante, este comportamiento sólo se puede apreciar para valores bajos de σ_i y σ_s.

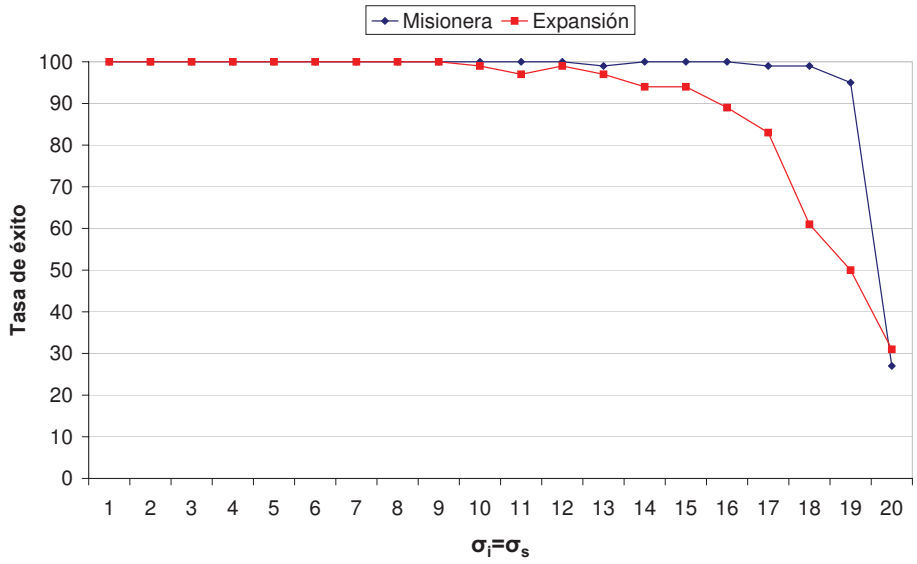


Figura 8.3: Comparación entre las técnicas misionera y de expansión: Tasa de éxito.

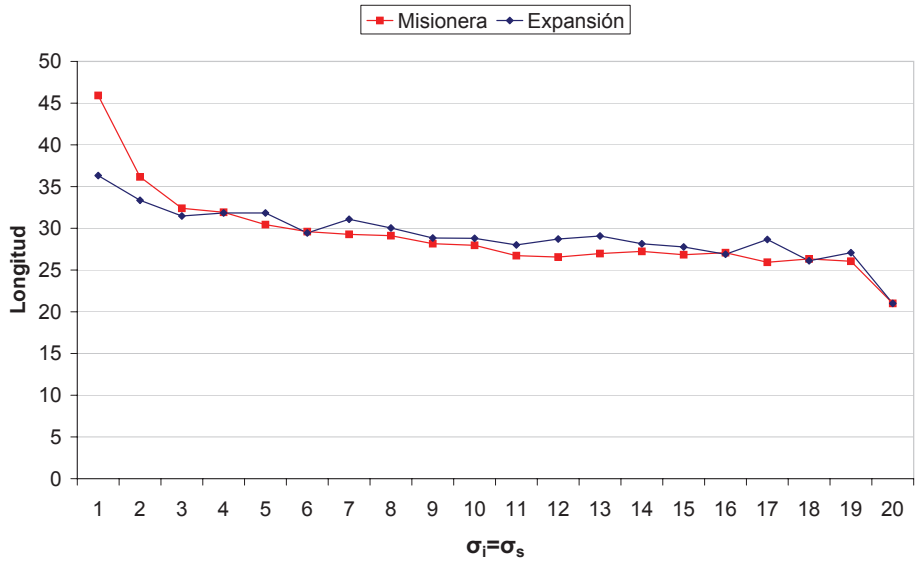


Figura 8.4: Comparación entre las técnicas misionera y de expansión: Longitud de las trazas de error.

Centrémonos ahora en los recursos (Figuras 8.5 y 8.7) y en los estados expandidos (Figura 8.6). La técnica de expansión requiere más memoria que la misionera. Una primera explicación para este hecho es que la técnica de expansión trabaja con caminos más largos. La segunda razón principal es que en

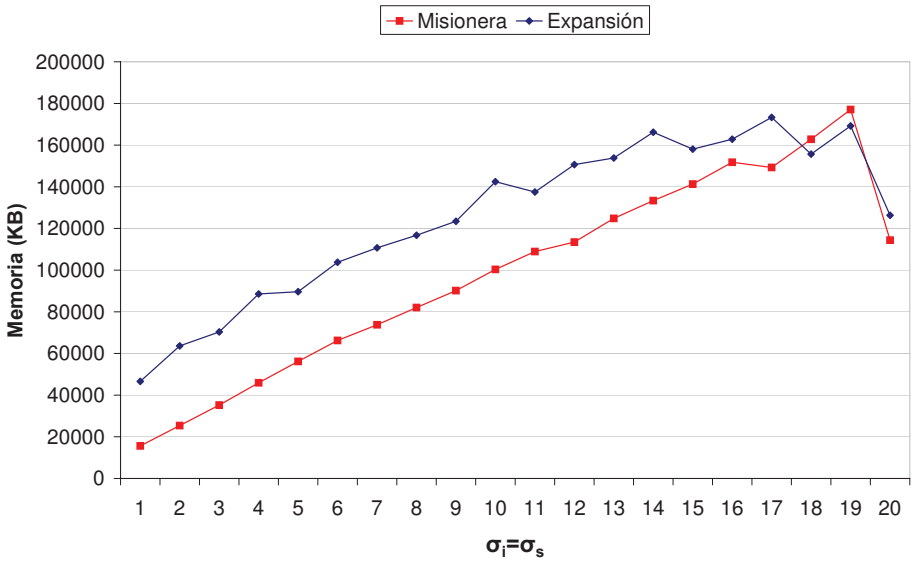


Figura 8.5: Comparación entre las técnicas misionera y de expansión: Memoria usada.

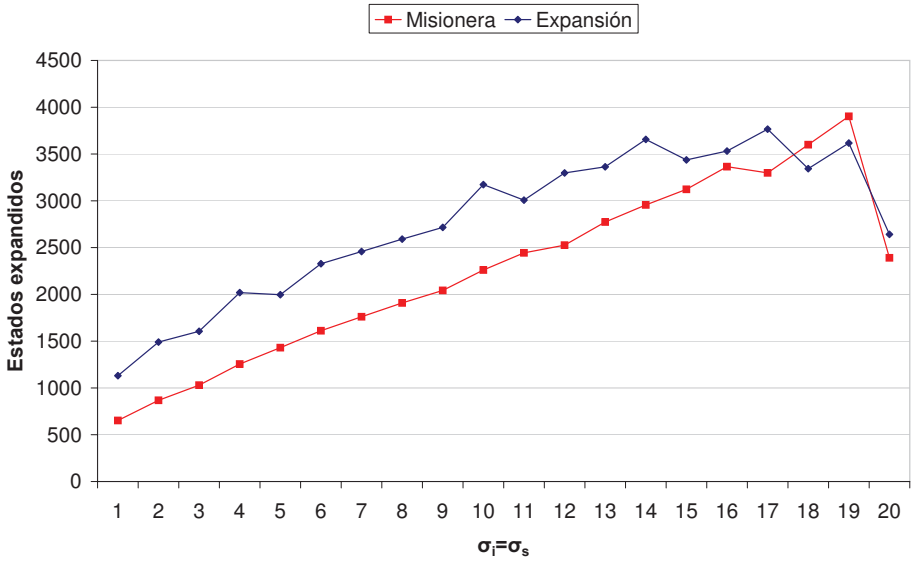


Figura 8.6: Comparación entre las técnicas misionera y de expansión: Estados expandidos.

la técnica misionera los rastros de feromona se descartan al final de cada etapa. Por esto, concluimos que la técnica misionera es más eficiente que la de expansión con respecto al consumo de memoria. Comparando las Figuras 8.5 y 8.6 podemos observar que existe una relación directa entre los estados

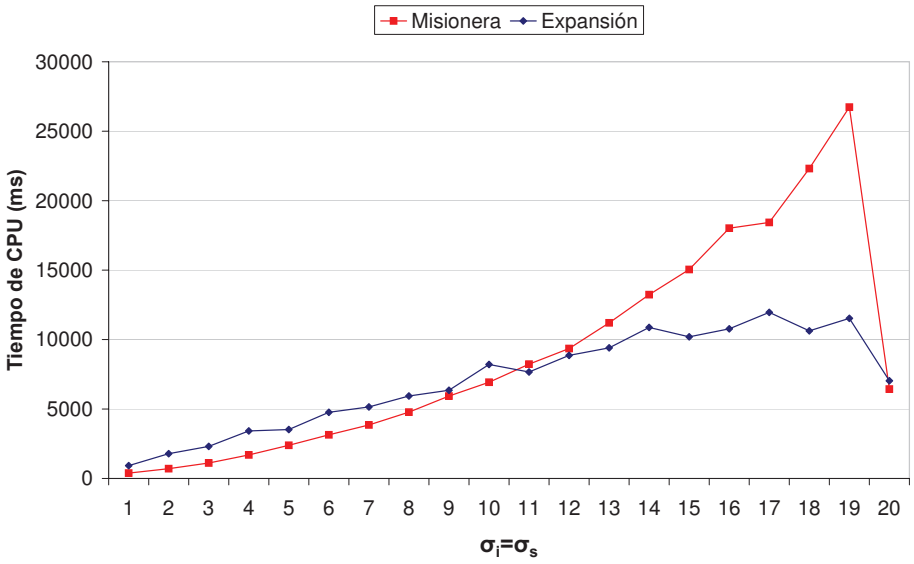


Figura 8.7: Comparación entre las técnicas misionera y de expansión: Tiempo de CPU requerido.

expandidos y la memoria requerida (las curvas son similares en ambas figuras). En el algoritmo ACOhg-h las principales fuentes de consumo de memoria son los estados expandidos y los rastros de feromona. Además, hay un rastro de feromona τ_{ij} asociado a cada arco del grafo de construcción, así que si hay n estados en la memoria podríamos esperar hasta n^2 rastros de feromona. No obstante, usando el modelo `phi20` observamos que la memoria requerida es proporcional al número de estados expandidos. La razón es que el número de arcos que salen de un nodo está acotado por el número de filósofos, es decir, 20; y, por esto, el número de arcos del grafo de construcción es proporcional al número de estados.

Si observamos el tiempo de ejecución requerido, encontramos un comportamiento diferente en ambas técnicas. Para valores bajos de σ_i la técnica de expansión requiere más tiempo que la misionera. No obstante, desde $\sigma_i = \sigma_s = 11$ esta tendencia cambia: la técnica misionera requiere más tiempo de CPU que la de expansión. La razón principal para este comportamiento es la caída en la tasa de éxito de la técnica de expansión. Los valores medios mostrados en las figuras consideran sólo las ejecuciones en las que se encuentra una traza de error. La reducción de la tasa de éxito en la técnica de expansión para valores altos de σ_i significa que algunas ejecuciones requieren más pasos que el máximo permitido (20) para conseguir una traza de error. Estas ejecuciones sin éxito no se están teniendo en cuenta para calcular los valores medios y, por esta razón, obtenemos un valor bajo e irreal para los recursos requeridos. Sólo cuando la tasa es del 100 % los valores medios de los recursos son fiables. Esto explica el gran salto en los valores medios de memoria, estados expandidos y tiempo de CPU cuando $\sigma_i = \sigma_s = 20$.

La conclusión final de este experimento es que la técnica misionera con valores intermedios para σ_s es la mejor opción para la búsqueda. Obtiene resultados de buena calidad con alta eficacia y eficiencia.

8.5. Comparación entre ACOhg y algoritmos exhaustivos

En esta sección comparamos los resultados de ACOhg con los de algoritmos exhaustivos encontrados previamente en la literatura especializada. Estos algoritmos son BFS, DFS, A* y BF (véase la Sección 3.3.6). BFS y DFS no usan información heurística mientras que los otros dos sí. Para hacer una comparación equilibrada usamos dos versiones de ACOhg: una que no usa información heurística (ACOhg-b) y otra que la usa (ACOhg-h). Comparamos ACOhg-b con BFS y DFS en la Tabla 8.7, y ACOhg-h con A* y BF en la Tabla 8.8. En las tablas podemos ver la tasa de éxito, la memoria requerida (en Kilobytes), el número de estados expandidos y el tiempo requerido (en milisegundos) por cada algoritmo. En el caso de los algoritmos ACOhg presentamos medias y desviaciones de las 100 ejecuciones independientes. Para los algoritmos exhaustivos sólo presentamos el resultado de una ejecución, por ser deterministas.

La primera observación que podemos hacer de los resultados de la Tabla 8.7 es que ACOhg-b es el único algoritmo capaz de encontrar una traza de error en todos los modelos. DFS falla en `marriers4` y `phi16`, mientras que BFS falla en esos dos modelos y en `giop22`. La razón para esta pérdida de eficacia en estos algoritmos exactos es que la memoria requerida por ellos excede a la disponible en la máquina. Es más, ACOhg-b encuentra una traza de error en todas las ejecuciones independientes (tasa de éxito del 100 %) en 6 de los 10 modelos. En vista de estos resultados podemos afirmar que ACOhg-b es mejor que DFS y BFS en la tarea de buscar violaciones de propiedades de seguridad en sistemas concurrentes.

Con respecto a la calidad de las soluciones (la longitud de las trazas de error), observamos que ACOhg-b obtiene trazas de error casi óptimas. La longitud óptima de las trazas de error es aquélla de las trazas obtenidas por BFS (cuando encuentra alguna), ya que está diseñado para obtener trazas óptimas. La longitud de las trazas encontradas por DFS es mucho mayor (mala calidad) que la de las obtenidas por los otros dos algoritmos (BFS y ACOhg-b). La única excepción es la del modelo `leader6` (la diferencia es estadísticamente significativa, véase la Tabla B.5).

Discutamos ahora los recursos computacionales usados por los algoritmos. Con respecto a la memoria usada, ACOhg-b requiere menos memoria que BFS en todos los modelos. En algunos modelos la diferencia es muy grande. Por ejemplo, en `garp`, el algoritmo BFS requiere 115 veces la memoria usada por ACOhg-b (y 1000 veces su tiempo de CPU). Es decir, incluso si ACOhg-b se ejecutase 5 veces (que es el número de ejecuciones requerido en media para encontrar una traza de error), una simple ejecución de BFS es más lenta y necesita más memoria, mientras que la reducción en la longitud de la traza de error es insignificante (2 estados). La memoria usada por ACOhg-b es también menor que la requerida por DFS en 5 de los 10 modelos. De los restantes 5 modelos, la diferencia entre DFS y ACOhg-b en `garp`, `giop22`, `lynch`, y `x509` es pequeña. Por tanto, podemos afirmar que ACOhg-b requiere una baja cantidad de memoria que es similar, y a veces menor que la usada por DFS.

Observando el tiempo de CPU requerido por los algoritmos podemos notar que BFS es el algoritmo más lento en 8 de los 10 modelos. Las únicas excepciones son las de los modelos `lynch` y `relay`, dos de los modelos más pequeños. Esto sugiere que para los modelos pequeños los algoritmos exhaustivos son capaces de encontrar errores con pocos recursos computacionales. La aplicación de ACOhg-b es interesante cuando hay que trabajar con modelos grandes (con ese propósito fue diseñado). ACOhg-b es 72 veces más rápido que DFS en `needham`. No obstante, esta es una situación excepcional: en los restantes modelos DFS es más rápido que ACOhg-b cuando encuentra una solución.

En general, podemos afirmar que ACOhg-b es un algoritmo robusto que es capaz de encontrar errores en todos los modelos propuestos. Además, combina las dos buenas características de BFS y DFS: obtiene trazas de error cortas, como BFS, y al mismo tiempo requiere una cantidad reducida de recursos computacionales (memoria y tiempo de CPU), como DFS.

Tabla 8.7: Resultados de los algoritmos sin información heurística.

Modelos	Medidas	BFS	DFS	ACOhg-b
	Tasa de éxito	1/1	1/1	19/100
	Long. (estados)	**16.00**	64.00	18.05 3.12
garp	Mem. (KB)	480256.00	**3357.00**	4142.26 383.09
	Estados exp.	56442.00	**65.00**	419.53 95.07
	tiempo (ms)	72230.00	**10.00**	73.68 14.59
	Tasa de éxito	0/1	1/1	100/100
	Long. (estados)	-	112.00	**45.80** 3.84
giop22	Mem. (KB)	-	**3945.00**	4814.12 328.85
	Estados exp.	-	**220.00**	1048.52 87.18
	tiempo (ms)	-	**30.00**	113.60 12.77
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**33.00**	36.00	50.79 3.18
leader6	Mem. (KB)	337920.00	**2365.00**	7057.52 173.41
	Estados exp.	75428.00	**36.00**	1116.00 89.41
	tiempo (ms)	3400.00	**10.00**	234.80 25.24
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**27.00**	44.00	27.22 0.41
lynch	Mem. (KB)	2149.00	**2001.00**	2136.04 21.16
	Estados exp.	77.00	**48.00**	554.48 10.58
	tiempo (ms)	10.00	**0.00**	16.10 4.88
	Tasa de éxito	0/1	0/1	57/100
	Long. (estados)	-	-	**92.18** 10.31
marriers4	Mem. (KB)	-	-	**5917.91** 157.60
	Estados exp.	-	-	**2045.84** 213.62
	tiempo (ms)	-	-	**257.19** 30.99
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**5.00**	11.00	6.39 1.00
needham	Mem. (KB)	23552.00	62464.00	**5026.36** 828.99
	Estados exp.	1141.00	11203.00	**100.21** 36.19
	tiempo (ms)	1110.00	18880.00	**262.00** 115.47
	Tasa de éxito	0/1	0/1	100/100
	Long. (estados)	-	-	**31.44** 9.43
phi16	Mem. (KB)	-	-	**10905.60** 488.42
	Estados exp.	-	-	**832.08** 293.50
	tiempo (ms)	-	-	**289.40** 63.37
	Tasa de éxito	1/1	1/1	49/100
	Long. (estados)	**5.00**	14.00	5.73 0.63
pots	Mem. (KB)	57344.00	12288.00	**9304.67** 3320.67
	Estados exp.	2037.00	1966.00	**176.47** 66.95
	tiempo (ms)	4190.00	**140.00**	441.63 321.86
	Tasa de éxito	1/1	1/1	98/100
	Long. (estados)	**12.00**	188.00	15.04 3.30
relay	Mem. (KB)	6313.00	7333.00	**3763.49** 287.54
	Estados exp.	915.00	**339.00**	1297.48 862.49
	tiempo (ms)	30.00	**20.00**	961.22 673.27
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**47.00**	52.00	47.65 0.68
x509	Mem. (KB)	20480.00	**2297.00**	3638.08 185.36
	Estados exp.	4501.00	**52.00**	921.11 23.11
	tiempo (ms)	110.00	**0.00**	52.10 4.96

Centrémonos en los resultados de la Tabla 8.8 (algoritmos que usan información heurística). De nuevo, ACOhg-h es capaz de encontrar trazas de error en todos los modelos. Observamos que A* no puede encontrar ninguna traza de error en `marriers4`. Podemos afirmar (comparando las Tablas 8.7 y 8.8) que A* y BF mejoran los resultados de BFS y DFS: A* y BF pueden encontrar errores en casi todos los modelos, obteniendo trazas de error más cortas y usando menos recursos que DFS y BFS. La razón es el uso de información heurística en A* y BF. Podemos decir lo mismo para los algoritmos ACOhg: ACOhg-h obtiene una tasa de éxito mayor que ACOhg-b debido a la información heurística (podemos

Tabla 8.8: Resultados de los algoritmos que usan información heurística.

Modelos	Medidas	A*	BF	ACOhg-h
	Tasa de éxito	1/1	1/1	28/100
	Long. (estados)	**16.00**	46.00	18.04 2.74
garp	Mem. (KB)	159744.00	**3593.00**	3883.00 408.52
	Estados exp.	16800.00	**240.00**	423.86 79.61
	Tiempo (ms)	3380.00	**10.00**	71.79 15.13
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**44.00**	**44.00**	44.20 1.40
giop22	Mem. (KB)	417792.00	**2873.00**	4482.12 259.13
	Estados exp.	83758.00	**168.00**	1001.78 38.98
	Tiempo (ms)	46440.00	**10.00**	112.40 8.14
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**37.00**	**37.00**	49.18 3.51
leader6	Mem. (KB)	132096.00	**2573.00**	6963.48 176.97
	Estados exp.	21332.00	**37.00**	1077.14 85.20
	Tiempo (ms)	1250.00	**0.00**	222.30 24.12
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**27.00**	**27.00**	27.16 0.37
lynch	Mem. (KB)	2117.00	**1981.00**	2137.64 18.41
	Estados exp.	70.00	**46.00**	556.76 21.45
	Tiempo (ms)	**0.00**	10.00	16.30 4.83
	Tasa de éxito	0/1	1/1	84/100
	Long. (estados)	-	108.00	**86.65** 7.66
marriers4	Mem. (KB)	-	41980.00	**5811.43** 138.17
	Estados exp.	-	9193.00	**1915.30** 153.38
	Tiempo (ms)	-	**190.00**	233.33 22.33
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**5.00**	10.00	6.12 1.06
needham	Mem. (KB)	19456.00	**4149.00**	4865.40 1121.12
	Estados exp.	814.00	**12.00**	87.47 45.14
	Tiempo (ms)	810.00	**20.00**	229.50 136.03
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**17.00**	81.00	23.08 5.38
phi16	Mem. (KB)	**2881.00**	10240.00	10680.32 506.96
	Estados exp.	**33.00**	893.00	587.53 141.66
	Tiempo (ms)	**10.00**	40.00	243.80 27.23
	Tasa de éxito	1/1	1/1	99/100
	Long. (estados)	**5.00**	7.00	5.44 0.76
pots	Mem. (KB)	57344.00	**6389.00**	6974.56 3148.77
	Estados exp.	1257.00	695.00	**110.48** 61.91
	Tiempo (ms)	6640.00	**50.00**	319.49 275.03
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**12.00**	29.00	14.12 2.82
relay	Mem. (KB)	6017.00	**2581.00**	3335.24 337.79
	Estados exp.	898.00	**49.00**	481.33 282.09
	Tiempo (ms)	20.00	**10.00**	300.10 208.54
	Tasa de éxito	1/1	1/1	100/100
	Long. (estados)	**47.00**	52.00	47.60 0.65
x509	Mem. (KB)	20480.00	**2433.00**	3500.04 192.44
	Estados exp.	4494.00	**112.00**	908.53 29.92
	Tiempo (ms)	110.00	**10.00**	50.90 5.12

apreciar esto en los modelos `garp`, `mariers4`, `pots`, y `relay`, ya que para los restantes modelos se obtiene un 100 % de éxito con ambos algoritmos). Por esto, podemos afirmar que la información heurística tiene una influencia neta positiva en la búsqueda. Por esta razón los *model checkers* del estado del arte incluyen búsqueda heurística en la práctica [117].

Con respecto a la calidad de la solución, observamos en la Tabla 8.8 que ACOhg-h obtiene trazas de error casi óptimas, similares a las de A*, normalmente mejorando las obtenidas por BF[2].

[2]Comparando las Tablas 8.7 y 8.8 podemos observar que A* no es capaz de encontrar una traza de error óptima en `leader6`. Esto es debido al modo en que se detectan las violaciones de asertos en HSF-SPIN.

Si nos centramos en la memoria requerida por los algoritmos, ACOhg-h normalmente requiere menos memoria que A* (excepto para `phi16` y, en menor medida, `lynch`) pero más que BF (excepto para `marriers4`). Por otro lado, ACOhg-h expande menos estados que A* y BF en 2 de los 10 modelos. Existe una relación entre los estados expandidos en A* y BF y la memoria requerida: cuanto más estados expandidos, mayor es el número de estados almacenados en memoria principal. Puesto que los estados almacenados en memoria son la principal fuente de consumo de memoria, esperamos que la memoria requerida por A* y BF sea mayor conforme más estados expandan. Podemos apreciar este hecho en la Tabla 8.8. No obstante, esta afirmación no se mantiene necesariamente para ACOhg-h, ya que en este último algoritmo hay otra fuente importante de consumo de memoria: los rastros de feromona. Por esta razón podemos observar que BF requiere menos memoria que ACOhg-h en `phi16` y `pots` a pesar de que BF expande más estados (en media) que ACOhg-h.

En general, podemos afirmar que ACOhg-h es el mejor compromiso entre calidad de la solución y memoria requerida: consigue soluciones casi óptimas con una baja cantidad de memoria.

Para resumir los resultados discutidos en esta sección, presentamos en la Figura 8.8 la calidad de las soluciones (longitud de las trazas de error) frente a la memoria requerida por todos los algoritmos en todos los modelos. Puesto que modelos diferentes tienen diferentes longitudes óptimas y diferentes requisitos de memoria, representamos en la figura valores normalizados de la longitud y la memoria. De este modo, podemos mantener todos los puntos en la misma gráfica y podemos comparar los algoritmos globalmente (sin restringir la discusión a un modelo específico). Para cada modelo dividimos la longitud de las trazas de error obtenidas por la longitud mínima obtenida por algún algoritmo para ese modelo. La misma normalización se aplica a la memoria.

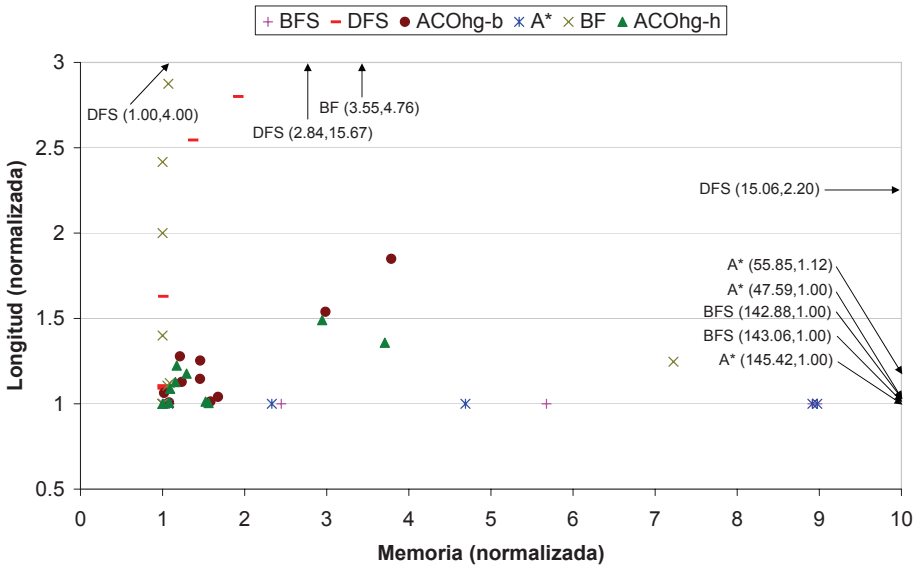


Figura 8.8: Longitud normalizada de las trazas de error frente a la memoria normalizada requerida por todos los algoritmos en todos los modelos.

Observamos en la Figura 8.8 que los resultados de los algoritmos ACOhg se concentran en la esquina inferior izquierda de la gráfica. Es decir, los algoritmos ACOhg son capaces de obtener trazas de error más

cortas (soluciones de buena calidad) con poca memoria. Es más, los algoritmos ACOhg son los únicos que mantienen todos los puntos dentro de una región de buena calidad (inferior izquierda). Los puntos de los otros algoritmos están dispersos en la gráfica. De hecho, hay tres puntos de A*, tres de DFS, dos de BFS y uno de BF que no aparecen en la gráfica porque están fuera del área mostrada en la figura (se han indicado sus coordenadas y se ha señalado con una flecha la región donde deben encontrarse). Esto significa que los algoritmos ACOhg son los algoritmos más robustos de los experimentos. Tienen un comportamiento similar para todos los modelos. Por el contrario, el comportamiento de los algoritmos exhaustivos depende en gran medida del modelo que resuelven. Además, podemos identificar dos comportamientos dentro de los algoritmos exhaustivos: los que requieren mucha memoria pero encuentran soluciones óptimas (BFS y A*) y los que pueden trabajar con poca memoria pero ofrecen soluciones de muy baja calidad (DFS y BF). ACOhg posee las ventajas de estas dos categorías. En vista de este hecho, señalamos los algoritmos ACOhg como técnicas muy prometedoras para encontrar errores en sistemas concurrentes.

8.6. Combinación de ACOhg y reducción de orden parcial

En esta sección combinamos ACOhg-h con reducción de orden parcial, POR (véase la Sección 3.3.5 para más detalles). Con estos experimentos queremos mostrar que ACOhg es compatible con esta técnica y puede beneficiarse de ella. Los parámetros usados son los de la Tabla 8.2, con la excepción de λ_{ant} que toma valor 20, σ_s que toma valor 4 y el número total de pasos que es 1000. Con esta configuración el algoritmo es capaz de alcanzar una mayor profundidad que usando los valores originales de la Tabla 8.2, aumentando la probabilidad de encontrar caminos alternativos al estado de aceptación.

Hemos escogido para este experimento los modelos escalables `giop`, `leader`, y `marriers` porque su autómata de Büchi se puede reducir con POR, tal y como se muestra en [181]. Usamos tres instancias de cada modelo escalable (pequeña, mediana y grande). En la Tabla 8.9 se presentan los resultados. La última columna indica si las diferencias en los valores medios son significativas o no (los tests estadísticos se realizaron siguiendo el procedimiento descrito en el Apéndice B).

En todos los casos se obtuvo un 100 % de tasa de éxito, lo cual resulta bastante positivo si tenemos en cuenta el tamaño de los modelos. Podemos mencionar, por ejemplo, que ACOhg-h es capaz de encontrar el estado de interbloqueo de `marriers20` en todas las ejecuciones, cuando en los experimentos de la Sección 8.5 tan sólo conseguía encontrarlo en el 84 % de los casos para `marriers4`. El motivo de esta mejora en la eficacia del algoritmo se debe a la nueva configuración que, como mencionamos anteriormente, aumenta la probabilidad de encontrar caminos que lleven al estado de aceptación en el autómata de Büchi.

8.6.1. Recursos computacionales

La primera observación de los resultados es que ACOhg-h+POR requiere en todos los modelos menos recursos computacionales (memoria y tiempo de CPU) que ACOhg-h. El test estadístico (con nivel de significancia $\alpha = 0.05$) muestra que todas las diferencias relacionadas con los recursos son significativas. Por esto, podemos afirmar después de estos experimentos que ACOhg-h+POR mejora el rendimiento de ACOhg-h, lo que confirma nuestras expectativas. La explicación es que el grafo a explorar es más pequeño y se tienen que almacenar menos estados y rastros de feromona en memoria. En la Figura 8.9 podemos ver claramente la ventaja de ACOhg-h+POR frente a ACOhg-h con respecto a la memoria requerida. La diferencia entre ambos algoritmos es mayor para los modelos `leader` y `marriers`. Hemos dibujado también una línea para cada modelo escalable indicando cómo aumenta la cantidad de memoria requerida con el parámetro del modelo. Podemos observar que en `leader` y `marriers` el crecimiento es

Tabla 8.9: Resultados de ACOhg-h y ACOhg-h+POR (marcamos los mejores resultados en negrita).

Modelos	Medidas	ACOhg-h		ACOhg-h+POR		Test Estad.
giop21	Long. (estados)	42.30	1.71	**42.10**	0.99	-
	Mem. (KB)	3428.44	134.95	**2979.48**	98.33	+
	Estados exp.	1844.10	29.39	**1831.64**	26.96	+
	Tiempo (ms)	202.00	9.06	**162.50**	5.55	+
giop41	Long. (estados)	70.21	7.56	**59.76**	5.79	+
	Mem. (KB)	9523.67	331.76	**7420.08**	422.94	+
	Estados exp.	2663.91	325.19	**2347.94**	363.91	+
	Tiempo (ms)	354.50	42.39	**264.90**	40.46	+
giop61	Long. (estados)	67.59	13.43	**61.74**	3.16	+
	Mem. (KB)	11970.56	473.59	**11591.68**	477.67	+
	Estados exp.	2603.47	597.36	**2398.55**	378.58	+
	Tiempo (ms)	440.60	71.02	**391.70**	43.86	+
leader6	Long. (estados)	**50.90**	4.52	56.36	3.04	+
	Mem. (KB)	16005.12	494.39	**3710.64**	410.29	+
	Estados exp.	**1894.28**	22.38	1955.23	82.64	+
	Tiempo (ms)	494.00	21.12	**98.80**	8.16	+
leader8	Long. (estados)	**60.83**	4.66	74.11	4.51	+
	Mem. (KB)	24381.44	515.98	**4831.40**	114.10	+
	Estados exp.	**2344.63**	320.90	2749.75	12.29	+
	Tiempo (ms)	1061.20	211.47	**198.90**	4.67	+
leader10	Long. (estados)	**73.84**	4.79	80.86	6.36	+
	Mem. (KB)	30167.04	586.82	**7178.05**	2225.78	+
	Estados exp.	**2764.42**	53.06	3114.22	315.07	+
	Tiempo (ms)	1910.70	45.02	**294.90**	66.96	+
marriers10	Long. (estados)	307.11	34.87	**233.19**	21.91	+
	Mem. (KB)	34170.88	494.39	**18319.36**	804.93	+
	Estados exp.	12667.11	1420.18	**9614.15**	1032.06	+
	Tiempo (ms)	8847.00	634.06	**1306.60**	126.56	+
marriers15	Long. (estados)	540.41	60.88	**395.10**	40.07	+
	Mem. (KB)	51148.80	223.18	**26050.56**	1256.81	+
	Estados exp.	22506.36	2526.52	**16458.42**	1671.93	+
	Tiempo (ms)	19740.50	1935.54	**3595.00**	316.59	+
marriers20	Long. (estados)	793.62	80.45	**569.99**	54.63	+
	Mem. (KB)	68003.84	503.64	**33351.68**	1442.75	+
	Estados exp.	33108.85	3364.88	**23747.43**	2309.39	+
	Tiempo (ms)	49446.30	7557.40	**8174.00**	707.71	+

casi lineal. Esto es un resultado muy prometedor, ya que el número de estados del autómata de Büchi crece normalmente de forma exponencial cuando los parámetros del modelo aumentan linealmente. Esto significa que, incluso con un crecimiento exponencial en el tamaño del autómata de Büchi, la memoria requerida por ACOhg-h y ACOhg-h+POR crece de forma lineal.

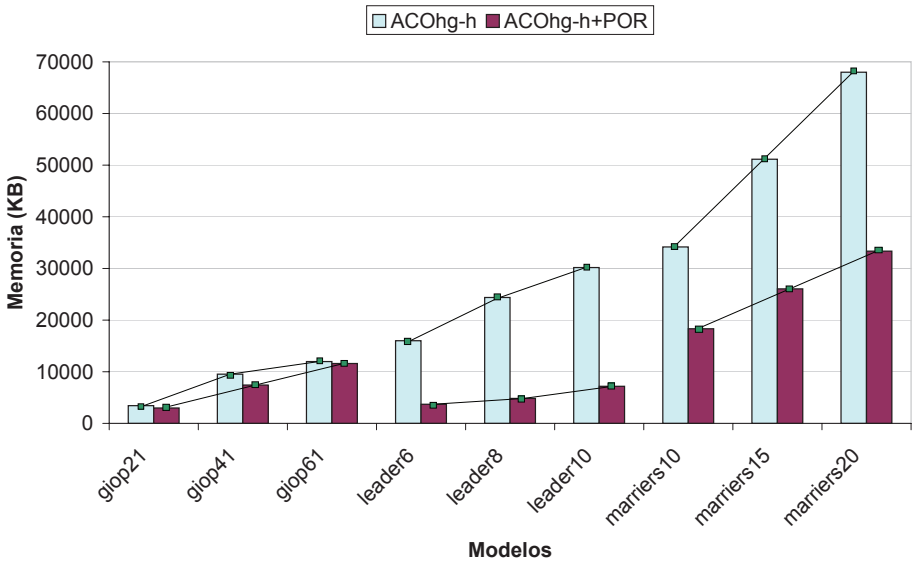


Figura 8.9: Memoria requerida por ACOhg-h y ACOhg-h+POR en los modelos.

8.6.2. Estados expandidos

También apreciamos en la Tabla 8.9 que ACOhg-h+POR no siempre expande menos estados que ACOhg-h. La razón para este comportamiento es que ACOhg-h y ACOhg-h+POR no mantienen en memoria todos los estados generados; por el contrario, la mayoría de ellos son eliminados. Cuando una hormiga construye su camino en la fase de construcción, el último nodo del camino actual se expande, la hormiga selecciona un nodo sucesor y el resto de nodos son eliminados de memoria. En resumen, en cada movimiento de una hormiga se expande un estado, independientemente del tamaño del autómata de Büchi. Es más, el número de movimientos de hormigas en un paso es aproximadamente el tamaño de la colonia (*colsize*) multiplicado por la longitud máxima del camino de una hormiga (λ_{ant}).

El mínimo número de pasos requeridos para encontrar un estado de aceptación es la longitud de la traza de error dividida entre λ_{ant} y multiplicada por el número de pasos por etapa σ_s, puesto que en cada etapa la profundidad de la región de exploración aumenta en λ_{ant}. Este razonamiento nos da una expresión que relaciona la longitud de las trazas de error (*len*) con el número mínimo de estados expandidos $exp_{min} \approx colsize \cdot \sigma_s \cdot \lambda_{ant} \cdot \lceil len/\lambda_{ant} \rceil$. Esta expresión es válida solamente para ACOhg-h y ACOhg-h+POR, no es una fórmula independiente del algoritmo. De acuerdo a la expresión previa, existe una relación casi lineal entre la longitud de las trazas de error y el número mínimo de estados expandidos. Además, el cociente exp_{min}/len debe ser aproximadamente $colsize \cdot \sigma_s$. Podemos constatar estas predicciones en la Figura 8.10, donde mostramos el número de estados expandidos frente a la longitud de las trazas de error para todas las ejecuciones independientes y todos los modelos de los experimentos. Podemos observar que la pendiente de la línea que mejor se ajusta a los puntos (41.96) se acerca a $colsize \cdot \sigma_s = 40$, el valor predicho para el cociente exp_{min}/len.

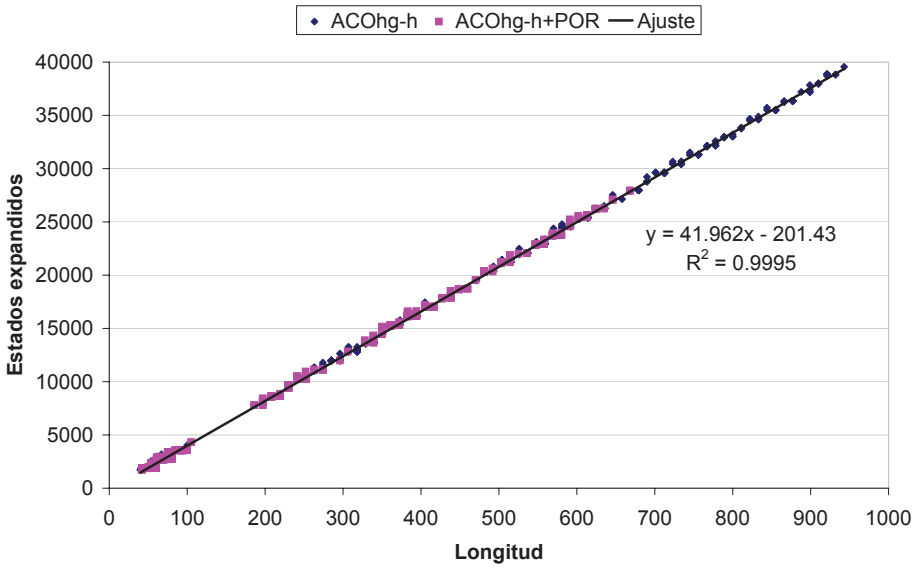


Figura 8.10: Relación lineal entre los estados expandidos y la longitud de las trazas de error.

8.6.3. Longitud de las trazas de error

Podemos observar en la Tabla 8.9 que ACOhg-h+POR obtiene trazas de error más cortas (con significancia estadística) que ACOhg-h en `giop41`, `giop61`, y las tres instancias de `marriers`. También podemos apreciar que ACOhg-h+POR mejora considerablemente las trazas de error obtenidas por ACOhg-h en las instancias de `marriers`. En las tres instancias de `leader` la longitud de las trazas de error obtenidas por ACOhg-h+POR son sólo unos estados más largas que las de ACOhg-h.

¿Por qué aumenta a veces la longitud de las trazas de error cuando se usa ACOhg-h+POR? La principal razón es que, en general, la reducción que realiza POR no mantiene los caminos óptimos. Es decir, los estados que pertenecen a un camino óptimo pueden ser reducidos y, por tanto, las trazas de error óptimas en el grafo reducido pueden ser más largas que las del modelo original. Este es un inconveniente bien conocido de POR que explica el resultado de las instancias de `leader` en la Tabla 8.9.

8.7. Resultados previos en la literatura

En esta sección se pretenden comparar los resultados de ACOhg-h con los de un algoritmo genético utilizado para el mismo problema en un trabajo de Godefroid y Khurshid [111]. En su trabajo, Godefroid y Khurshid tratan de encontrar un interbloqueo en el problema de los filósofos con 17 filósofos y el ataque de Lowe en el protocolo Needham-Schroeder. Mostramos en la Tabla 8.10 la tasa de éxito, el tiempo de CPU (en segundos) y la memoria requerida (en Kilobytes) por ACOhg-h y el GA propuesto por Godefroid y Khurshid en [111].

Observamos que ACOhg-h es capaz de encontrar siempre un error (100 % de éxito) en ambos programas, mientras que el GA encuentra un error sólo en el 52 % de los casos en `phi17` y en el 3 % en `needham`. Con respecto al tiempo de ejecución, los resultados revelan una gran diferencia entre los algoritmos:

Tabla 8.10: Resultados de ACOhg-h y el GA de [111].

Modelo	Algoritmo	Tasa de éxito (%)	Tiempo (s)	Memoria (KB)
`phi17`	GA	52	197.00	no disponible
	ACOhg-h	100	0.28	11274
`needham`	GA	3	3068.00	no disponible
	ACOhg-h	100	0.23	4865

ACOhg-h es entre dos y tres órdenes de magnitud más rápido que el GA. Esta diferencia no se puede explicar únicamente con la distinta potencia computacional de las máquinas usadas en los experimentos. El GA se ejecutó en un Pentium III a 700 MHz con 256 MB de RAM y el ACOhg-h en un Pentium 4 a 2.8 GHz con 512 MB de RAM. La cantidad máxima de memoria requerida por ACOhg-h es 11 MB. Esto significa que si ACOhg-h se hubiese ejecutado en la máquina de [111] no se habría producido un gran trasiego de bloques del proceso entre memoria primaria y secundaria (*swapping*) y el tiempo requerido para ejecutar ACOhg-h habría sido aproximadamente cuatro veces mayor que en el Pentium 4 (véase la Figura 8.11, que muestra los resultados del conjunto de pruebas SPEC CPU2000 para las dos máquinas). A pesar de esto, el tiempo de CPU requerido por ACOhg-h habría sido menor que el requerido por GA en dos o tres órdenes de magnitud.

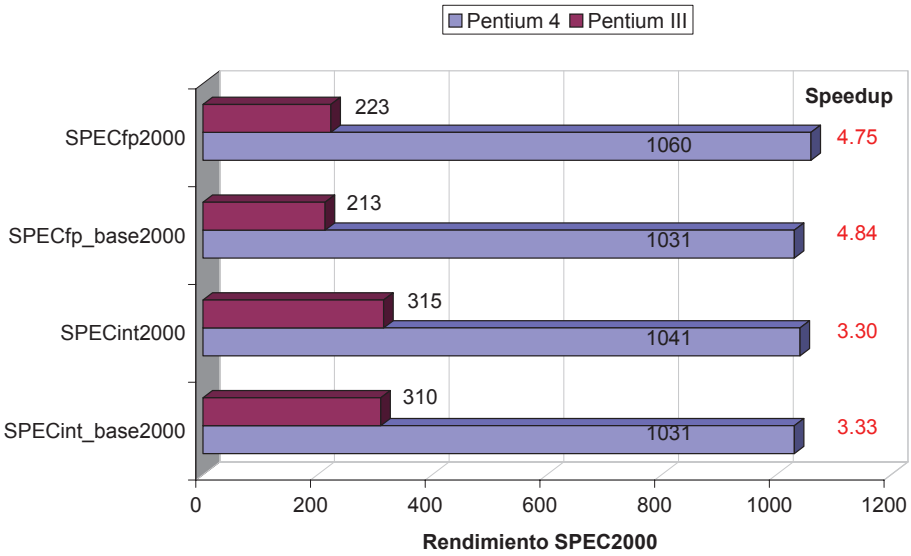


Figura 8.11: Resultados del conjunto de pruebas SPEC CPU2000 para las máquinas usadas en los experimentos con ACOhg-h y GA.

No obstante, aunque parece que ACOhg-h es mejor que GA, no podemos confirmar esta afirmación con los resultados de la Tabla 8.10 debido a tres motivos: los programas están implementados usando distintos lenguajes de programación (C en su caso y Promela en el nuestro), los *model checkers* usados en los experimentos son distintos (VeriSoft y HSF-SPIN, respectivamente), y la implementación de los algoritmos (el problema de los filósofos y el protocolo Needham-Schroeder) puede ser diferente. Por esto,

sólo mencionamos aquí los resultados obtenidos por Godefroid y Khurshid para ilustrar el estado del arte en la búsqueda de errores en sistemas concurrentes usando metaheurísticas.

8.8. Conclusiones

En este capítulo hemos presentado los resultados obtenidos al aplicar el nuevo modelo algorítmico ACOhg al problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. Hemos comparado el algoritmo con otros métodos exhaustivos y los resultados muestran que ACOhg es capaz de mejorar estos algoritmos en eficacia y eficiencia. Requiere una cantidad muy reducida de memoria y tiempo de CPU y es capaz de encontrar errores incluso en modelos en los que los algoritmos exhaustivos fallan en la práctica. Hemos estudiado también la influencia de varios parámetros del algoritmo en los resultados para dar algunas guías a los investigadores que usen este algoritmo. ACOhg se puede usar con otras técnicas tales como reducción de orden parcial, reducción de simetría o compresión de estados. En este capítulo hemos mostrado experimentalmente que puede combinarse con reducción de orden parcial para mejorar su rendimiento. Por último, hemos comparado los resultados de ACOhg con los de un trabajo que, hoy en día, representa el estado del arte en la aplicación de técnicas metaheurísticas al problema aquí tratado. Aunque la comparación no se ha realizado en igualdad de condiciones, los resultados obtenidos por ACOhg aventajan a los del GA, convirtiéndose, por tanto, en el nuevo estado del arte en este dominio.

Parte III

Conclusiones y trabajo futuro

Conclusiones

En esta tesis doctoral se estudia la aplicación de técnicas metaheurísticas a algunos problemas de optimización surgidos en el seno de la Ingeniería del Software. Para ello, y con el objetivo de seleccionar problemas de optimización que representen los intereses de la comunidad software, realizamos en primer lugar una revisión de la literatura en busca de problemas de Ingeniería del Software que puedan plantearse como tareas de optimización. Hemos clasificado los problemas de optimización en distintas categorías, que se corresponden con la fase del proceso de desarrollo software en la que se enmarcan. Observamos que la gran mayoría de los trabajos se centran en la fase de pruebas. La segunda categoría con más trabajos es la de gestión. Por este motivo, hemos decidido seleccionar un problema de gestión y dos de la fase de pruebas. En concreto los problemas son: la planificación de proyectos software, la generación automática de casos de prueba y la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes.

El siguiente paso ha consistido en definir formalmente los problemas seleccionados como problemas de optimización. En el caso de la planificación de proyectos y la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes ya existe una definición formal de los problemas. En cambio, para la generación de casos de prueba, a pesar de ser el problema de optimización más popular de la Ingeniería del Software, no encontramos una formalización adecuada para nuestros propósitos y hemos decidido realizarla nosotros. En particular, se formalizan distintas medidas de cobertura y criterios de adecuación. La formalización propuesta permite demostrar resultados teóricos propios del problema, algunos de los cuales no están muy extendidos por la comunidad científica. Nos referimos en concreto al hecho de que la cobertura de condiciones implica a la de ramas en programas con evaluación en cortocircuito de expresiones lógicas. Debido a este resultado, hemos decidido usar la cobertura de condiciones como criterio de adecuación para el problema de generación de casos de prueba.

A continuación estudiamos las técnicas metaheurísticas para decidir cuáles aplicar a cada problema seleccionado. A partir del estudio de sus características generales, se ha derivado un modelo formal de las metaheurísticas, basado en el propuesto por Gabriel Luque en su tesis doctoral, que define con precisión la evolución de los algoritmos metaheurísticos secuenciales durante la búsqueda teniendo en cuenta la componente estocástica de los mismos. También estudiamos en esta etapa de la investigación las propuestas paralelas de algoritmos metaheurísticos. Realizadas las indagaciones oportunas, seleccionamos los algoritmos para aplicar a cada problema.

Al abordar la resolución de los problemas, hemos descubierto que uno de ellos, la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes, presenta un grave inconveniente: los autómatas de Büchi que se deben explorar para resolver este problema tienen normalmente un tamaño excesivo para su almacenamiento en la memoria de una máquina. En ocasiones, incluso, el tamaño se desconoce. Para subsanar este inconveniente hemos desarrollado un nuevo modelo de algoritmo de optimización basado en colonias de hormigas que es capaz de afrontar problemas con estas características usando una cantidad

reducida de recursos computacionales. Aunque en esta tesis sólo se ha aplicado al mencionado problema, este modelo puede ser útil para resolver otros problemas con las mismas particularidades.

Para resolver el problema de planificación de proyectos software hemos utilizado un algoritmo genético con representación binaria. Usamos el algoritmo para proponer soluciones de planificación a distintos proyectos software automáticamente generados con un generador de instancias. Este generador, desarrollado durante el transcurso de nuestras investigaciones, permite analizar las soluciones propuestas por el algoritmo genético para proyectos con muy diversas características, ofreciendo resultados que ayudan al gestor de proyectos a tomar determinadas decisiones de planificación. Los experimentos realizados muestran que el algoritmo genético puede ser una técnica muy útil para la planificación de proyectos.

En el problema de generación de casos de prueba hemos estudiado la aplicación de algoritmos paralelos con población descentralizada. Los resultados apuntan a que la descentralización, tal y como se ha llevado a cabo, no mejora la búsqueda de casos de prueba en las instancias consideradas del problema. Asimismo, hemos usado dos técnicas metaheurísticas que nunca antes habían sido aplicadas a este problema. Se trata de las estrategias evolutivas y la optimización basada en cúmulos de partículas. Además de suponer una novedad, su aplicación a la generación de casos de prueba revela que ambas técnicas superan en eficacia a los algoritmos genéticos, ampliamente utilizados para esta labor durante muchos años.

La propuesta algorítmica seleccionada para buscar violaciones de propiedades de seguridad en sistemas concurrentes, ACOhg, es capaz de encontrar trazas de error cortas (buena calidad) usando una cantidad de memoria y tiempo muy reducida. Hemos realizado un estudio de las distintas alternativas del modelo algorítmico para seleccionar una configuración adecuada para el problema. Los resultados muestran que nuestra propuesta mejora algoritmos exactos que son el estado del arte en *model checking*. Dentro de los algoritmos exactos distinguimos dos grupos: los que requieren una reducida cantidad de recursos computacionales obteniendo largas trazas de error y los que consiguen trazas de error óptimas a cambio de un gran consumo de recursos. Nuestra propuesta consigue un equilibrio óptimo entre estos dos extremos para los sistemas concurrentes analizados. Hemos combinado ACOhg con una técnica para reducir el número de estados del autómata a explorar: reducción de orden parcial. Los resultados muestran una ventaja de la combinación de ambas técnicas en los modelos analizados.

Conclusions

In this PhD dissertation we study the application of metaheuristic techniques to some optimization problems from the software engineering domain. First, we have performed a review of the literature, searching for software engineering problems that can be translated into optimization tasks. We have classified the optimization problems in different categories according to the phase in which they appear in the software development process. We have observed that most of the work is focused on the testing phase. The second more popular category is that of software project management. For this reason, we have decided to select one problem from the management category and two from the testing phase in order to tackle problems that represent the current interest of the software community. In particular, the problems are: the software project scheduling problem, the test case generation problem, and the search for safety property violations in concurrent systems.

The next step have been the formal definition of the selected problems. For the project management and the search for safety property violations in concurrent systems there exists a formal definition. However, for the test case generation, despite the fact that it is the most popular software engineering optimization problem, we do not find an appropriate formalization for our purposes and we have decided to develop it. In particular, we formalize several coverage measurements and adequacy criteria. The proposed formalization allows to proof theoretical results, some of which are not very well spread in the scientific community. In particular, we refer to the fact that condition coverage implies branch coverage in programs with short-circuit evaluation of boolean expressions. Due to this result, we have decided to utilize condition coverage as adequacy criterion for the test case generation.

We study the metaheuristic techniques in order to decide which ones to utilize for solving each selected problem. From the study of the general metaheuristic features, we have derived a formal model based on the one proposed by Gabriel Luque in his PhD dissertation that defines with precision the evolution of the sequential metaheuristic algorithms along the search taking into account the stochastic component. We also study the parallel proposals for the metaheuristic algorithms. Once we perform the appropriate inquiries, we select the algorithms for each problem.

When we tackled the selected problems, we discovered that one of them, the search for safety property violations in concurrent systems, suffers from an important drawback: the Büchi automata that must be explored in order to solve this problem usually have a very large size and cannot be completely stored in the memory of one computer. Furthermore, sometimes the size of the automata is unknown. In order to overcome this drawback, we have developed a new algorithmic model based on ant colony optimization that is able to tackle optimization problems with the same features using only a reduced amount of computational resources. Although we only apply this model to the mentioned problem in this dissertation, the model can be useful for solving problems with similar characteristics.

We have utilized a genetic algorithm with binary representation for solving the software project scheduling problem. The algorithm proposes scheduling solutions for different software projects that are automatically generated by an instance generator. This generator, developed along the research, allows us to analyze the solutions proposed by the genetic algorithm for projects with very different features, offering results that help the project manager to take decisions about the scheduling. The experiments show that the genetic algorithm can be a very useful technique for software project scheduling.

In the test case generation problem we have studied the application of parallel algorithms with decentralized population. The results show that the decentralization model used does not improve the search for test cases in the instances tackled. We have also utilized two algorithms that have never been used for this problem in the past: evolutionary strategies and particle swarm optimization. In addition to the novelty, the application of these algorithms to the test case generation has revealed that both techniques outperform in efficacy the genetic algorithms, widely utilized for this task during many years.

The algorithmic proposal selected for searching for safety property violations in concurrent systems, ACOhg, is able to find short error trails (good quality) with a very reduced amount of memory and time. We have performed a study of the different alternatives of the algorithmic model in order to select a suitable configuration for the problem. The results show that our proposal outperforms exact algorithms that are the state-of-the-art in model checking. From these exact algorithms we distinguish two groups according to their behaviour: the ones that require a low amount of computational resources obtaining long error trails, and the ones that obtain optimal error trails with a very high resource consumption. Our proposal achieves an optimal trade-off between those two extremes for the concurrent systems analyzed. We have combined ACOhg with a technique for reducing the number of states of the automaton to explore: partial order reduction. The results show an advantage of the combination of both techniques in the analyzed models.

Trabajo futuro

Como fruto de la investigación desarrollada durante esta tesis son muchas las líneas de trabajo futuro que surgen. En los siguientes párrafos sintetizamos las principales.

Planificación de proyectos

El problema de planificación de proyectos, tal y como se ha definido en esta tesis, resulta ser un modelo muy simple de las tareas reales de gestión que se llevan a cabo en las empresas de software. Una línea de trabajo futuro en este campo puede ser el desarrollo de un modelo más complejo y realista que se adapte a las necesidades de los gestores de proyectos. Esto exige un estudio previo de las acciones de gestión que se llevan a cabo en la industria del software. Con un nuevo modelo más realista del problema, sería interesante aplicar las técnicas metaheurísticas a casos reales de la industria, comparando los resultados con los de gestores expertos.

El planteamiento multiobjetivo del problema es otra línea de trabajo que resulta de gran interés para el gestor de software. En la formulación del problema utilizada en esta tesis, tratamos de optimizar coste y duración del proyecto. Hemos podido comprobar que estos son dos objetivos contrapuestos. Nosotros hemos abordado una formulación mono-objetivo del problema asignando pesos al coste y la duración del proyecto. Estos pesos representan la importancia que tiene para el gestor del proyecto cada uno de esos aspectos. Sin embargo, es posible aplicar técnicas de optimización multi-objetivo para resolver este problema, en cuyo caso, el gestor dispondría de todo un conjunto de soluciones de planificación de las cuáles puede seleccionar la que más le interese dependiendo de las circunstancias. Una ventaja adicional de la formulación multi-objetivo es que desaparecen los pesos y la necesidad de ajustarlos.

Generación automática de casos de prueba

En la generación de casos de prueba, los paradigmas dinámico y simbólico han sido principalmente abordados por separado en la literatura. Existen algunos trabajos preliminares, no obstante, que muestran que su combinación puede ser muy beneficiosa [112, 249]. Una posible línea de trabajo futuro consiste en combinar las técnicas metaheurísticas con otras técnicas que ayuden a generar casos de prueba, como la ejecución simbólica o el análisis estático.

Por otro lado, para abordar la generación de casos de prueba basada en el paradigma de caja blanca, generalmente se parte del código fuente del programa y se incluyen instrucciones en éste para calcular medidas de cobertura. Existe, sin embargo, la posibilidad, poco explorada, de aplicar las técnicas de

generación de casos de prueba directamente a código ejecutable. Esto permite probar, e incluso depurar a bajo nivel, software del cuál no se posee el código fuente.

Para la generación de casos de prueba no es necesario tener el programa objeto. Es posible obtener casos de prueba a partir de modelos más abstractos del software. Un claro ejemplo es la extracción de secuencias de prueba a partir de modelos de uso de software [81] permite planificar. Estos modelos de uso suelen ser cadenas de Markov que contienen información acerca de la probabilidad de utilizar una u otra funcionalidad del software. Las cadenas de Markov tienen limitaciones que pueden evitarse usando extensiones de ellas [80]. En esta línea, una clara aportación novedosa sería la aplicación de técnicas metaheurísticas a la generación de secuencias de prueba usando extensiones de cadenas de Markov para modelar el uso del software.

Una cuarta línea de investigación que queda abierta en esta tesis es profundizar en los motivos que impiden que las versiones con población descentralizada de los algoritmos evolutivos mejoren los resultados de las versiones con población centralizada. Una comprensión profunda de estos motivos puede servir de guía para el diseño de un modelo descentralizado competitivo.

Búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes

Por lo que respecta a la búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes, surgen muchas líneas trabajo futuro, algunas de las cuáles se encuentran ya en desarrollo. En primer lugar, se puede extender la búsqueda de errores a propiedades de viveza. Esto se puede realizar en dos fases: primero, el algoritmo busca un estado de aceptación y, después, intenta encontrar un camino a ese estado partiendo de él mismo. Se puede aplicar también la idea de clasificar los componentes fuertemente conexos del autómata de Büchi para cambiar la forma en que se realiza la búsqueda en cada componente dependiendo de su tipo, tal y como se hace en [92].

Otra línea de trabajo consiste en desarrollar un *model checker* basado en ACOhg para verificar sistemas concurrentes. Dicha aplicación sería capaz de encontrar rápidamente errores cuando existen, mientras que certificaría la corrección del programa cuando no existen.

Por otro lado, los *model checkers* trabajando en paralelo en un *cluster*, o incluso un *grid*, de máquinas están ganando importancia en la comunidad de métodos formales. Asimismo, existe mucho trabajo afirmando la alta eficiencia y eficacia de las metaheurísticas paralelas [4]. Por esto, una línea clara de trabajo futuro sería el uso de versiones de ACOhg paralelas para buscar errores en grandes sistemas concurrentes, o incluso, software concurrente de una aplicación real.

Planeamos combinar también ACOhg con otras técnicas propias de *model checking* para reducir el espacio de búsqueda, tales como la reducción de simetría o la compresión de estados. Una última línea de investigación futura persigue integrar el algoritmo ACOhg dentro de Java PathFinder, que es capaz de trabajar con programas en lenguaje Java, mucho más familiar y popular para la comunidad informática que el lenguaje Promela.

Metaheurísticas

En el dominio de las metaheurísticas son dos las aportaciones principales de esta tesis, cada una de las cuales abre una posible línea de trabajo futuro. En primer lugar, el modelo matemático de metaheurísti-

cas secuenciales presentado permite estudiar desde un punto de vista teórico las técnicas metaheurísticas, llegando, posiblemente, a resultados bien conocidos en la práctica pero no fundamentados en la teoría. El estudio detallado de este modelo para sacarle el máximo partido puede ser una prometedora línea de trabajo futuro. Por otro lado, hemos desarrollado un nuevo modelo algorítmico basado en colonias de hormigas, ACOhg, que permite resolver problemas hasta ahora inabordables con las técnicas metaheurísticas conocidas. Las ideas de este modelo se pueden trasladar a otras metaheurísticas para elaborar nuevos modelos algorítmicos que resuelvan este tipo de problemas.

Parte IV

Apéndices

Apéndice A

Análisis de la configuración de los algoritmos

En este apéndice presentamos algunos resultados secundarios, obtenidos durante la realización de la tesis doctoral, en los que se analiza la configuración de los algoritmos utilizados en los problemas de generación de casos de prueba y de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. A continuación, presentamos los estudios realizados para cada problema en dos secciones.

A.1. Generación de casos de prueba

En esta sección realizamos un estudio de parámetros para la estrategia evolutiva y el algoritmo genético utilizados para resolver el problema de generación de casos de prueba. La metodología seguida en estos estudios consiste en modificar un parámetro mientras que se mantienen fijos los demás. Una vez que se ha encontrado el mejor valor para dicho parámetro, se fija y se procede de igual modo con otro. Este modo de proceder asume que no existen dependencias entre parámetros, lo cual no es cierto en todos los casos. Sin embargo, el procedimiento descrito arroja una menor cantidad de datos para tratar, lo cual hace el proceso más sencillo y comprensible.

A.1.1. Selección de parámetros para ES

En primer lugar, realizamos un estudio de parámetros para ES. En concreto, variamos el número de hijos generados en cada iteración (λ) y el tamaño de la población (μ). No ajustamos los parámetros del operador de mutación porque están bien establecidos en la literatura [27, 240]. La condición de parada de ES consiste en encontrar una solución o realizar un máximo de 100 evaluaciones. Para cada programa realizamos un test estadístico (con $\alpha = 0.05$) entre las diferentes configuraciones del algoritmo para comprobar si las diferencias son significativas. Los detalles del test estadístico realizado y sus resultados se encuentran en la Sección B.2. Primero, mantenemos el valor de $\mu = 10$ y probamos tres valores para λ: 1, 2 y 3. La Tabla A.1 y la Figura A.1 resumen los resultados obtenidos.

Como podemos observar en la Tabla A.1, la influencia del número de hijos en la cobertura es insignificante; sólo en `triangle`, `calday`, `sa` y `netflow` se obtiene una cobertura ligeramente superior aumentando

Tabla A.1: Resultados obtenidos al cambiar el número de hijos λ en ES.

Programas	(10+1)-ES		(10+2)-ES		(10+3)-ES	
	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.
`triangle`	99.76 0.97	873.50 528.24	**100.00** 0.00	**675.20** 460.35	99.84 0.88	1292.80 1045.59
`gcd`	**100.00** 0.00	**20.33** 1.80	**100.00** 0.00	21.40 7.31	**100.00** 0.00	20.43 1.86
`calday`	94.24 4.05	**1275.13** 369.85	**96.36** 4.13	1768.13 781.56	96.06 4.02	2233.03 1344.22
`crc`	**100.00** 0.00	15.87 14.68	**100.00** 0.00	44.73 89.06	**100.00** 0.00	**10.00** 0.00
`insertion`	**100.00** 0.00	**10.00** 0.00	**100.00** 0.00	**10.00** 0.00	**100.00** 0.00	**10.00** 0.00
`shell`	**100.00** 0.00	**10.00** 0.00	**100.00** 0.00	**10.00** 0.00	**100.00** 0.00	**10.00** 0.00
`quicksort`	**94.12** 0.00	**1110.00** 0.00	**94.12** 0.00	2110.00 0.00	**94.12** 0.00	3110.00 0.00
`heapsort`	**100.00** 0.00	**10.00** 0.00	**100.00** 0.00	**10.00** 0.00	**100.00** 0.00	**10.00** 0.00
`select`	**83.33** 0.00	**1119.00** 3.00	**83.33** 0.00	2118.67 3.40	**83.33** 0.00	3119.33 2.49
`bessel`	**97.56** 0.00	**1193.33** 78.62	97.48 0.44	2363.07 441.01	97.48 0.44	3507.23 524.83
`sa`	98.76 0.75	**1102.50** 244.94	98.98 0.83	1694.80 862.86	**99.27** 0.84	2482.37 1192.86
`netflow`	98.13 0.16	**1497.03** 261.72	**98.17** 0.00	2509.07 324.17	**98.17** 0.00	3658.47 389.03

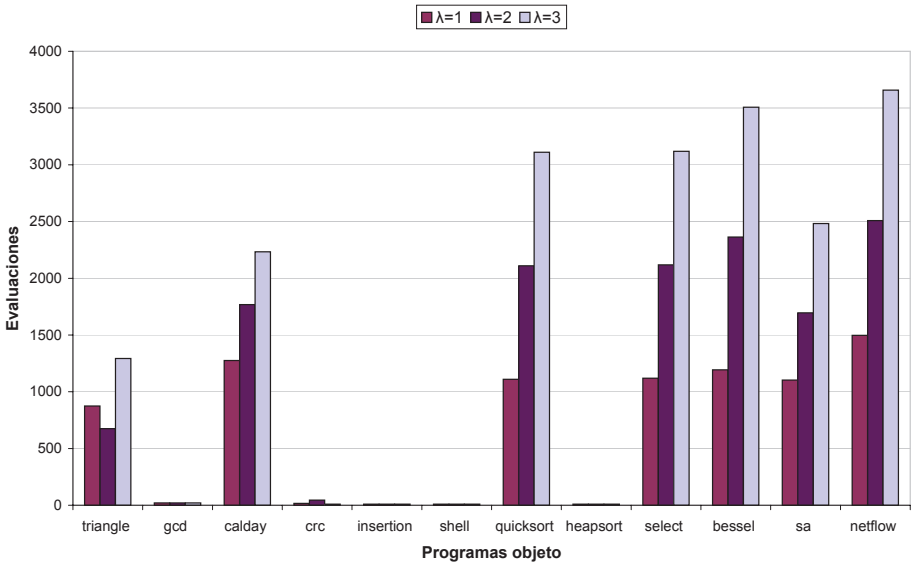


Figura A.1: Número de evaluaciones en ES para λ=1, 2 y 3. El valor de μ es 10.

λ. De hecho, hay diferencias estadísticamente significativas sólo en el caso de `sa`. Esta pequeña influencia puede deberse a la alta cobertura intrínseca que obtiene ES en todos los programas objeto. El número de evaluaciones aumenta, en general, con el número de hijos, ya que en cada paso de la ES hay más individuos para evaluar. La diferencia en el número de evaluaciones es estadísticamente significativa para al menos un par de configuraciones en `triangle`, `calday`, `quicksort`, `select`, `bessel`, `sa` y `netflow` (véase la Figura B.9). A partir de los resultados, concluimos que $\lambda = 1$ parece ser el mejor valor para el número de hijos. Estudiaremos ahora la influencia del tamaño de la población μ. Fijamos el número

de hijos al mejor valor obtenido previamente $\lambda = 1$. En la Tabla A.2 y la Figura A.2 presentamos los resultados de este segundo experimento.

Tabla A.2: Resultados obtenidos al cambiar el tamaño de la población μ en ES.

Programas	(1+1)-ES				(5+1)-ES				(10+1)-ES				(20+1)-ES				(30+1)-ES			
	Cob. (%)		Eval.		Cob. (%)		Eval.		Cob. (%)		Eval.		Cob. (%)		Eval.		Cob. (%)		Eval.	
triangle	33.01	12.75	1020.00	0.00	99.43	2.65	**704.80**	454.16	99.76	0.97	873.50	528.24	99.76	0.97	981.90	409.52	**99.84**	0.88	1107.13	484.76
gcd	82.00	4.00	1020.00	0.00	**100.00**	0.00	**16.80**	4.35	**100.00**	0.00	20.33	1.80	**100.00**	0.00	30.00	0.00	**100.00**	0.00	40.00	0.00
calday	80.30	4.44	1020.00	0.00	**97.27**	3.64	**888.07**	386.73	94.24	4.05	1275.13	369.85	91.97	2.54	1391.27	353.13	91.06	0.82	1448.30	204.22
crc	99.80	1.06	43.67	181.30	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	15.87	14.68	**100.00**	0.00	16.83	15.82	**100.00**	0.00	25.77	59.00
insertion	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00
shell	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00
quicksort	**94.12**	0.00	**1020.00**	0.00	**94.12**	0.00	1060.00	0.00	**94.12**	0.00	1110.00	0.00	**94.12**	0.00	1210.00	0.00	**94.12**	0.00	1310.00	0.00
heapsort	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00	**100.00**	0.00	**10.00**	0.00
select	69.26	25.77	**1020.00**	0.00	**83.33**	0.00	1064.83	0.90	**83.33**	0.00	1119.00	3.00	**83.33**	0.00	1228.00	6.00	**83.33**	0.00	1336.00	10.20
bessel	81.95	8.20	**1020.00**	0.00	97.24	1.75	1252.13	274.15	**97.56**	0.00	1193.33	78.62	**97.56**	0.00	1349.60	142.94	**97.56**	0.00	1464.80	203.98
sa	96.55	0.30	**1020.00**	0.00	98.59	0.63	1042.80	285.51	98.76	0.75	1102.50	244.94	**99.04**	0.84	1070.50	375.57	98.87	0.80	1140.23	463.70
netflow	91.44	1.93	**1020.00**	0.00	98.13	0.16	1522.70	314.55	98.13	0.16	1497.03	261.72	**98.17**	0.00	1580.50	230.99	**98.17**	0.00	1722.83	223.70

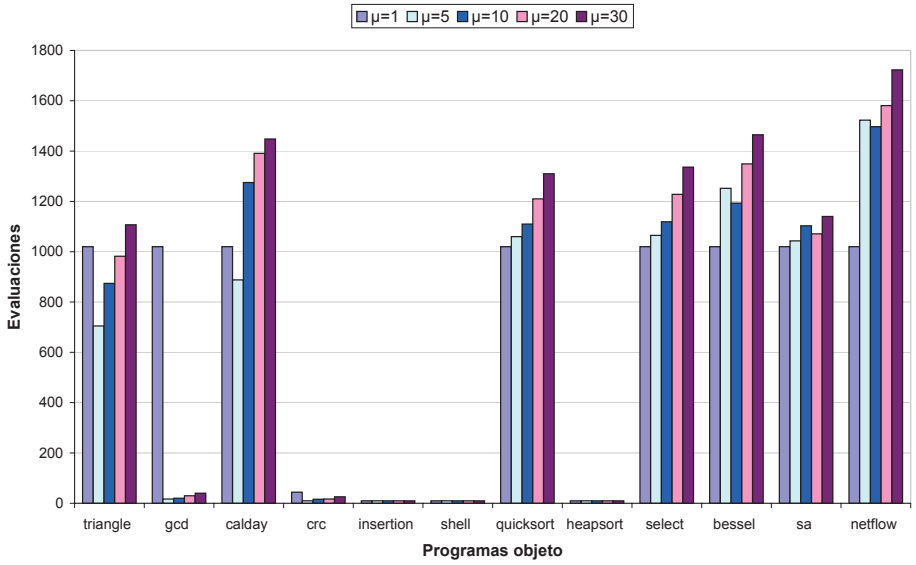


Figura A.2: Número de evaluaciones en ES para μ=1, 5, 10, 20 y 30. El valor de λ es 1.

En este caso, la cobertura obtenida por las diferentes configuraciones es similar cuando la población tiene más de un individuo. En `triangle`, `gcd`, `calday`, `select`, `bessel`, `sa` y `netflow` la diferencia entre la cobertura de la (1+1)-ES y el resto de configuraciones es estadísticamente significativa. No obstante, no hay diferencia significativa en la cobertura entre el resto de configuraciones excepto para `calday` entre la (5+1)-ES y las (20+1)-ES y (30+1)-ES. Con respecto al número de evaluaciones, aumenta, en general, con el tamaño de población porque hay más individuos en la población inicial para evaluar. La mayoría de las diferencias estadísticamente significativas apoyan esta observación. No obstante, existen excepciones notables a esta regla en (1+1)-ES para `triangle` y `gcd`. Con una población de un individuo, la ES no puede conseguir cobertura total en ambos programas y esto explica el alto número de evaluaciones requerido por el generador.

Como conclusión global, podemos afirmar que el número de hijos (λ) tiene una escasa influencia en la cobertura, pero una influencia importante en el número de evaluaciones requerido y, como consecuencia, en el tiempo de ejecución del generador. Con respecto al tamaño de la población (μ), observamos que la cobertura es aproximadamente la misma cuando el tamaño de la población está por encima de unos pocos individuos. El uso de un solo individuo en la población debe evitarse para conseguir una alta cobertura. En general, el número de evaluaciones aumenta con el tamaño de la población. No obstante, el crecimiento es moderado en comparación con el que provoca un aumento de λ. Para finalizar esta discusión, concluimos que la mejor configuración de ES para este banco de pruebas es $\mu = 5$ y $\lambda = 1$.

A.1.2. Selección de parámetros para GA

A continuación realizamos un estudio de la influencia de varios parámetros de GA en los resultados. La motivación de este estudio es realizar una comparación justa entre los algoritmos usados en el Capítulo 7. Aquí probamos diferentes parámetros para la mutación y la recombinación del GA. En concreto, vamos a modificar la desviación estándar de la mutación normal, la probabilidad de mutación, el operador de recombinación y su probabilidad. Para los experimentos usaremos los programas `triangle`, `gcd`, `calday`, `crc`, `select`, `bessel` y `netflow`. En todos los casos usamos 25 individuos en la población, selección aleatoria, reemplazo ($\mu + \lambda$) y el algoritmo se detiene cuando encuentra una solución o alcanza un máximo de 500 evaluaciones. La elección del parámetro en cada caso se ha realizado teniendo en cuenta los resultados de un test estadístico aplicado a los mismos. Los detalles y resultados de este test se encuentran en el Apéndice B. Para el primer experimento usamos recombinación uniforme con probabilidad 1.0 y mutación normal con probabilidad 0.2. La media de la mutación es 0 y para la desviación estándar probamos tres valores: 1, 10 y 100. Los resultados están en la Tabla A.3.

Tabla A.3: Resultados obtenidos cuando cambia la desviación estándar de la mutación en GA.

Programa	$\sigma = 1$		$\sigma = 10$		$\sigma = 100$	
	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.
`triangle`	**99.43** $_{1.36}$	3961.27 $_{2842.92}$	98.86 $_{1.96}$	**3175.30** $_{2324.51}$	97.15 $_{2.68}$	4718.13 $_{2273.11}$
`gcd`	**100.00** $_{0.00}$	**190.93** $_{125.36}$	**100.00** $_{0.00}$	253.73 $_{177.68}$	**100.00** $_{0.00}$	1213.80 $_{984.96}$
`calday`	90.91 $_{0.00}$	224.30 $_{404.98}$	90.91 $_{0.00}$	**114.30** $_{106.60}$	**91.97** $_{1.92}$	1548.00 $_{1766.80}$
`crc`	**100.00** $_{0.00}$	**10.30** $_{1.19}$	**100.00** $_{0.00}$	10.80 $_{2.01}$	**100.00** $_{0.00}$	10.60 $_{1.62}$
`select`	**83.33** $_{0.00}$	196.37 $_{163.81}$	**83.33** $_{0.00}$	**171.70** $_{211.17}$	**83.33** $_{0.00}$	517.93 $_{699.79}$
`bessel`	**97.56** $_{0.00}$	**264.13** $_{223.66}$	**97.56** $_{0.00}$	293.27 $_{167.85}$	**97.56** $_{0.00}$	1038.87 $_{861.67}$
`netflow`	96.33 $_{0.00}$	525.47 $_{435.14}$	96.12 $_{1.15}$	**454.07** $_{328.43}$	**96.36** $_{0.16}$	706.10 $_{578.18}$

A partir de los resultados de la tabla, podemos ver que la desviación estándar tiene un influencia diferente en cada programa. Esto no es sorprendente: significa que cada programa tiene asociado una configuración óptima particular. Aquí, una desviación estándar baja parece ser buena para la cobertura de `triangle`, pero no tan buena para la cobertura de `calday` y `netflow`. No obstante, la mayoría de las diferencias no son estadísticamente significativas. Sólo en el caso de `triangle` y `calday` podemos encontrar diferencias significativas y contradictorias. Por otro lado, el número de evaluaciones es más alto con significancia estadística en `gcd`, `calday` y `bessel` para una desviación estándar de 100. Atendiendo a los resultados, decidimos mantener una desviación estándar baja ($\sigma = 1$). En el siguiente experimento analizamos cinco valores para la probabilidad de mutación: 0.2, 0.4, 0.6, 0.8 y 1.0. Los resultados se encuentran en la Tabla A.4.

Tabla A.4: Resultados obtenidos cambiando la probabilidad de mutación en GA.

Programa	p_m = 0.2		p_m = 0.4		p_m = 0.6		p_m = 0.8		p_m = 1.0	
	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.
triangle	99.43 1.36	3961.27 2842.92	**99.76** 0.73	3057.07 2094.38	99.19 2.12	3175.23 2531.06	99.51 1.16	4033.43 2679.74	99.51 1.59	**2327.27** 1615.96
gcd	**100.00** 0.00	190.93 125.36	**100.00** 0.00	193.33 133.89	**100.00** 0.00	153.43 100.55	**100.00** 0.00	165.60 105.90	**100.00** 0.00	**119.40** 55.94
calday	**90.91** 0.00	224.30 404.98	**90.91** 0.00	109.73 81.56	**90.91** 0.00	82.80 56.25	**90.91** 0.00	84.73 58.99	**90.91** 0.00	**67.30** 42.96
crc	**100.00** 0.00	**10.30** 1.19	**100.00** 0.00	10.83 1.77	**100.00** 0.00	10.37 1.52	**100.00** 0.00	10.57 1.31	**100.00** 0.00	10.50 1.57
select	**83.33** 0.00	196.37 163.81	**83.33** 0.00	173.83 118.21	**83.33** 0.00	**112.70** 76.88	**83.33** 0.00	124.33 112.37	**83.33** 0.00	117.00 98.70
bessel	**97.56** 0.00	264.13 223.66	**97.56** 0.00	182.97 123.48	**97.56** 0.00	198.73 149.36	**97.56** 0.00	188.80 127.80	**97.56** 0.00	**135.50** 91.49
netflow	96.33 0.00	525.47 435.14	96.33 0.00	710.80 561.99	**96.39** 0.23	789.53 1101.82	96.33 0.00	**473.40** 321.08	96.36 0.16	692.53 887.97

No podemos observar una influencia clara de la probabilidad de mutación en la cobertura. De hecho, el test estadístico muestra que las diferencias no son significativas. No obstante, con respecto al número de evaluaciones observamos un descenso en varios programas cuando la probabilidad es alta. De nuevo, las diferencias no son significativas (excepto en el caso de `bessel` para probabilidades 0.2 y 1.0) así que no podemos concluir que un valor de probabilidad es mejor que otro. Finalmente, decidimos asignar a la probabilidad de mutación un valor intermedio: 0.6. A continuación, vamos a utilizar tres operadores de recombinación diferentes: cruce uniforme, cruce de un punto y cruce de dos puntos. Los resultados están en la Tabla A.5.

Tabla A.5: Resultados obtenidos cambiando el operador de recombinación en GA.

Programa	Uniforme		Un punto		Dos puntos	
	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.
`triangle`	99.19 2.12	**3175.23** 2531.06	99.51 1.16	4088.60 1950.89	**99.67** 1.04	3209.47 1985.48
`gcd`	**100.00** 0.00	**153.43** 100.55	**100.00** 0.00	317.77 293.13	**100.00** 0.00	257.20 198.82
`calday`	**90.91** 0.00	82.80 56.25	**90.91** 0.00	255.07 240.14	**90.91** 0.00	**75.03** 42.81
`crc`	**100.00** 0.00	**10.37** 1.52	**100.00** 0.00	10.80 1.62	**100.00** 0.00	**10.37** 1.97
`select`	**83.33** 0.00	112.70 76.88	**83.33** 0.00	110.43 100.78	**83.33** 0.00	**83.20** 70.96
`bessel`	**97.56** 0.00	198.73 149.36	**97.56** 0.00	**185.77** 129.48	**97.56** 0.00	533.03 418.76
`netflow`	96.39 0.23	789.53 1101.82	96.33 0.00	**626.27** 516.77	**96.42** 0.36	917.90 974.56

Observamos una ligera ventaja (no significativa) del cruce de dos puntos con respecto a la cobertura. No obstante, en general, el número de evaluaciones es también alto con este operador. A pesar de ello, decidimos seleccionar DPX como operador de cruce. El último experimento de esta sección se usa para seleccionar la probabilidad de recombinación. Probamos cinco valores para la probabilidad: 0.2, 0.4, 0.6, 0.8 y 1.0. Mostramos los resultados en la Tabla A.6.

Tabla A.6: Resultados obtenidos al cambiar la probabilidad de recombinación en GA.

Programa	p_c = 0.2		p_c = 0.4		p_c = 0.6		p_c = 0.8		p_c = 1.0	
	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.	Cob. (%)	Eval.
triangle	98.13 2.72	4814.97 2793.13	97.97 2.75	**3208.60** 2134.57	99.02 2.14	3817.57 2336.91	99.51 1.32	3310.90 1868.48	**99.67** 1.04	3209.47 1985.48
gcd	**100.00** 0.00	**117.53** 62.86	**100.00** 0.00	131.03 61.46	**100.00** 0.00	145.20 78.53	**100.00** 0.00	210.17 129.63	**100.00** 0.00	257.20 198.82
calday	**90.91** 0.00	75.37 42.66	**90.91** 0.00	92.97 58.28	**90.91** 0.00	**58.83** 38.60	**90.91** 0.00	75.00 46.34	**90.91** 0.00	75.03 42.81
crc	**100.00** 0.00	11.03 2.74	**100.00** 0.00	11.30 4.87	**100.00** 0.00	11.13 4.59	**100.00** 0.00	**10.00** 0.00	**100.00** 0.00	10.37 1.97
select	**83.33** 0.00	92.07 62.02	**83.33** 0.00	99.50 81.25	**83.33** 0.00	93.50 62.65	**83.33** 0.00	107.77 79.15	**83.33** 0.00	**83.20** 70.96
bessel	**97.56** 0.00	**127.10** 80.54	**97.56** 0.00	170.10 98.43	**97.56** 0.00	193.17 135.18	**97.56** 0.00	264.90 279.08	**97.56** 0.00	533.03 418.76
netflow	96.36 0.16	1277.07 755.76	96.36 0.16	1236.60 1025.29	96.36 0.16	1060.63 1016.00	96.33 0.00	**902.67** 693.20	**96.42** 0.36	917.90 974.56

En general, observamos una mayor cobertura y menor eficiencia con probabilidades altas. Las pocas diferencias que son estadísticamente significativas apoyan esta observación. Por esta razón, elegimos la más alta probabilidad, 1.0, para la recombinación de dos puntos. La configuración final del GA está formada

por el cruce de dos puntos con probabilidad 1.0 y la mutación normal con media 0, desviación estándar 1 y probabilidad 0.6. Esta es la configuración de los generadores de casos de prueba basados en GA y dGA de los experimentos del Capítulo 7.

A.2. Búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes

En esta sección analizamos la influencia de varios parámetros de ACOhg en los resultados. El objetivo de estos experimentos es comprender el funcionamiento del algoritmo para proporcionar a los investigadores algunas guías para aplicarlo. El modelo usado para estos experimentos es el que implementa el problema de los filósofos de Dijkstra (`phi`n). Usamos este modelo porque es simple y escalable. Por esto, podemos usar una versión del modelo tan grande como queramos. Su simplicidad nos permite estudiarlo de forma teórica para calcular el número de estados del modelo. La versión con n filósofos tiene 3^n estados y un solo estado de interbloqueo. Es más, la traza de error óptima tiene longitud $n + 1$.

En la Sección A.2.2 analizamos la influencia en los resultados de la longitud del camino de las hormigas λ_{ant}. Este primer estudio extiende al realizado en la Sección 8.3. La Sección A.2.3 analiza la técnica misionera y la Sección A.2.4 la de expansión, comparándose ambas técnicas en la Sección A.2.5. De nuevo, los experimentos realizados en estas tres secciones extienden a los ya mostrados en la Sección 8.4. Posteriormente, en las Secciones A.2.6, A.2.7 y A.2.8 se estudian la escalabilidad del algoritmo, la influencia del factor de potencia de la heurística y la influencia de las penalizaciones de la función de *fitness*, respectivamente. Omitimos en todos estos experimentos el número de estados expandidos por no ser una medida relevante desde el punto de vista del algoritmo.

A.2.1. Parámetros base

Para los experimentos usamos un ACOhg con la configuración base mostrada en la Tabla A.7. Salvo que se diga lo contrario, se usa la técnica misionera. En las siguientes secciones cambiaremos el valor de algunos parámetros para estudiar su influencia. Si no se menciona explícitamente que un parámetro cambia, se asume que su valor es el de la Tabla A.7.

En los experimentos se realizan 100 ejecuciones para cada configuración del algoritmo y mostramos en las tablas de resultados las medias y desviaciones de todas las ejecuciones. Realizamos tests estadísticos de todas las comparaciones, cuyos resultados se muestran en el Apéndice B. En las siguientes secciones, tan sólo mencionaremos si el test estadístico correspondiente apoya las observaciones o no. Con respecto a la información heurística, salvo que se diga lo contrario, usamos $\eta_{ij} = 1/(1 + H_{ap}(j))$, donde $H_{ap}(j)$ es el número de procesos activos en el estado j, es decir, se trata de un ACOhg-h.

A.2.2. Longitud del camino de las hormigas

En este primer experimento cambiamos la longitud máxima de los caminos que construyen las hormigas durante la fase de construcción. Probamos diferentes valores para esta longitud máxima, que va desde la longitud de una solución óptima d_{opt} hasta $3d_{opt}$ en pasos de tres unidades. Este experimento es una extensión de los experimentos mostrados en la Sección 8.3 y, como allí, mantenemos el valor de λ_{ant} constante durante toda la búsqueda. Además, todas las hormigas comienzan la construcción en el nodo

Tabla A.7: Parámetros base para ACOhg.

Parámetro	Valor
Iteraciones	10
colsize	5
λ_{ant}	10
σ_s	2
s	10
ξ	0.8
a	5
ρ	0.6
α	1.0
β	1.0
p_p	1000
p_c	1000

inicial: no se usa la técnica misionera. Usamos una versión del modelo con $n = 20$ filósofos. Dicho modelo es suficientemente grande para observar la influencia de λ_{ant} en los resultados, mostrados en la Tabla A.8.

Tabla A.8: Resultados obtenidos con $n = 20$ filósofos para distintos valores deλ_{ant}.

λ_{ant}	Tasa éxito	Long.	Mem. (KB)	Tiempo (ms)
21	24.00	$21.00_{0.00}$	$32554.75_{13945.80}$	$603.33_{341.56}$
24	23.00	$21.00_{0.00}$	$30939.00_{12947.75}$	$543.91_{300.60}$
27	53.00	$22.58_{1.96}$	$30762.89_{15645.57}$	$534.72_{359.24}$
30	72.00	$24.94_{3.23}$	$29163.39_{17890.15}$	$519.58_{425.19}$
33	86.00	$27.00_{4.13}$	$31830.63_{19103.86}$	$567.33_{442.27}$
36	78.00	$26.74_{4.28}$	$30263.77_{18417.56}$	$531.28_{448.12}$
39	93.00	$28.48_{4.88}$	$29127.08_{17771.35}$	$498.92_{439.61}$
42	99.00	$30.98_{6.19}$	$25090.01_{17284.00}$	$407.17_{408.36}$
45	99.00	$33.00_{7.46}$	$21951.78_{14184.09}$	$330.10_{319.59}$
48	99.00	$33.36_{7.30}$	$24521.23_{17074.12}$	$394.34_{414.96}$
51	100.00	$34.48_{8.27}$	$21850.14_{12720.28}$	$322.40_{281.04}$
54	100.00	$38.20_{8.90}$	$19057.19_{10966.23}$	$262.80_{211.58}$
57	100.00	$38.96_{9.11}$	$17681.94_{10205.60}$	$228.90_{216.30}$
60	100.00	$38.04_{10.85}$	$17203.26_{8511.75}$	$216.80_{181.97}$
63	100.00	$40.36_{11.49}$	$17581.37_{10522.31}$	$228.50_{202.96}$

Observamos en la segunda columna de la Tabla A.8 que la tasa de éxito aumenta cuando las hormigas construyen caminos más largos. La razón es que la parte del grafo explorada crece con λ_{ant} y las hormigas encuentran caminos distintos al nodo objetivo. No obstante, estos caminos son, en general, más largos que el óptimo. Esto se puede apreciar en la tercera columna de la Tabla A.8, donde la longitud media de las trazas de error aumenta con λ_{ant}. De hecho, todas las diferencias que son estadísticamente significativas confirman esta observación (véase la Sección B.3).

El hecho de que el algoritmo encuentre más fácilmente las trazas de error cuando λ_{ant} aumenta, influye también en la memoria y el tiempo de CPU requerido para encontrar el error. En las columnas cuarta y quinta de la Tabla A.8 podemos observar que tanto la memoria como el tiempo disminuyen cuando λ_{ant} aumenta (con confianza estadística). La probabilidad de encontrar una traza de error crece con λ_{ant} y el número de pasos requeridos para encontrarla disminuye. De este modo, además de la reducción en el tiempo, el número de rastros de feromona almacenados en memoria también se reduce y, con ello, la memoria requerida.

A.2.3. Técnica misionera

En el siguiente experimento usamos de nuevo $n = 20$ filósofos para estudiar la eficacia de la técnica misionera. Este análisis extiende aquél de la Sección 8.4.1. Son tres los parámetros que gobiernan esta técnica: s, σ_s y λ_{ant}. Estudiamos diferentes valores para cada uno de estos tres parámetros con el objetivo de investigar su influencia en los resultados. Primero, fijamos s a 10, como en la configuración base, y cambiamos los valores de σ_s y λ_{ant}. En las Tablas A.9, A.10, A.11 y A.12 presentamos la tasa de éxito, la longitud de las trazas de error, la memoria requerida y el tiempo de CPU requerido, respectivamente.

Tabla A.9: Análisis de la técnica misionera. Tasa de éxito.

σ_s	λ_{ant}				
	5	10	15	20	25
1	38	91	99	100	100
2	10	64	95	99	100
3	0	41	89	99	100
4	0	39	84	98	100
5	0	0	63	84	99
6	0	0	61	85	97
7	0	0	51	84	96
8	0	0	40	76	95
9	0	0	17	53	82
10	0	0	0	0	60

A partir de los resultados concluimos que la tasa de éxito es mayor cuando el número de pasos por etapa σ_s es bajo, es decir, cuando se realizan más etapas. Este es un resultado esperado, ya que de este modo las hormigas pueden alcanzar nodos más profundos en el grafo y pueden encontrar más caminos diferentes que alcanzan el nodo objetivo. Observamos de nuevo que la tasa de éxito aumenta con λ_{ant}. Por esta razón los valores más altos de éxito (100 %) se encuentran para valores bajos de σ_s y valores altos de λ_{ant}.

Con respecto a la longitud de las trazas de error (Tabla A.10), observamos que aumenta con el número de etapas realizadas (σ_s bajo) y con λ_{ant}. Esta observación está apoyada por los tests estadísticos. Un gran número de etapas implica la exploración de nodos más profundos en el grafo y, como consecuencia, la probabilidad de encontrar caminos más largos aumenta. Lo mismo sucede cuando aumenta λ_{ant}, como vimos en la sección anterior. En general, valores bajos de σ_s y altos de λ_{ant} implican mayor tasa de éxito, pero trazas de error más largas. El usuario del algoritmo debe buscar el balance ideal entre eficacia y calidad de la solución.

Tabla A.10: Análisis de la técnica misionera. Longitud.

σ_s	λ_{ant}				
	5	10	15	20	25
1	$36.58_{8.20}$	$51.73_{17.53}$	$56.64_{22.36}$	$58.20_{26.49}$	$55.28_{24.43}$
2	$22.60_{1.96}$	$35.88_{8.99}$	$41.84_{13.37}$	$41.57_{14.74}$	$42.20_{13.05}$
3	-	$26.95_{3.96}$	$32.33_{8.61}$	$35.10_{9.98}$	$34.36_{9.05}$
4	-	$25.31_{3.19}$	$28.90_{6.17}$	$31.08_{6.86}$	$33.96_{8.77}$
5	-	-	$24.68_{2.88}$	$28.19_{4.85}$	$30.98_{7.38}$
6	-	-	$23.75_{3.03}$	$29.05_{5.04}$	$30.44_{6.85}$
7	-	-	$25.31_{3.05}$	$28.57_{4.66}$	$27.79_{6.38}$
8	-	-	$24.80_{3.09}$	$27.95_{4.83}$	$27.99_{7.18}$
9	-	-	$24.76_{2.90}$	$26.58_{4.81}$	$27.63_{6.03}$
10	-	-	-	-	$22.87_{2.00}$

Tabla A.11: Análisis de la técnica misionera. Memoria requerida.

σ_s	λ_{ant}				
	5	10	15	20	25
1	$4016.89_{72.68}$	$6436.65_{77.30}$	$8310.70_{116.36}$	$9719.76_{172.69}$	$10280.96_{200.66}$
2	$5507.40_{133.39}$	$8467.06_{113.41}$	$11210.11_{228.66}$	$13229.25_{314.75}$	$14399.85_{1291.05}$
3	-	$10364.88_{335.08}$	$15118.38_{434.78}$	$18245.82_{394.95}$	$19191.94_{3297.21}$
4	-	$13180.72_{342.34}$	$18834.29_{500.11}$	$23071.35_{511.04}$	$24465.29_{3883.55}$
5	-	-	$22641.78_{321.81}$	$27928.38_{483.18}$	$28289.27_{6804.83}$
6	-	-	$26523.28_{402.19}$	$32635.48_{561.71}$	$31614.64_{9662.81}$
7	-	-	$30378.67_{482.72}$	$37388.19_{535.69}$	$33507.68_{12847.60}$
8	-	-	$34124.80_{531.47}$	$42213.05_{540.80}$	$37339.66_{14026.63}$
9	-	-	$37827.76_{240.94}$	$46930.11_{620.67}$	$40800.35_{16018.36}$
10	-	-	-	-	$32902.08_{15991.07}$

Si echamos un vistazo a la cantidad de recursos requeridos por el algoritmo, apreciaremos que tanto la memoria como el tiempo de CPU aumentan con σ_s con confianza estadística. Es decir, como ocurre con la tasa de éxito, se prefieren valores bajos de σ_s para reducir la cantidad de recursos requeridos. La explicación está relacionada con la eliminación de rastros de feromona tras una etapa. En las Figuras A.3 y A.4 mostramos una traza de la memoria requerida a lo largo de la evolución del algoritmo. Durante una etapa, la cantidad de memoria requerida aumenta linealmente. Cuando la siguiente etapa comienza, se eliminan los rastros de feromona y la memoria requerida cae a un valor bajo. Podemos observar esto en las figuras. Si la eliminación de feromona es muy frecuente (σ_s bajo) la cantidad media de memoria requerida es baja, pero si la eliminación de feromona no es tan frecuente (σ_s alto), la memoria media requerida es mayor. La influencia de λ_{ant} no está tan clara. Cuando λ_{ant} aumenta, lo mismo sucede con el incremento de memoria por paso. Por esta razón, la pendiente de la curva es mayor para $\lambda_{ant} = 20$ (Figura A.4) que para $\lambda_{ant} = 10$ (Figura A.3). La eliminación de feromona introduce un factor de incertidumbre que impide predecir la influencia de λ_{ant} en el consumo de memoria. No obstante, podemos ver en las Figuras A.3 y A.4 que la memoria media requerida en las etapas dos, tres y siguientes es aproximadamente la misma. Sólo en la primera etapa hay una diferencia notable en la memoria requerida. Esta diferencia

en la primera etapa parece ser la razón del aumento en el consumo de memoria cuando λ_{ant} aumenta. Los tests estadísticos apoyan esta observación. De hecho, en los casos en los que la regla general no se cumple (cuarta y quinta columnas de la Tabla A.11 para $\sigma_s \geq 6$) las diferencias en los resultados no son estadísticamente significativas.

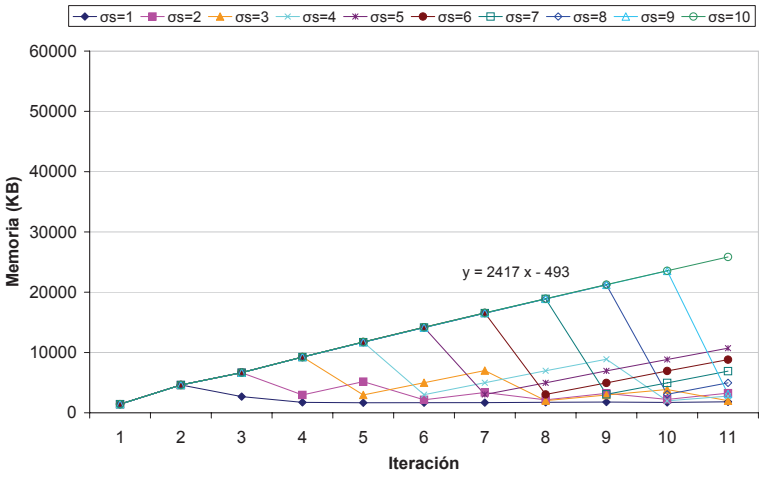


Figura A.3: Evolución del consumo de memoria cuando $\lambda_{ant} = 10$.

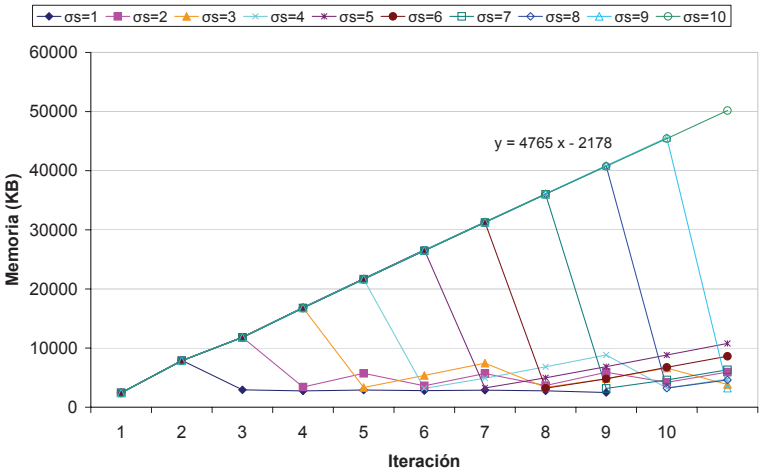


Figura A.4: Evolución del consumo de memoria cuando $\lambda_{ant} = 20$.

Con respecto al tiempo de CPU, la influencia de σ_s sigue una tendencia clara (estadísticamente confirmada). Cuando σ_s es bajo, la probabilidad de encontrar una solución es mayor y se encuentra más rápidamente. La influencia de λ_{ant}, sin embargo, no es tan clara. Por un lado, cuando λ_{ant} aumenta se requiere más tiempo para construir el camino de una hormiga (columnas 1 a 4 en la Tabla A.12); los tests

Tabla A.12: Análisis de la técnica misionera. Tiempo de CPU requerido.

σ_s	λ_{ant}				
	5	10	15	20	25
1	$96.05_{8.44}$	$147.03_{25.18}$	$176.46_{32.08}$	$207.60_{39.98}$	$221.40_{42.10}$
2	$168.00_{7.48}$	$271.56_{29.85}$	$321.16_{50.45}$	$382.42_{66.89}$	$395.30_{78.64}$
3	-	$381.95_{24.81}$	$470.79_{52.24}$	$580.20_{67.48}$	$565.50_{173.72}$
4	-	$520.51_{18.67}$	$653.81_{58.39}$	$810.20_{66.61}$	$820.40_{240.88}$
5	-	-	$837.14_{60.35}$	$1093.81_{68.49}$	$1023.54_{420.65}$
6	-	-	$1070.00_{43.98}$	$1406.47_{137.86}$	$1258.66_{608.86}$
7	-	-	$1317.45_{50.52}$	$1741.55_{60.35}$	$1411.46_{857.98}$
8	-	-	$1578.75_{44.00}$	$2135.39_{74.36}$	$1653.47_{1049.52}$
9	-	-	$1872.35_{58.36}$	$2555.66_{82.68}$	$1967.20_{1336.91}$
10	-	-	-	-	$953.67_{919.50}$

estadísticos apoyan esta observación. Por otro lado, la probabilidad de encontrar una solución es mayor y el tiempo medio para encontrarla se reduce. Aunque este fenómeno se observa en la quinta columna de la Tabla A.12, las diferencias en los resultados no son estadísticamente significativas, así que sólo podemos afirmar que el tiempo de CPU requerido aumenta con λ_{ant}.

En conclusión, desde un punto de vista práctico, podemos afirmar lo siguiente: si se requiere una traza de error rápida y/o usando una baja cantidad de memoria, se debe usar un valor de σ_s bajo; pero si se prefieren trazas de error cortas, se debe usar un valor alto de σ_s, o incluso evitar la técnica misionera. Con respecto a λ_{ant}, un valor alto puede aumentar la probabilidad de encontrar una solución pero también aumenta la cantidad de recursos requeridos cuando se usa la técnica misionera.

Soluciones guardadas

Ahora vamos a estudiar la influencia en los resultados de s, el número de soluciones guardadas entre etapas. Para este experimento usamos $\lambda_{ant} = 25$. En la Tablas A.13, A.14 y A.15 mostramos la tasa de éxito, la longitud de las trazas de error y la cantidad de memoria requerida, respectivamente, cuando cambian s y σ_s. El máximo valor de σ_s es nueve, porque diez o más significa que la técnica misionera no se utiliza (ya que el máximo número de pasos del algoritmo es diez de acuerdo a la Tabla A.7) y, en este caso, guardar soluciones entre etapas no tiene sentido porque sólo hay una etapa. En otras palabras, s no influiría en los resultados.

Observamos en la Tabla A.13 que s no tiene influencia en la tasa de éxito. Es decir, usar más nodos como puntos de partida para las hormigas no aumenta la probabilidad de encontrar una solución. No obstante, este comportamiento puede deberse a las particulares características del modelo usado para los experimentos, en el que el estado de interbloqueo se puede alcanzar desde cualquier otro estado. En los modelos con estados que no pueden alcanzar ningún nodo objetivo, quizá podamos encontrar una configuración particular en la que s tenga influencia en la tasa de éxito.

Con respecto a la longitud de las trazas de error, observamos que cuando s aumenta, la longitud media de las trazas de error se alarga también para valores bajos de σ_s con confianza estadística. El algoritmo siempre guarda las mejores s soluciones. Por esto, si s es bajo, la longitud media de las soluciones guardadas es menor que si s es alto. Es decir, si s es alto, las hormigas pueden comenzar sus caminos en nodos con una mayor profundidad y las soluciones que construyen son más largas. Este comportamiento es claro

Tabla A.13: Análisis de las soluciones guardadas. Tasa de éxito.

σ_s	s				
	2	4	6	8	10
1	100	100	100	100	100
2	100	100	100	100	100
3	100	100	100	100	100
4	100	100	99	100	100
5	98	98	99	99	99
6	100	100	100	98	97
7	99	98	98	98	96
8	98	99	99	98	95
9	96	92	90	87	82

Tabla A.14: Análisis de las soluciones guardadas. Longitud.

σ_s	s				
	2	4	6	8	10
1	$41.52_{14.48}$	$52.44_{17.80}$	$55.24_{23.42}$	$54.40_{22.97}$	$55.28_{24.43}$
2	$35.72_{10.54}$	$37.92_{10.75}$	$40.12_{11.35}$	$39.68_{13.16}$	$42.20_{13.05}$
3	$33.56_{7.74}$	$34.64_{8.91}$	$33.04_{8.59}$	$34.36_{9.44}$	$34.36_{9.05}$
4	$30.00_{6.65}$	$31.52_{6.71}$	$32.64_{8.14}$	$33.48_{8.51}$	$33.96_{8.77}$
5	$29.85_{6.16}$	$29.82_{6.41}$	$30.70_{7.19}$	$31.30_{8.34}$	$30.98_{7.38}$
6	$28.88_{6.80}$	$29.20_{6.26}$	$29.28_{6.28}$	$30.96_{7.26}$	$30.44_{6.85}$
7	$28.20_{5.00}$	$29.00_{5.91}$	$28.18_{6.02}$	$29.57_{7.38}$	$27.79_{6.38}$
8	$27.23_{4.84}$	$27.79_{6.71}$	$27.46_{6.12}$	$28.88_{7.24}$	$27.99_{7.18}$
9	$26.64_{4.77}$	$26.17_{5.13}$	$26.29_{4.76}$	$27.76_{6.84}$	$27.63_{6.03}$

cuando el número de etapas es alto (σ_s bajo) pero no se observa cuando σ_s es alto. De hecho, de acuerdo a los tests estadísticos, las diferencias entre las longitudes de las trazas de error no son estadísticamente significativas cuando $\sigma_s \geq 5$, es decir, la influencia de s en la longitud de las trazas de error se reduce conforme σ_s aumenta.

Si echamos un vistazo a la Tabla A.15 podemos observar que no hay una clara influencia de s en la memoria requerida. Quizá podríamos esperar un ligero aumento en la memoria requerida conforme s aumenta (las únicas dos diferencias estadísticamente significativas apoyan esta hipótesis). No obstante, los resultados muestran que este ligero aumento en la memoria requerida queda enmascarado por la aleatoriedad del algoritmo. Por esto, concluimos que s no tiene influencia apreciable en la memoria requerida, al menos para $s \leq 10$. No mostramos los resultados del tiempo de CPU requerido porque el comportamiento es similar al obtenido en la memoria.

En conclusión, podemos afirmar que es preferible un valor bajo de s porque la longitud media de las trazas de error es menor en este caso y tanto la tasa de éxito como la memoria no dependen de s.

Tabla A.15: Análisis de las soluciones guardadas. Memoria requerida.

σ_s	s									
	2		4		6		8		10	
1	10112.95	823.60	10156.94	865.67	10214.54	621.49	10058.16	1020.52	10280.96	200.66
2	14171.23	1581.27	14248.48	1302.37	14235.47	1741.02	14406.50	1372.04	14399.85	1291.05
3	19660.60	2690.12	19606.93	2490.63	18364.97	4706.13	19181.42	3266.93	19191.94	3297.21
4	23982.08	4816.46	24113.39	4914.82	24101.95	4471.87	24499.62	4004.68	24465.29	3883.55
5	27962.86	7127.85	27902.32	7489.48	27783.02	6931.23	27126.36	8041.50	28289.27	6804.83
6	30155.75	10346.14	33189.07	8233.85	30892.58	9257.78	33144.63	7921.21	31614.64	9662.81
7	35313.72	11882.09	35325.66	12209.06	35863.67	11508.24	33544.72	12923.98	33507.68	12847.60
8	40044.09	12303.05	38334.01	13626.18	40759.55	11862.62	40125.62	12991.84	37339.66	14026.63
9	40853.95	15651.99	37787.13	16058.37	43417.22	14001.06	43634.08	13765.52	40800.35	16018.36

Eliminación de feromona

Finalmente, queremos comprobar en esta sección si la eliminación de todos los rastros de feromona entre las etapas tiene alguna influencia en los resultados. Para hacerlo, comparamos una versión del algoritmo en la que se eliminan todos los rastros de feromona después de cada etapa, y otra en la que no se eliminan. Para este experimento hacemos $\lambda_{ant} = 25$ y $s = 2$. En la Tabla A.16 mostramos la tasa de éxito, la longitud de las trazas de error y la memoria requerida. Mostramos también los resultados de un test estadístico (detalles en el Apéndice B) comparando las dos variantes de los algoritmos con respecto a la longitud y a la memoria.

Tabla A.16: Análisis de la eliminación de feromona.

σ_s	No se elimina			Se elimina			Test estad.	
	Éxito	Long.	Mem. (KB)	Éxito	Long.	Mem. (KB)	Long.	Mem.
1	100.00	43.68 14.22	11519.17 1706.41	100.00	41.52 14.48	10112.95 823.60	-	+
2	100.00	33.88 8.63	16555.47 2535.88	100.00	35.72 10.54	14171.23 1581.27	-	+
3	100.00	31.76 8.70	20146.70 4584.13	100.00	33.56 7.74	19660.60 2690.12	+	+
4	100.00	28.60 6.29	24442.83 6240.22	100.00	30.00 6.65	23982.08 4816.46	-	+
5	100.00	27.00 5.04	28595.77 8195.32	99.00	29.85 6.16	27962.86 7127.85	+	+
6	100.00	27.68 5.63	33315.26 9706.80	100.00	28.88 6.80	30155.75 10346.14	-	+
7	99.00	27.10 5.10	36525.46 11231.01	100.00	28.20 5.00	35313.72 11882.09	-	+
8	99.00	26.37 4.80	38855.26 14212.31	97.00	27.23 4.84	40044.09 12303.05	-	-
9	93.00	25.52 4.77	39883.61 17758.00	95.00	26.64 4.77	40853.95 15651.99	+	-
10	50.00	23.24 1.99	30753.10 15645.00	56.00	23.07 2.00	31547.95 16281.97	-	-

Podemos observar en la tabla que la eliminación de rastros de feromona tras cada etapa no tiene una gran influencia en la tasa de éxito o la longitud media de las trazas de error encontradas (sólo hay tres diferencias estadísticamente significativas), pero, como se esperaba, tiene una ligera influencia en la memoria requerida (siete diferencias estadísticamente significativas). No obstante, la memoria extra requerida cuando no se elimina la feromona no es demasiado elevada. Esto indica que la mayoría de la memoria se utiliza en la primera etapa. Para ilustrar esto, mostramos en las Figuras A.5 y A.6 la evolución de la memoria cuando se elimina la feromona y cuando no se elimina, respectivamente.

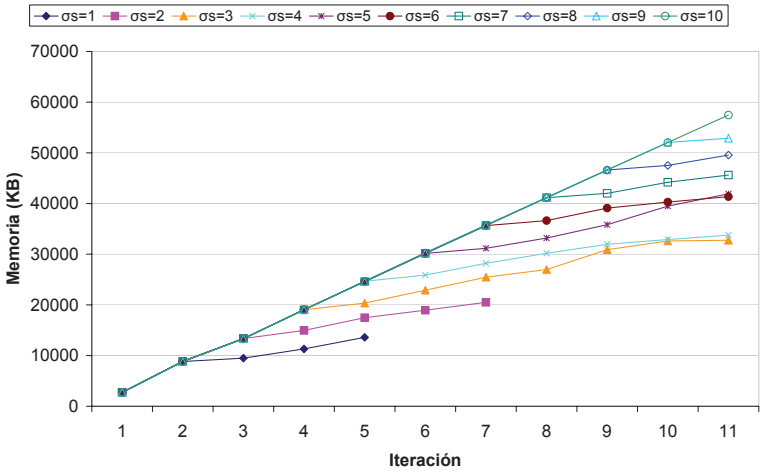


Figura A.5: Evolución del consumo de memoria cuando no se eliminan los rastros de feromona.

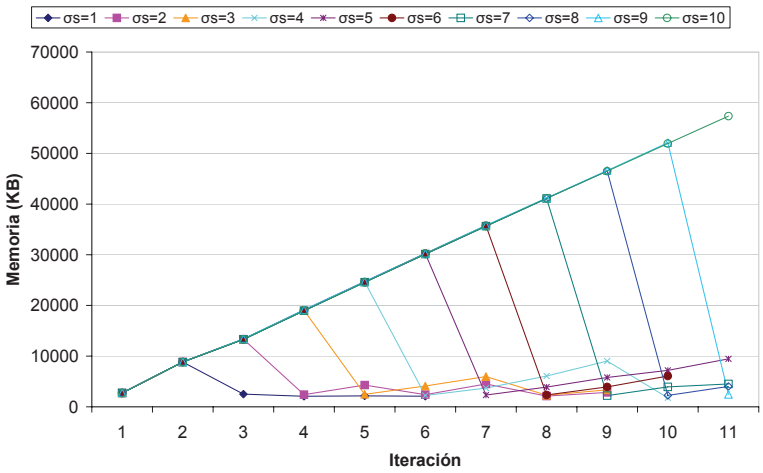


Figura A.6: Evolución del consumo de memoria cuando se eliminan los rastros de feromona.

En la Figura A.5 observamos que la pendiente de la curva decrece cuando el algoritmo comienza la segunda etapa en todos los casos. Esto significa que el número de caminos que se pueden trazar desde el nodo inicial es mayor que el del resto de los nodos.

A.2.4. Técnica de expansión

En esta sección usamos de nuevo $n = 20$ filósofos para estudiar la eficacia de la técnica de expansión. Existen dos parámetros que gobiernan esta técnica: σ_i, y δ_l. Estudiamos diferentes valores para estos parámetros con el objetivo de investigar su influencia en los resultados. Establecemos el valor inicial de

λ_{ant} al mismo valor que δ_l. De este modo, la profundidad máxima que el algoritmo puede alcanzar en cada paso del algoritmo es el mismo que en la técnica misionera analizada en la sección anterior, permitiendo su comparación directa. En las Tablas A.17, A.18, A.19 y A.20, mostramos la tasa de éxito, la longitud de las trazas de error, la memoria requerida y el tiempo requerido, respectivamente.

Tabla A.17: Análisis de la técnica de expansión. Tasa de éxito.

σ_i	δ_l				
	5	10	15	20	25
1	83	100	100	100	100
2	13	86	100	100	100
3	0	42	91	100	100
4	0	15	75	95	100
5	0	0	53	73	95
6	0	0	39	64	92
7	0	0	21	62	92
8	0	0	17	43	73
9	0	0	12	25	65
10	0	0	0	0	55

A partir de la Tabla A.17 obtenemos las mismas conclusiones que en la técnica misionera. La tasa de éxito aumenta cuando σ_i es bajo, es decir, cuando λ_{ant} aumenta frecuentemente. En esta situación el algoritmo puede encontrar diferentes caminos al nodo objetivo, aumentando la probabilidad de encontrar una solución. La tasa de éxito aumenta también cuando δ_l crece, ya que la porción del grafo explorado crece con este parámetro.

Tabla A.18: Análisis de la técnica de expansión. Longitud.

σ_i	δ_l				
	5	10	15	20	25
1	$27.80_{6.29}$	$32.72_{8.91}$	$32.36_{9.36}$	$33.20_{9.46}$	$34.56_{10.16}$
2	$22.54_{1.95}$	$29.09_{6.51}$	$31.20_{8.11}$	$32.28_{7.87}$	$33.12_{9.92}$
3	-	$25.86_{4.50}$	$29.40_{6.44}$	$30.88_{8.44}$	$31.00_{8.38}$
4	-	$25.27_{3.41}$	$28.36_{5.95}$	$29.97_{7.08}$	$30.80_{8.70}$
5	-	-	$24.09_{2.65}$	$28.34_{5.26}$	$30.31_{8.24}$
6	-	-	$24.38_{2.80}$	$28.62_{5.60}$	$27.35_{7.67}$
7	-	-	$25.95_{3.00}$	$27.45_{5.67}$	$27.13_{6.74}$
8	-	-	$24.53_{2.33}$	$27.23_{5.20}$	$27.08_{6.62}$
9	-	-	$25.00_{3.27}$	$27.40_{5.66}$	$26.11_{6.33}$
10	-	-	-	-	$23.47_{1.94}$

La influencia de σ_i y δ_l en la longitud media de las trazas de error es similar a la influencia de σ_s y λ_{ant} en la técnica misionera. Cuando la tasa de éxito es alta (valor de σ_s bajo y δ_l alto), la longitud de las trazas de error está lejos de ser óptima (observación apoyada por los tests estadísticos mostrados en la Sección B.3). Esta configuración favorece la construcción de caminos más largos al nodo objetivo, lo que aumenta la probabilidad de encontrar una solución, pero también produce trazas de error más largas.

Tabla A.19: Análisis de la técnica de expansión. Memoria requerida.

σ_i	δ_l				
	5	10	15	20	25
1	$39528.87_{11621.10}$	$35973.12_{13120.52}$	$27228.16_{10647.72}$	$26275.84_{9334.95}$	$22835.03_{8850.82}$
2	$33083.08_{2735.46}$	$43781.95_{11503.85}$	$37109.76_{13268.70}$	$33597.44_{10347.67}$	$27924.48_{10173.07}$
3	-	$41179.43_{7609.08}$	$40431.12_{14030.01}$	$41502.72_{13812.58}$	$33232.14_{14469.32}$
4	-	$40413.87_{2893.09}$	$43499.52_{11265.07}$	$46500.38_{13440.72}$	$38780.55_{17534.43}$
5	-	-	$40361.06_{7401.39}$	$45743.34_{10503.21}$	$41158.63_{16196.22}$
6	-	-	$38098.05_{6304.60}$	$47568.00_{8247.67}$	$39905.42_{17513.09}$
7	-	-	$43203.05_{4017.46}$	$50935.74_{5945.16}$	$42128.70_{15125.65}$
8	-	-	$41984.00_{2809.83}$	$52390.70_{3661.78}$	$41161.03_{19258.29}$
9	-	-	$43264.00_{609.40}$	$53616.64_{955.34}$	$39422.82_{19094.53}$
10	-	-	-	-	$31028.69_{15029.95}$

Tabla A.20: Análisis de la técnica de expansión. Tiempo de CPU requerido.

σ_i	δ_l				
	5	10	15	20	25
1	$752.05_{285.28}$	$623.90_{336.19}$	$419.40_{218.51}$	$392.30_{192.09}$	$323.30_{158.96}$
2	$620.77_{69.78}$	$848.14_{293.77}$	$658.80_{317.41}$	$549.40_{223.16}$	$428.30_{210.26}$
3	-	$792.38_{194.51}$	$759.34_{358.33}$	$759.20_{357.02}$	$565.00_{324.47}$
4	-	$787.33_{95.81}$	$842.40_{293.89}$	$907.05_{362.04}$	$716.80_{458.72}$
5	-	-	$761.32_{196.33}$	$873.97_{288.17}$	$764.74_{395.81}$
6	-	-	$716.41_{165.61}$	$923.91_{227.12}$	$753.70_{426.28}$
7	-	-	$842.38_{106.45}$	$1040.65_{171.58}$	$801.52_{370.61}$
8	-	-	$820.59_{78.33}$	$1088.14_{133.04}$	$796.85_{465.04}$
9	-	-	$875.83_{23.96}$	$1146.00_{61.84}$	$763.23_{481.59}$
10	-	-	-	-	$542.18_{356.74}$

Los recursos requeridos (memoria y tiempo de CPU) toman sus valores mínimos normalmente cuando σ_i es bajo. Un valor bajo de σ_i aumenta la cantidad de memoria requerida en cada paso porque λ_{ant} aumenta muy frecuentemente (véase la Figura A.7). Un aumento en λ_{ant} implica también un aumento en el tiempo de CPU. No obstante, cuando σ_i es bajo, la probabilidad de encontrar una solución antes es mayor y hay un equilibrio entre ambas tendencias. Podemos observar en la Tabla A.19 que, en este caso, el aumento en la probabilidad de encontrar una solución es más importante, y normalmente la memoria requerida se reduce cuando σ_i es bajo. La influencia de δ_l en el consumo de memoria tiene una explicación similar. Cuando δ_l aumenta encontramos dos tendencias opuestas: el aumento en la memoria requerida por cada paso y el aumento en la probabilidad de encontrar una solución. No obstante, en este caso no está claro qué factor tiene más importancia y no podemos observar una tendencia clara en los resultados.

A.2.5. Comparación entre la técnica misionera y la de expansión

En esta sección comparamos la técnica misionera y la de expansión, que han sido analizadas separadamente en las secciones previas. Comparamos los resultados de los algoritmos que implementan la técnica

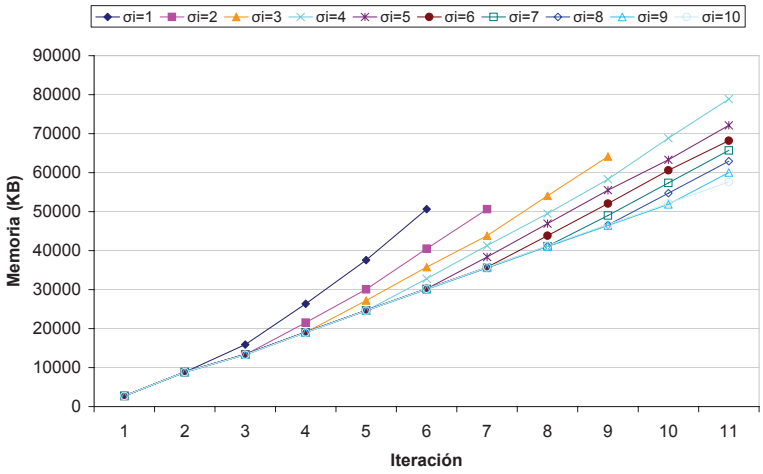


Figura A.7: Evolución de la memoria requerida en la búsqueda para $\delta_l = 10$.

misionera con aquéllos que implementan la técnica de expansión cuando $\sigma_s = \sigma_i$ y λ_{ant}(misionera)= δ_l (el resto de los parámetros son los de la Tabla A.7). De este modo, podemos hacer una comparación justa entre ambas técnicas, ya que la profundidad máxima que alcanzan los algoritmos durante la búsqueda en cada paso es la misma. Presentamos en las Figuras A.8, A.9, A.10 y A.11 la tasa de éxito, la longitud de las trazas de error, la memoria usada y el tiempo de CPU requerido por las dos técnicas. Estas figuras contienen toda la información mostrada en las tablas de las dos secciones anteriores.

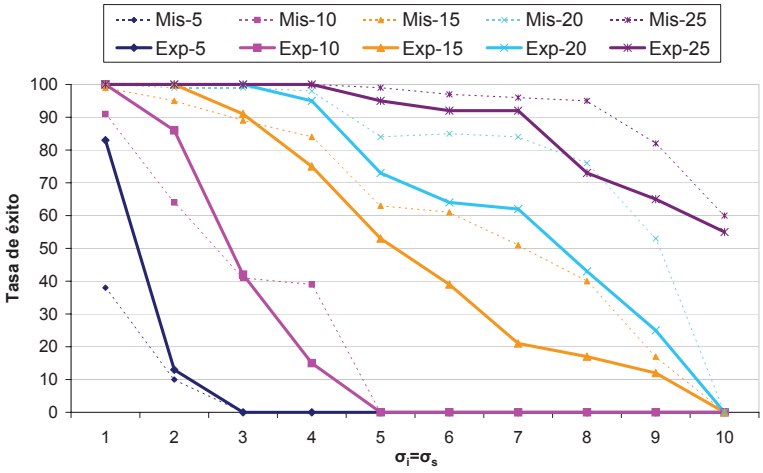


Figura A.8: Comparación entre la técnica misionera y la de expansión. Tasa de éxito.

En la Figura A.8 podemos observar que la técnica de expansión consigue una tasa de éxito mayor que la misionera para todos los valores de σ_i y σ_s cuando los incrementos en la profundidad de exploración máxima (δ_l y λ_{ant} respectivamente) son bajos (Mis-5 y Exp-5). No obstante, esta tendencia cambia cuando

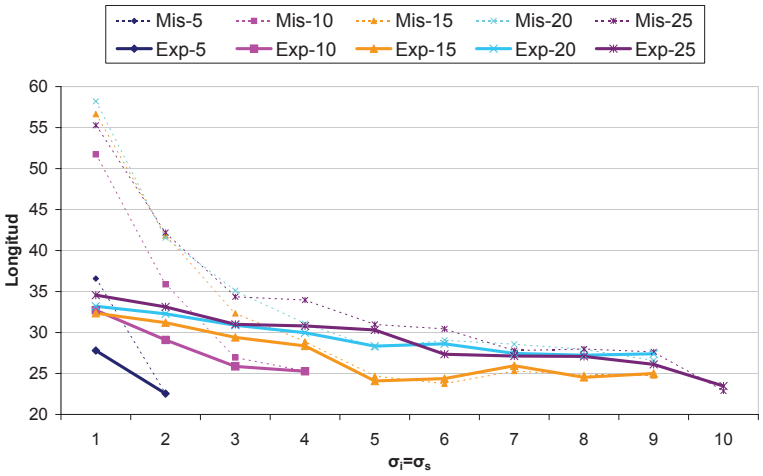


Figura A.9: Comparación entre la técnica misionera y la de expansión. Longitud de las trazas de error.

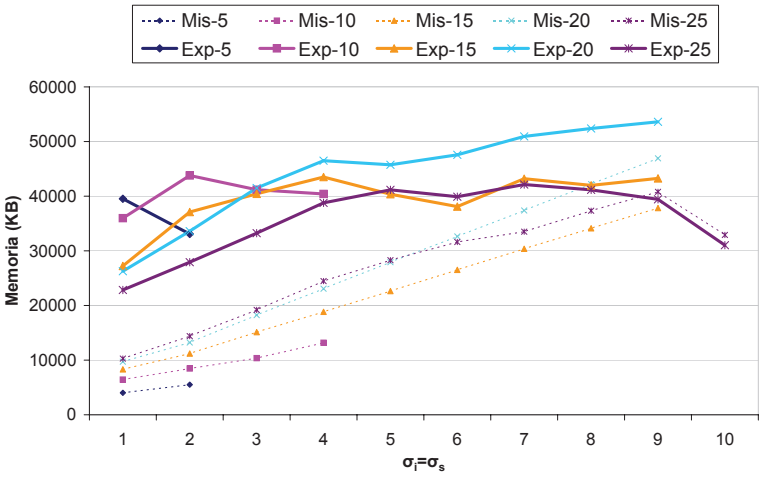


Figura A.10: Comparación entre la técnica misionera y la de expansión. Memoria usada.

λ_{ant} y δ_l aumentan. Durante la búsqueda, la técnica de expansión aumenta la región explorada y los rastros de feromona de los pasos previos guían la construcción de las nuevas soluciones parciales. Cuando δ_l es bajo, el algoritmo es capaz de encontrar buenas soluciones parciales porque la expansión es lenta. Por el contrario, la técnica misionera no se guía por los rastros de feromona de las etapas previas y puede alejarse de los caminos óptimos. Cuando δ_l aumenta en la técnica de expansión, ésta explora regiones mayores del grafo y, en este caso, los rastros de feromona hacen que el algoritmo explore regiones que ya han sido exploradas anteriormente, con la consecuente pérdida de recursos. La técnica misionera se comporta mejor en esta situación, ya que se concentra en regiones más prometedoras del grafo de construcción.

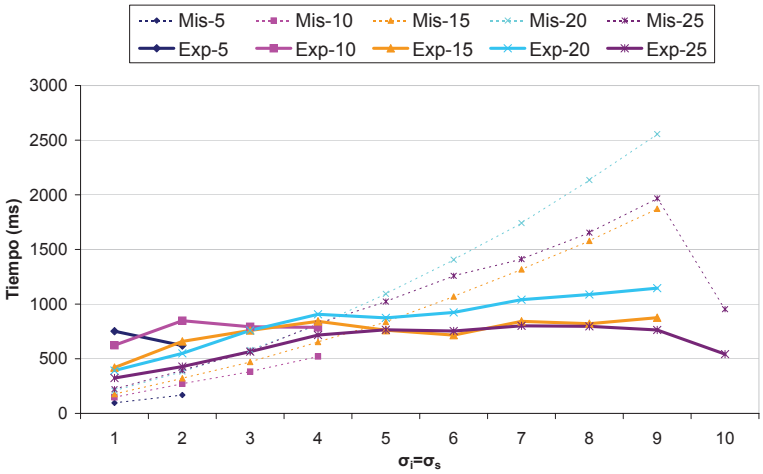


Figura A.11: Comparación entre la técnica misionera y la de expansión. Tiempo de CPU requerido.

Analizando la longitud de las trazas de error encontradas (Figura A.9) observamos que las dos técnicas están igualadas. Podemos apreciar sólo una ventaja (estadísticamente significativa) de la técnica de expansión para valores bajos de σ_i. Como dijimos en el párrafo anterior, cuando λ_{ant} aumenta en la técnica de expansión, se explora una región mayor del grafo de construcción. Esto da la oportunidad al algoritmo de encontrar trazas de error cortas. En el caso de la técnica misionera, las diferentes etapas explotan una región de tamaño similar pero a diferentes profundidades y, por esto, es más probable encontrar trazas de error más largas. No obstante, este comportamiento sólo se aprecia con bajos valores de σ_i y σ_s.

Centrémonos en los recursos (Figuras A.10 y A.11). La técnica de expansión requiere más memoria que la misionera. Esta observación está apoyada por los tests estadísticos. Una razón para este comportamiento es que la técnica de expansión trabaja con caminos más largos construidos por las hormigas. La segunda razón principal es que en la técnica misionera los rastros de feromona son eliminados al final de cada etapa. Por esto, concluimos que la técnica misionera es más eficiente que la de expansión con respecto al consumo de memoria. En el caso del tiempo de CPU requerido, encontramos un comportamiento diferente en ambas técnicas. Para bajos valores de σ_i la técnica de expansión requiere más tiempo que la técnica misionera. No obstante, a partir de $\sigma_i = \sigma_s$ entre 3 y 5, esta tendencia cambia: la técnica misionera requiere más tiempo que la de expansión. Esta observación está apoyada por los tests estadísticos (véase la Sección B.3). Pensamos que la principal razón para este comportamiento es la caída en la tasa de éxito de la técnica de expansión. Los valores medios mostrados en las figuras se calculan considerando sólo las ejecuciones en las que se encontró alguna traza de error. La reducción en la tasa de éxito para la técnica de expansión con respecto a la técnica misionera para valores altos de σ_i, indica que algunas ejecuciones requieren más iteraciones que las máximas permitidas (10) para encontrar una traza de error. Estas ejecuciones sin éxito no se tienen en cuenta para calcular los valores medios y, por esta razón, obtenemos un valor más bajo e irreal de los recursos requeridos. Sólo cuando la tasa de éxito es del 100 %, los valores medios son fiables. Esto explica el salto en la memoria y el tiempo cuando $\sigma_i = \sigma_s = 10$.

La conclusión final de este experimento es que la técnica misionera con valores intermedios para σ_s es la mejor opción para la búsqueda. Esta configuración constituye el mejor compromiso entre calidad de la solución, eficacia y recursos requeridos (eficiencia).

A.2.6. Análisis de escalabilidad

En el siguiente experimento estudiamos la escalabilidad del algoritmo. En el modelo, variamos el número de filósofos de 2 a 40 y observamos la longitud de las trazas de error, la memoria y el tiempo requerido por el algoritmo. Seguimos las recomendaciones comentadas en la sección previa y usamos la técnica misionera con $s = 2$ y $\sigma_s = 2$. Para la longitud máxima de los caminos de las hormigas usamos la expresión $\lambda_{ant} = \lceil 1.25n \rceil$. Usando esta expresión obtuvimos una tasa de éxito del 100 % en todos los casos[1]. El resto de los parámetros son los de la Tabla A.7. En la Tabla A.21 y las Figuras A.12, A.13 y A.14 mostramos los resultados.

Tabla A.21: Resultados de escalabilidad. Número de filósofos de 2 a 40.

n	Longitud	Memoria (KB)	Tiempo (ms)
2	3.00 0.00	1909.00 0.00	5.80 4.94
4	5.00 0.00	2003.16 43.99	8.30 4.01
6	7.16 0.97	2401.44 146.60	12.80 5.84
8	9.84 1.98	3084.40 391.36	18.20 7.67
10	13.60 3.68	4325.40 644.85	35.00 12.37
12	17.36 4.49	5906.52 776.96	58.00 16.31
14	20.56 4.86	7835.92 917.53	93.30 20.98
16	25.56 7.48	9697.73 1303.92	133.60 31.32
18	28.48 9.39	11509.30 1599.74	185.10 51.23
20	34.84 9.79	14291.11 1266.33	272.80 54.61
22	40.48 12.36	17142.51 2372.52	387.00 83.64
24	46.28 12.04	20664.32 964.74	512.00 60.43
26	53.12 12.88	24248.32 1012.47	685.90 74.97
28	54.08 13.96	27777.01 2405.05	870.60 124.42
30	61.44 13.13	32194.56 1745.73	1123.10 182.92
32	67.20 17.30	36177.92 1411.90	1382.20 145.74
34	69.24 17.14	41052.16 1738.36	1735.30 171.40
36	76.64 19.48	45895.68 1048.09	2097.70 162.92
38	83.36 18.42	51251.20 2282.28	2561.50 251.27
40	88.44 19.41	56545.28 1069.87	3145.20 185.79

Observamos que la longitud media de las trazas de error crece de forma casi lineal (Figura A.12). Necesitamos estudiar más valores de n para tener una idea de qué tipo de curva sigue la longitud de las trazas de error (exponencial, potencial, etc.). Por otro lado, los gráficos de recursos (memoria y tiempo de CPU) siguen un crecimiento de tipo exponencial (Figuras A.13 y A.14). Esta es una consecuencia directa del crecimiento exponencial del grafo de construcción.

A.2.7. Análisis de la influencia de la heurística

En esta sección estudiamos la influencia de la información heurística en los resultados. Esta información se usa durante la fase de construcción para seleccionar el siguiente nodo a visitar (véase la Sección 4.4.3). Para determinar la importancia de la información heurística en la fase de construcción,

[1]Esto no es una regla general, simplemente funcionó con el modelo de los filósofos usado en los experimentos.

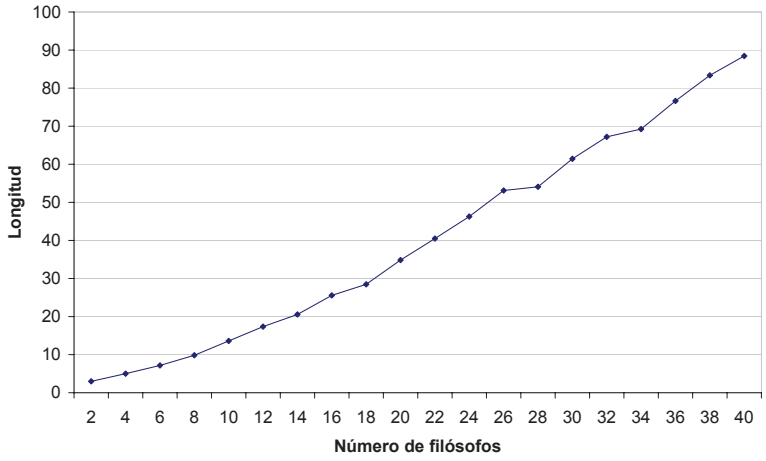


Figura A.12: Resultados de escalabilidad. Longitud de las trazas de error.

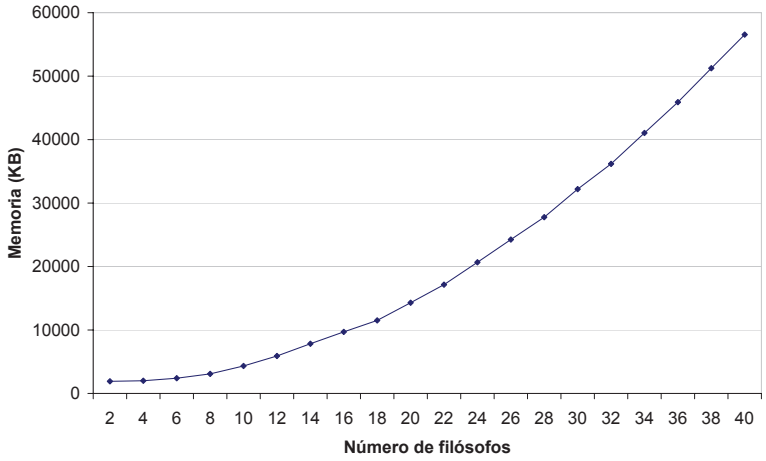


Figura A.13: Resultados de escalabilidad. Memoria requerida.

estudiamos aquí distintos valores para el parámetro β. El resto de parámetros son los de la Tabla A.7. En la Tabla A.22 mostramos la tasa de éxito, la longitud de las trazas de error, la memoria requerida y el tiempo de CPU requerido cuando β cambia.

Como podemos observar a partir de los resultados, el uso de información heurística es beneficioso para la búsqueda. Cuando $\beta > 0$, aumenta la probabilidad de encontrar un camino hasta el nodo objetivo (tasa de éxito más alta) sin que lo haga la longitud de las trazas de error, así que la calidad de las soluciones no disminuye (las diferencias en los valores no son estadísticamente significativas). La influencia de la heurística en el consumo de memoria y el tiempo de CPU es pequeña pero real: los tests estadísticos confirman que el uso de información heurística reduce los recursos requeridos. No obstante, no hay diferencias estadísticamente significativas cuando $\beta \geq 1$. También observamos que la tasa de éxito no

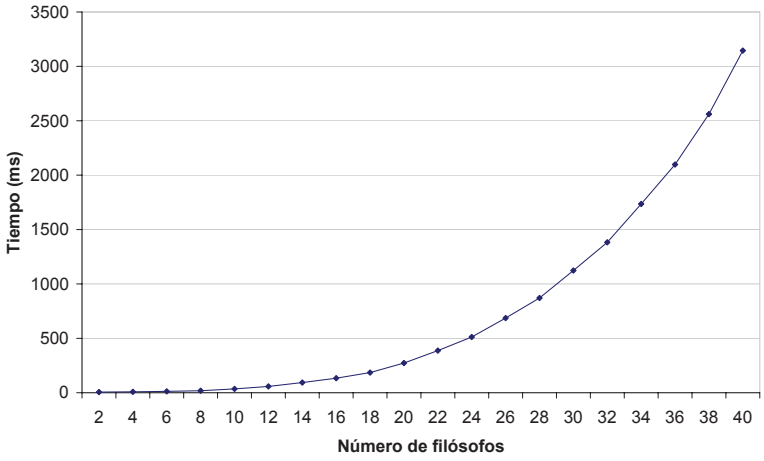


Figura A.14: Resultados de escalabilidad. Tiempo de CPU requerido.

Tabla A.22: Influencia de β en los resultados.

β	Tasa de éxito	Longitud	Memoria (KB)	Tiempo (ms)
0	9.00	$35.22_{7.08}$	$8645.00_{78.84}$	$256.67_{32.66}$
1	60.00	$36.20_{7.82}$	$8515.60_{105.78}$	$217.33_{20.97}$
2	61.00	$37.72_{8.80}$	$8486.77_{133.66}$	$218.85_{20.57}$
3	58.00	$35.14_{7.80}$	$8454.03_{118.47}$	$210.17_{19.96}$

aumenta cuando β es mayor que uno. En conclusión, podemos afirmar que la información heurística es una importante ayuda para las hormigas durante la fase de construcción y debe ser usada para conseguir una alta tasa de éxito. No obstante, un valor de $\beta = 1$ es suficiente (al menos en este modelo) para conseguir el máximo beneficio de la información heurística.

A.2.8. Análisis de las penalizaciones

En esta sección queremos analizar la influencia en los resultados de las dos constantes de penalización utilizadas en la función de *fitness* y discutidas en la Sección 5.3.3: la constante de penalización para soluciones parciales, p_p, y la constante de penalización para ciclos, p_c. En este experimento el algoritmo se detiene cuando se alcanza el número máximo de iteraciones y no cuando se encuentra una solución. Hacemos esto para apreciar la influencia de p_p en los resultados (si tiene alguna). Si permitiéramos al algoritmo detenerse cuando encontrase la primera solución, la mayoría de los caminos trazados por las hormigas habrían sido penalizados durante la búsqueda y sólo unos pocos caminos no lo estarían. Como consecuencia, no habría influencia de p_p en los resultados. Estudiamos diferentes valores para p_p y p_c y mostramos la tasa de éxito, la longitud de las trazas de error, la memoria requerida y el tiempo de CPU requerido por el algoritmo en la Tablas A.23, A.24, A.25 y A.26, respectivamente.

A partir de las tablas podemos observar que no hay influencia de p_p y p_c en ninguna medida. Un test estadístico confirma esta observación. Es decir, los valores de p_p y p_c no son relevantes cuando tratamos de encontrar el estado de interbloqueo en el modelo de los filósofos. Este es un resultado inesperado,

Tabla A.23: Análisis de las penalizaciones. Tasa de éxito.

p_p	p_c			
	0	10	100	1000
0	52	55	59	67
10	62	61	56	61
100	60	59	63	54
1000	61	58	60	63

Tabla A.24: Análisis de las penalizaciones. Longitud de las trazas de error.

p_p	p_c			
	0	10	100	1000
0	$35.38_{8.05}$	$35.84_{7.62}$	$35.71_{8.35}$	$32.28_{8.46}$
10	$35.13_{7.83}$	$35.56_{8.45}$	$33.71_{7.37}$	$35.03_{8.53}$
100	$34.67_{7.74}$	$34.63_{7.94}$	$34.21_{7.29}$	$34.33_{8.04}$
1000	$33.20_{7.22}$	$34.79_{7.87}$	$35.47_{7.87}$	$35.60_{7.55}$

Tabla A.25: Análisis de las penalizaciones. Memoria reqerida.

p_p	p_c			
	0	10	100	1000
0	$8485.38_{93.40}$	$8497.36_{121.38}$	$8498.42_{103.66}$	$8489.00_{120.28}$
10	$8493.19_{138.61}$	$8484.48_{102.32}$	$8495.14_{130.51}$	$8481.52_{132.64}$
100	$8491.00_{122.14}$	$8475.51_{110.60}$	$8502.52_{103.38}$	$8469.30_{146.70}$
1000	$8493.85_{109.56}$	$8468.38_{146.03}$	$8509.27_{125.31}$	$8461.32_{126.15}$

Tabla A.26: Análisis de las penalizaciones. Tiempo de CPU requerido.

p_p	p_c			
	0	10	100	1000
0	$237.31_{15.70}$	$235.45_{13.73}$	$235.93_{13.91}$	$233.73_{14.64}$
10	$234.03_{14.42}$	$238.69_{13.36}$	$234.46_{14.87}$	$236.72_{17.25}$
100	$238.00_{11.08}$	$235.76_{13.05}$	$237.30_{14.28}$	$235.37_{11.17}$
1000	$236.72_{15.33}$	$234.66_{13.03}$	$241.67_{14.96}$	$233.49_{14.60}$

ya que las penalizaciones se usan para guiar la búsqueda de caminos más prometedores en el grafo de construcción. Debido a esto, esperábamos, al menos, un aumento en la tasa de éxito y una reducción en la longitud de las trazas de error cuando se usaran penalizaciones más altas. Sin embargo, en este modelo particular no ocurre esto. La longitud máxima de los caminos de las hormigas λ_{ant} es baja (10) y esto podría explicar por qué no hay influencia de p_c, ya que es relativamente fácil para el algoritmo encontrar soluciones sin ciclos. La inexistencia de influencia de p_p quizá pueda explicarse por el bajo número de pasos del algoritmo o por la escasa complejidad de este modelo. No obstante, son necesarios más experimentos para comprender completamente cuál es el motivo de esta ausencia de influencia.

Apéndice B

Validación estadística de resultados

En este apéndice incluimos los tests estadísticos realizados en la tesis doctoral para todos los experimentos. Ésta es una práctica muy importante que los investigadores del dominio de las metaheurísticas, y los algoritmos no estocásticos en general, deben incluir en sus trabajos. Hoy en día, los autores que no hacen tests estadísticos puede mostrar "claras" ventajas para sus propuestas basadas en mejoras numéricas bastante insignificantes.

En cada caso, el procedimiento para generar la información estadística presentada en las tablas y figuras es el siguiente. Primero se aplica el test de Kolmogorov-Smirnov para comprobar si las variables aleatorias son normales o no y el test de Levene para comprobar la homocedasticidad de las muestras (igualdad de varianzas). Si las muestras pasan ambos tests (son normales y tienen varianzas iguales), se realiza un análisis de varianza ANOVA 1. En caso contrario, se aplica el test de Kruskal-Wallis, siempre con un nivel de confianza del 95 % ($\alpha = 0.05$). Tras esto, se aplica un test de comparaciones múltiples cuyos resultados son los que mostramos en este apéndice.

Cuando la comparación se realiza entre dos muestras, generalmente mostramos los resultados del test en tablas y usamos los signos + y − para indicar si hay, o no, diferencia significativa entre ellas, respectivamente. Si la comparación se hace entre tres o más muestras, utilizaremos la representación compacta de la Figura B.1, donde vemos una comparación múltiple de ejemplo con cinco muestras. La celda (i, j) de la figura es negra si la diferencia entre las muestras i y j es estadísticamente significativa. Es decir, un cuadrado negro en la posición $(2, 3)$ indica que las distribuciones de las muestras 2 y 3 tienen estadísticos de localización (medias o medianas) diferentes con confianza estadística.

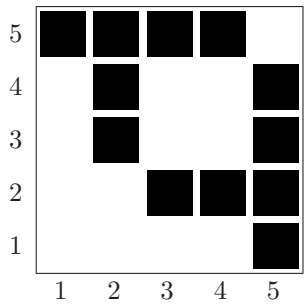


Figura B.1: Un ejemplo de comparación múltiple.

B.1. Planificación de proyectos software

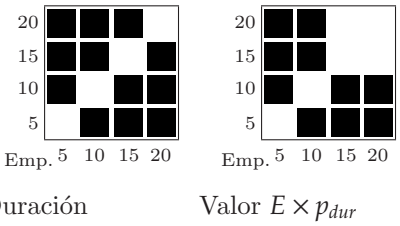


Figura B.2: Resultados del test estadístico comparando los resultados del GA para escenarios con diferente número de empleados (Tabla 6.2).

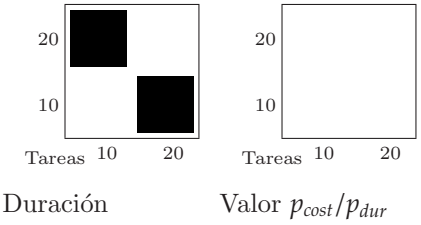


Figura B.3: Resultados del test estadístico comparando los resultados del GA para escenarios con diferente número de tareas (Tabla 6.3).

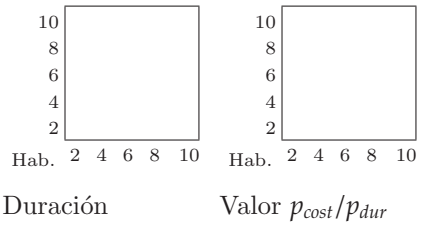


Figura B.4: Resultados del test estadístico comparando los resultados del GA para escenarios con diferente número de habilidades por empleado (Tabla 6.4).

B.2. Generación de casos de prueba

Tabla B.1: Resultados del test estadístico para la comparativa entre dES y ES (Tabla 7.5).

Programa	Cobertura	Evaluaciones	Tiempo
`triangle`	-	+	+
`gcd`	-	+	+
`calday`	-	-	+
`crc`	-	-	-
`insertion`	-	-	+
`shell`	-	-	-
`quicksort`	-	-	+
`heapsort`	-	-	-
`select`	-	+	+
`bessel`	-	+	+
`sa`	-	-	+
`netflow`	-	+	+

Tabla B.2: Resultados del test estadístico para la comparativa entre dGA y GA (Tabla 7.6).

Programa	Cobertura	Evaluaciones	Tiempo
`triangle`	-	-	+
`gcd`	-	+	-
`calday`	-	+	+
`crc`	-	-	-
`insertion`	-	-	-
`shell`	-	-	-
`quicksort`	-	-	+
`heapsort`	-	-	-
`select`	-	+	+
`bessel`	-	-	+
`sa`	-	-	+
`netflow`	-	-	+

Tabla B.3: Resultados del test estadístico para los diferentes modos de búsqueda en dES (Tabla 7.8).

Modos de búsqueda	Cobertura	Evaluaciones	Tiempo
same vs. *diff*	+	+	+

Tabla B.4: Resultados del test estadístico para los diferentes criterios de parada en dES (Tabla 7.9).

Criterios de parada	Cobertura	Evaluaciones	Tiempo
obj vs. *any*	-	-	-

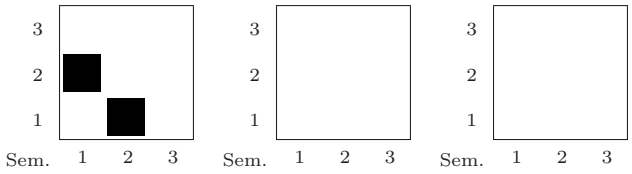


Figura B.5: Resultados del test estadístico para las diferentes semillas en dES (Tabla 7.10).

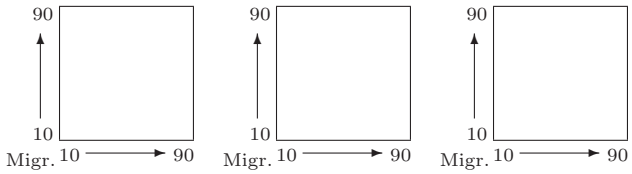


Figura B.6: Resultados del test estadístico para el periodo de migración en dES (Tabla 7.11).

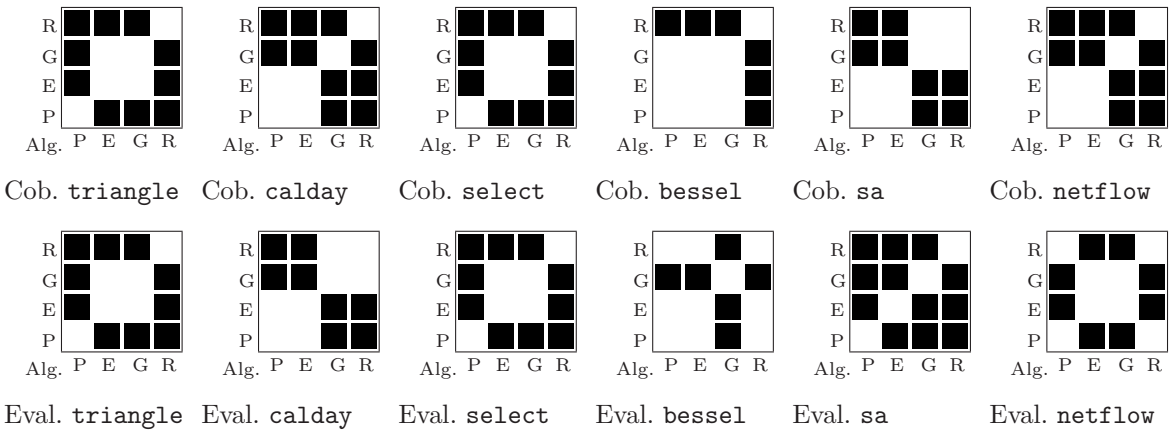


Figura B.7: Resultados del test estadístico comparando los algoritmos PSO, ES, GA y RND (Tabla 7.12).

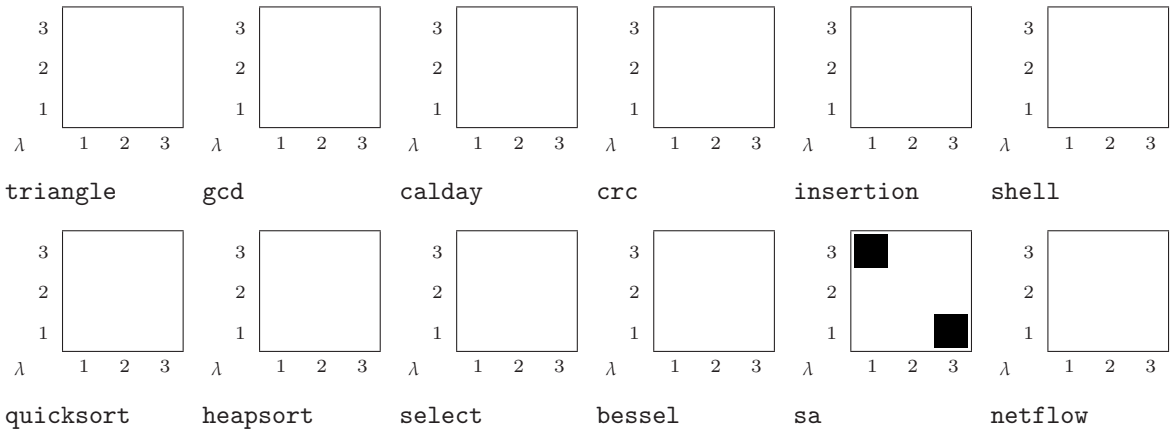


Figura B.8: Resultados del test estadístico para la influencia de λ en la cobertura de ES (Tabla A.1).

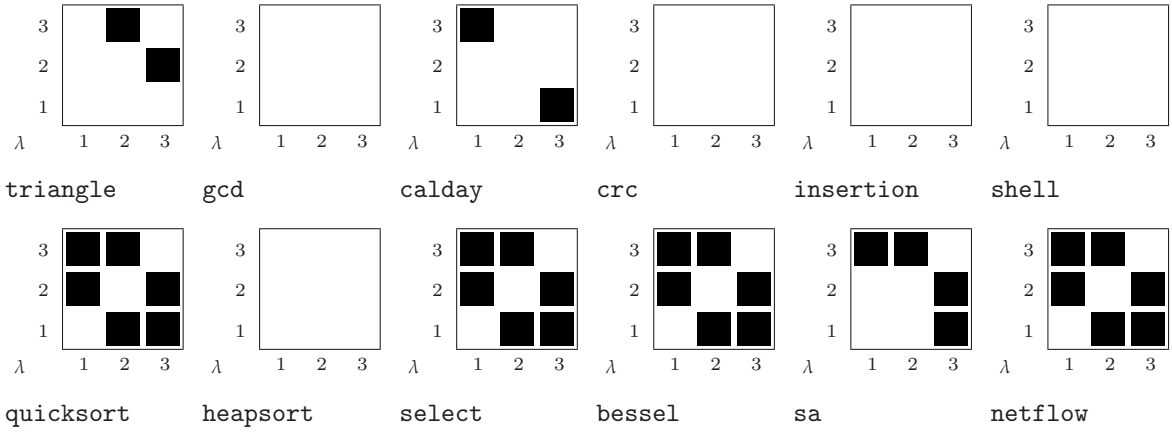


Figura B.9: Resultados del test estadístico para la influencia de λ en las evaluaciones de ES (Tabla A.1).

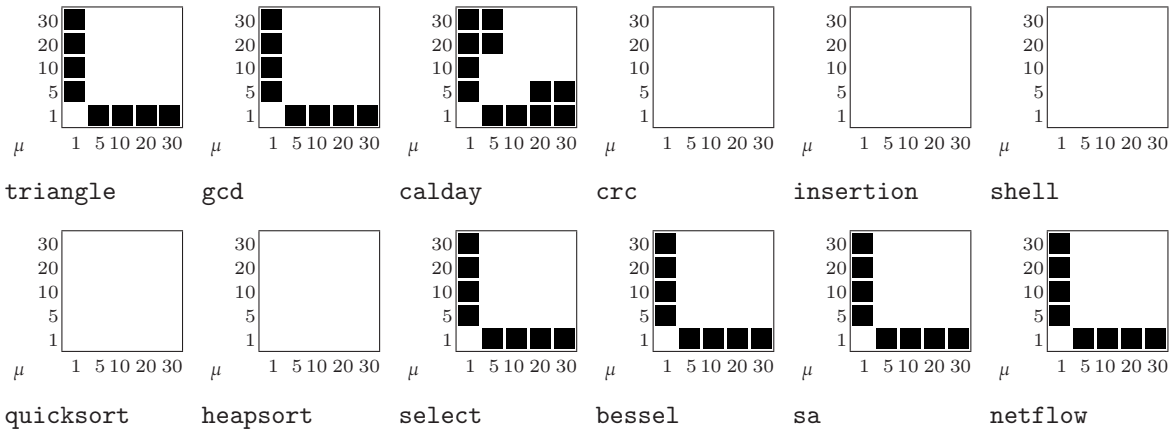


Figura B.10: Resultados del test estadístico para la influencia de μ en la cobertura de ES (Tabla A.2).

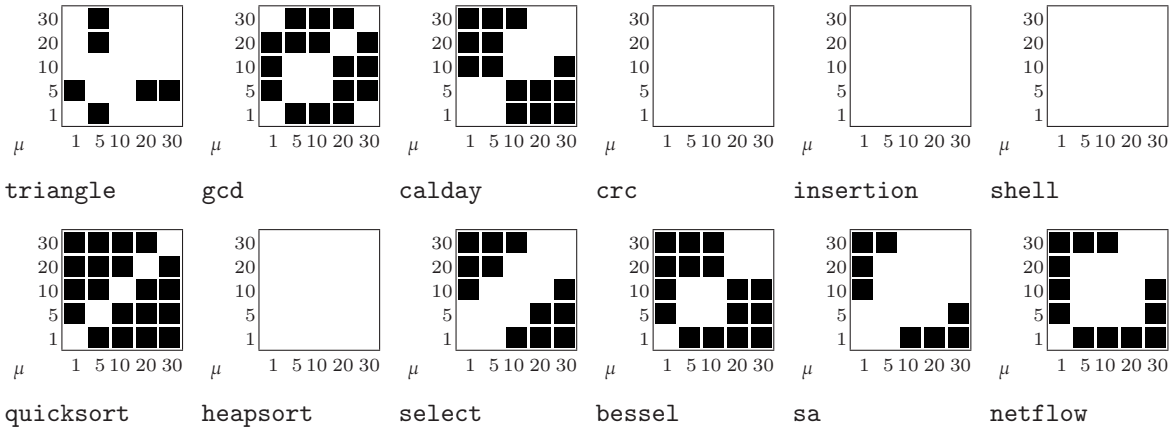


Figura B.11: Resultados del test estadístico para la influencia de μ en las evaluaciones de ES (Tabla A.2).

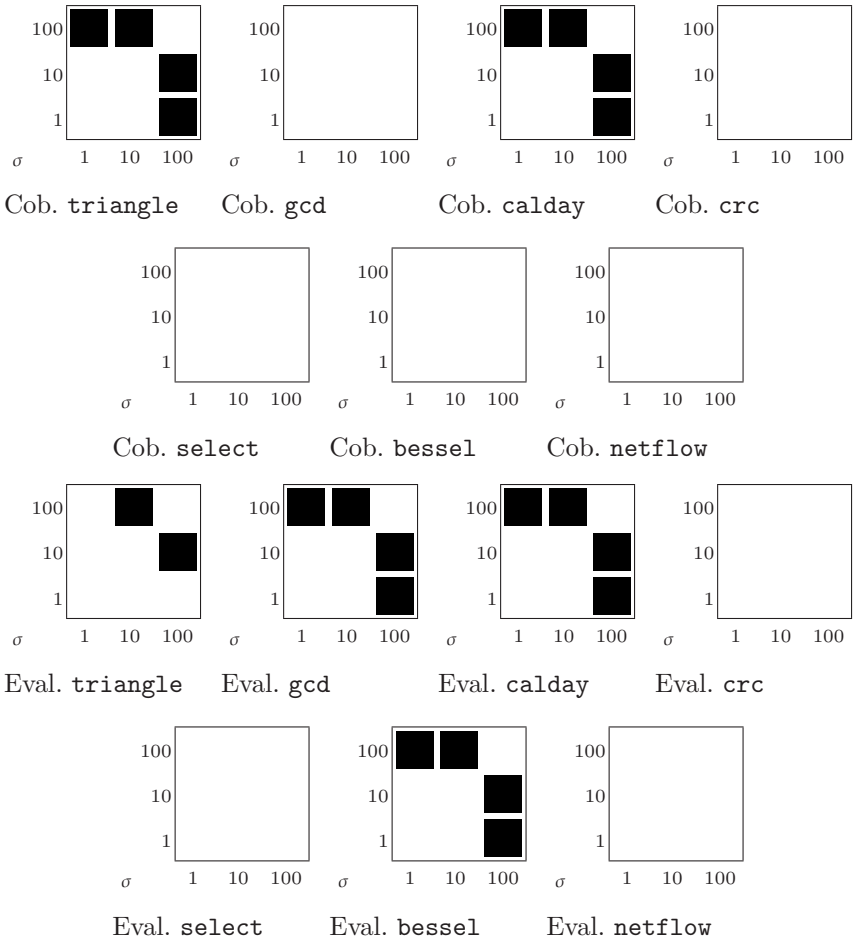


Figura B.12: Resultados del test estadístico para el estudio de la influencia de la desviación estándar de la mutación σ en la cobertura y el número de evaluaciones obtenidas por GA (Tabla A.3).

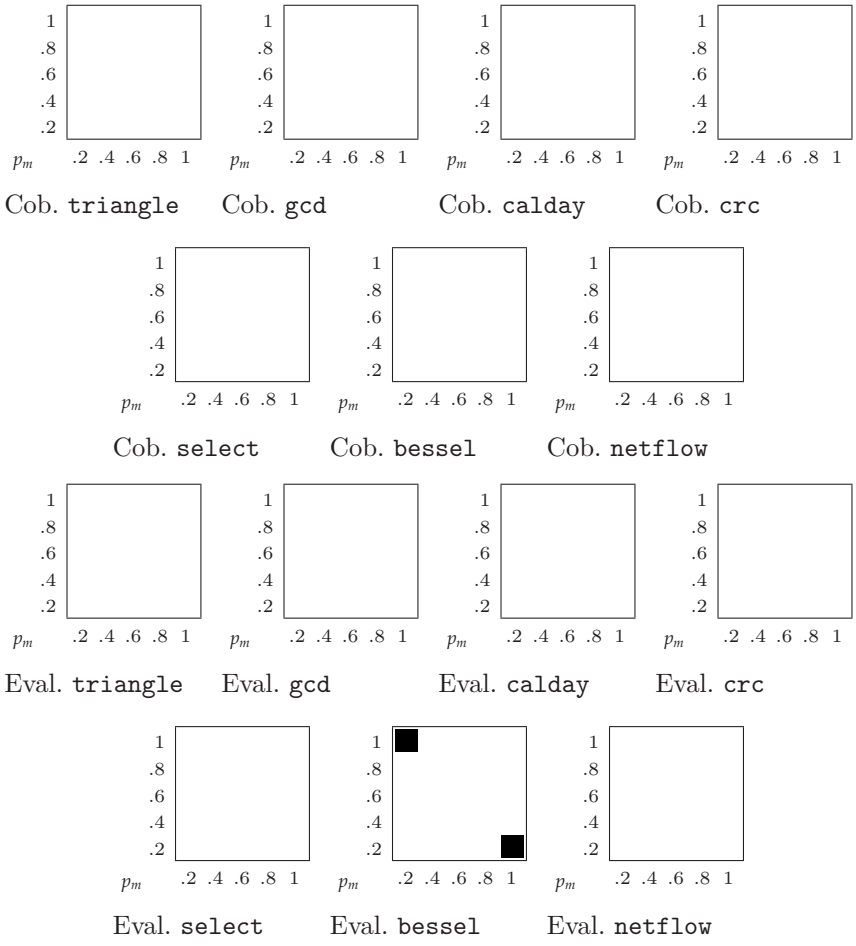


Figura B.13: Resultados del test estadístico para el estudio de la influencia de la probabilidad de mutación p_m en la cobertura y el número de evaluaciones obtenidas por GA (Tabla A.4).

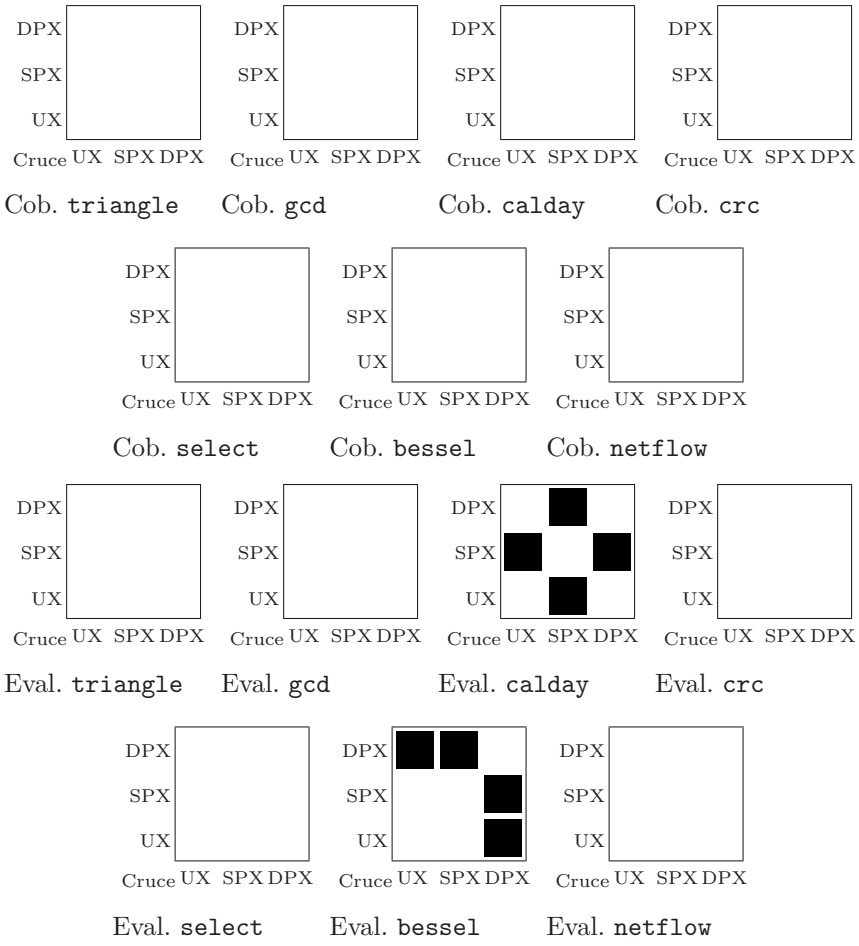


Figura B.14: Resultados del test estadístico para el estudio de la influencia del operador de cruce en la cobertura y el número de evaluaciones obtenidas por GA (Tabla A.5).

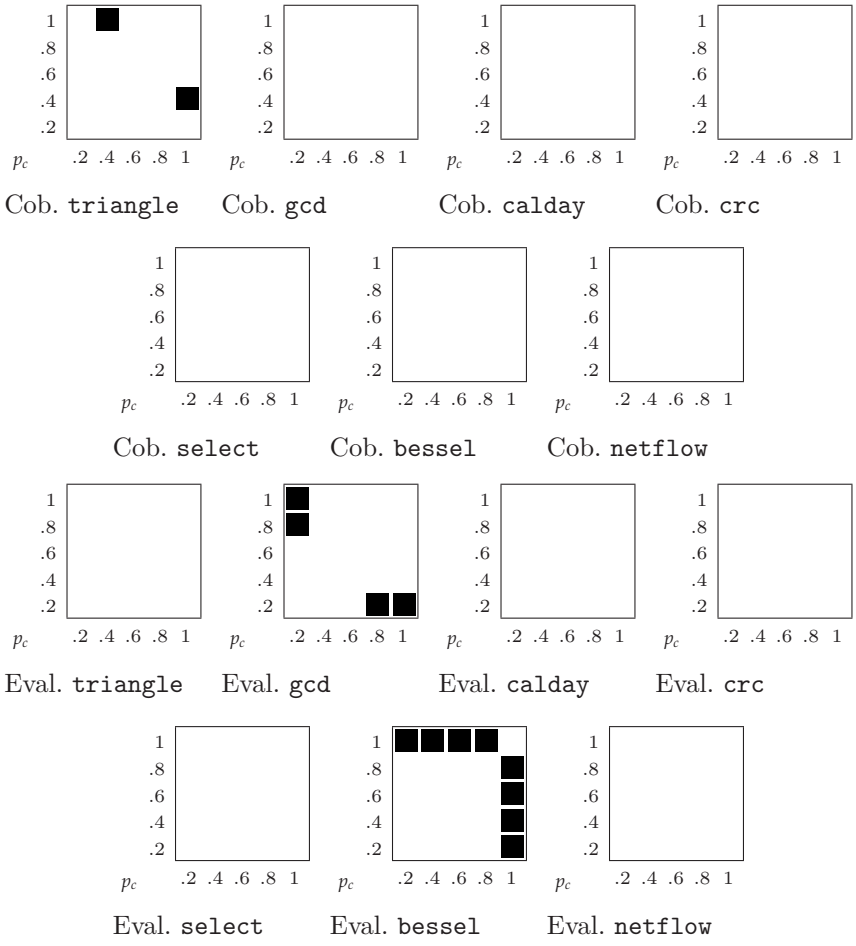


Figura B.15: Resultados del test estadístico para el estudio de la influencia de la probabilidad de cruce p_c en la cobertura y el número de evaluaciones obtenidas por GA (Tabla A.6).

B.3. Búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes

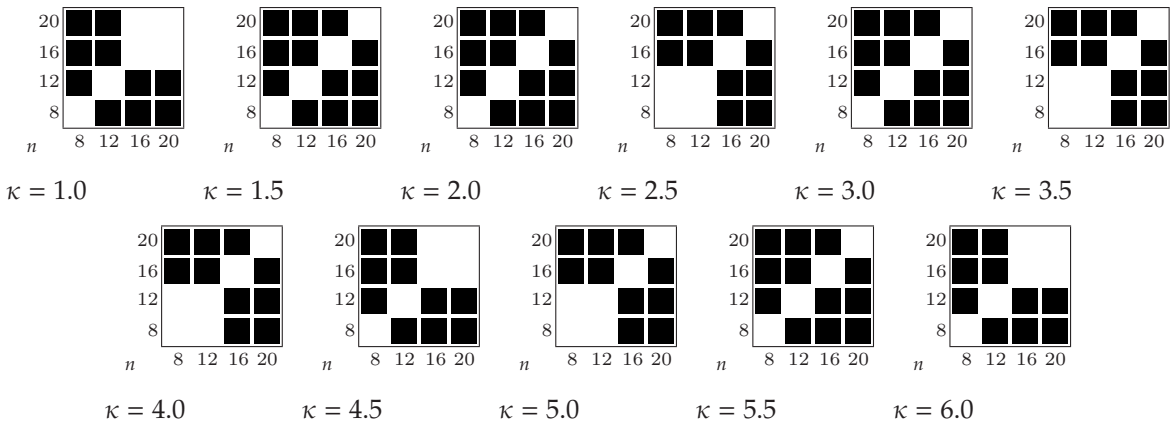


Figura B.16: Longitud de las trazas de error para distinto número de filósofos n (Tabla 8.4).

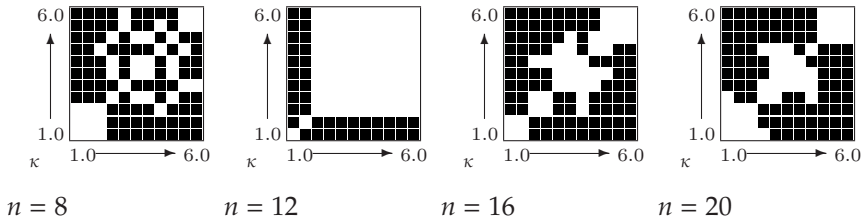


Figura B.17: Longitud de las trazas de error para distinto valor de κ (Tabla 8.4).

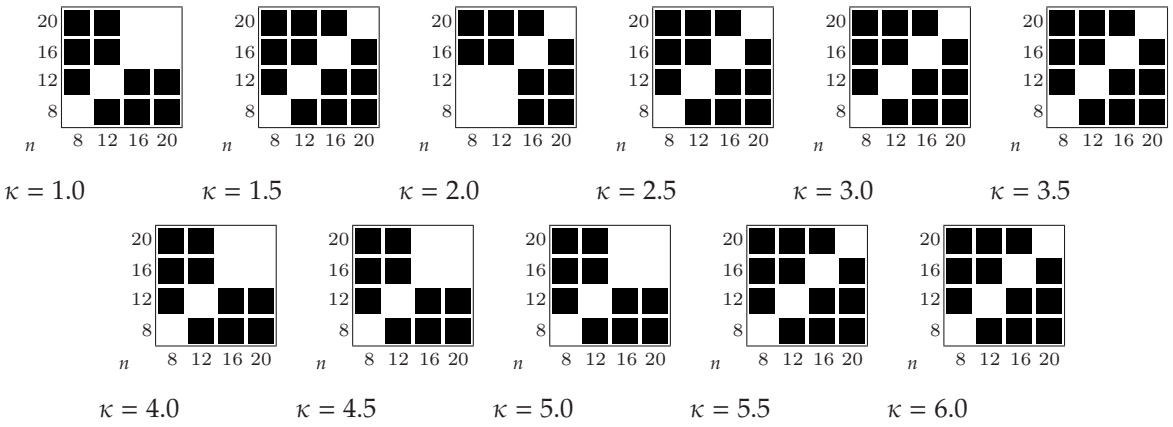


Figura B.18: Memoria requerida para distinto número de filósofos n (Figura 8.1).

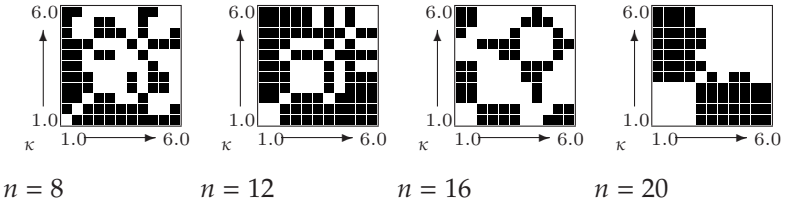


Figura B.19: Memoria requerida para distinto valor de κ (Figura 8.1).

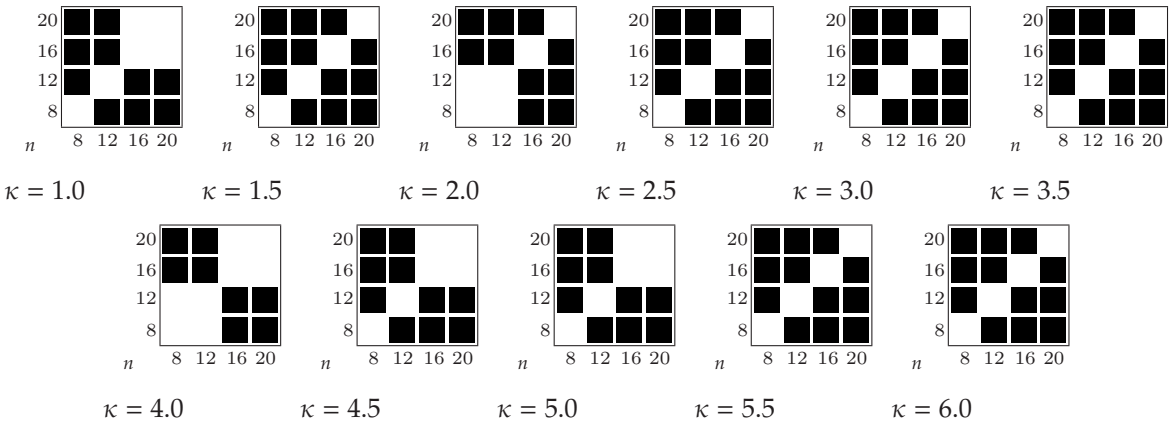


Figura B.20: Tiempo requerido para distinto número de filósofos n (Figura 8.2).

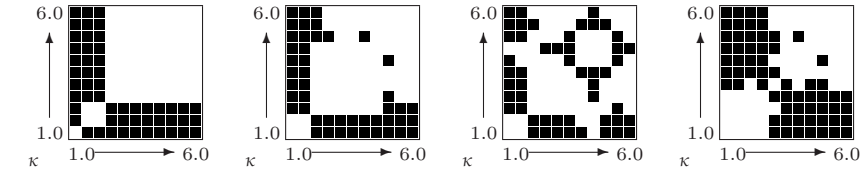


Figura B.21: Tiempo requerido para distinto valor de κ (Figura 8.2).

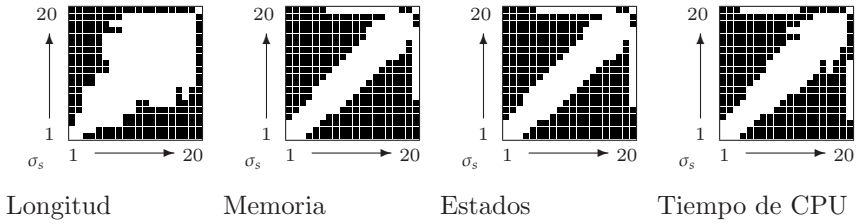


Figura B.22: Influencia de σ_s en los resultados de la técnica misionera (Tabla 8.5).

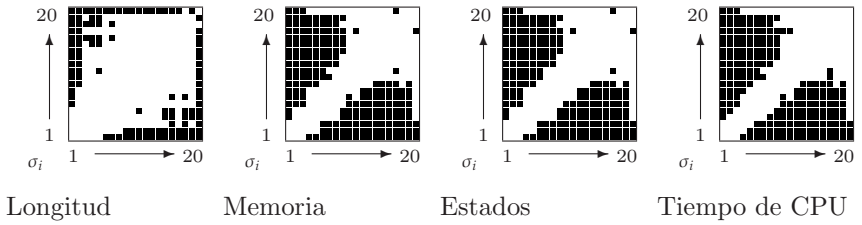


Figura B.23: Influencia de σ_i en los resultados de la técnica de expansión (Tabla 8.6).

Tabla B.5: Comparación entre ACOhg y los algoritmos exactos (Tablas 8.7 y 8.8). En este caso se ha usado el test de Wilcoxon para una muestra. Comparamos siempre un ACOhg con un algoritmo exacto.

Modelos	Medidas	ACOhg-b		ACOhg-h	
		BFS	DFS	A*	BF
garp	Long. (estados)	+	+	+	+
	Mem. (KB)	+	+	+	+
	Estados exp.	+	+	+	+
	Tiempo (ms)	+	+	+	+
giop22	Long. (estados)	•	+	-	-
	Mem. (KB)	•	+	+	+
	Estados exp.	•	+	+	+
	Tiempo (ms)	•	+	+	+
leader6	Long. (estados)	+	+	+	+
	Mem. (KB)	+	+	+	+
	Estados exp.	+	+	+	+
	Tiempo (ms)	+	+	+	+
lynch	Long. (estados)	+	+	+	+
	Mem. (KB)	+	+	+	+
	Estados exp.	+	+	+	+
	Tiempo (ms)	+	+	+	+
marriers4	Long. (estados)	•	•	•	+
	Mem. (KB)	•	•	•	+
	Estados exp.	•	•	•	+
	Tiempo (ms)	•	•	•	+
needham	Long. (estados)	+	+	+	+
	Mem. (KB)	+	+	+	+
	Estados exp.	+	+	+	+
	Tiempo (ms)	+	+	+	+
phi16	Long. (estados)	•	•	+	+
	Mem. (KB)	•	•	+	+
	Estados exp.	•	•	+	+
	Tiempo (ms)	•	•	+	+
pots	Long. (estados)	+	+	+	+
	Mem. (KB)	+	+	+	-
	Estados exp.	+	+	+	+
	Tiempo (ms)	+	+	+	+
relay	Long. (estados)	+	+	+	+
	Mem. (KB)	+	+	+	+
	Estados exp.	+	+	+	+
	Tiempo (ms)	+	+	+	+
x509	Long. (estados)	+	+	+	+
	Mem. (KB)	+	+	+	+
	Estados exp.	+	+	+	+
	Tiempo (ms)	+	+	+	+

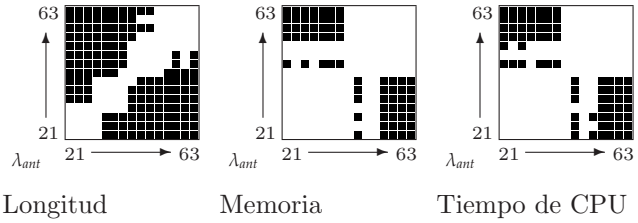


Figura B.24: Influencia de λ_{ant} en los resultados (Tabla A.8).

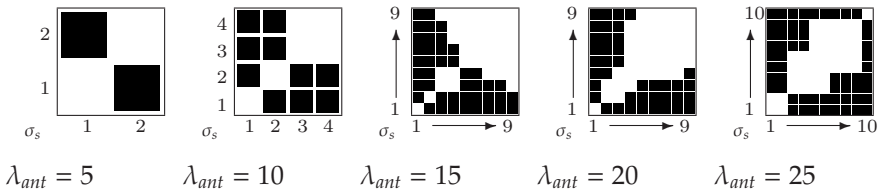


Figura B.25: Técnica misionera: influencia de σ_s en la longitud de las trazas de error (Tabla A.10).

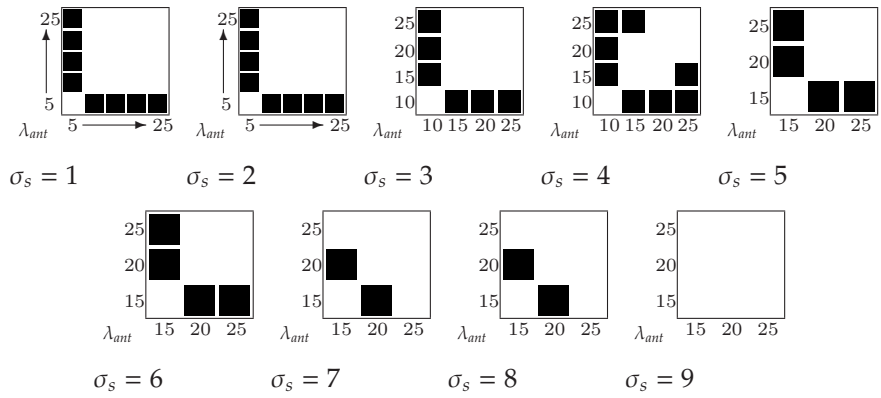


Figura B.26: Técnica misionera: influencia de λ_{ant} en la longitud de las trazas de error (Tabla A.10).

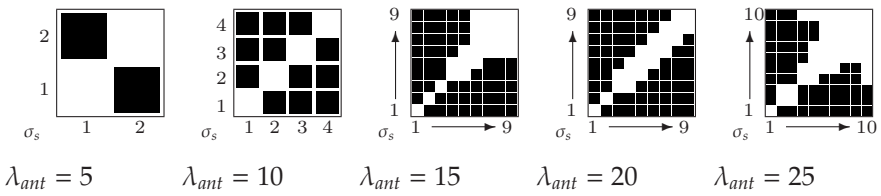


Figura B.27: Técnica misionera: influencia de σ_s en la memoria requerida (Tabla A.11).

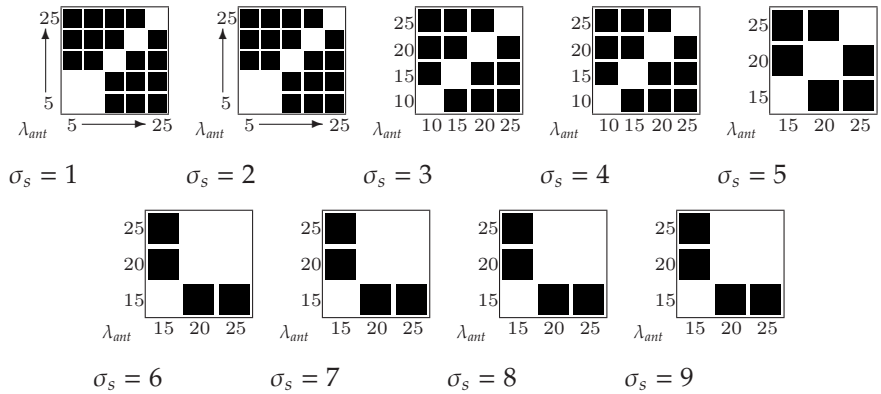


Figura B.28: Técnica misionera: influencia de λ_{ant} en la memoria requerida (Tabla A.11).

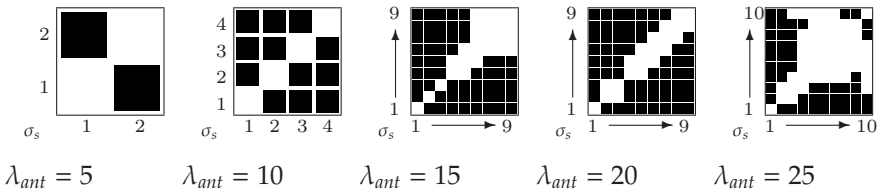


Figura B.29: Técnica misionera: influencia de σ_s en el tiempo de CPU requerido (Tabla A.12).

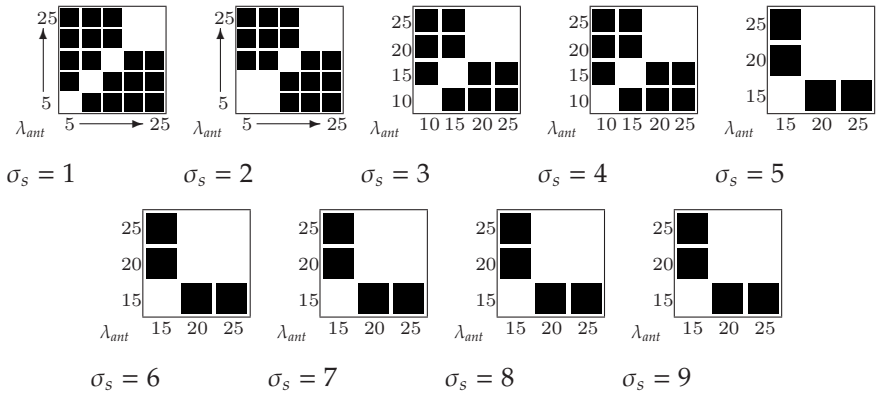


Figura B.30: Técnica misionera: influencia de λ_{ant} en el tiempo de CPU requerido (Tabla A.12).

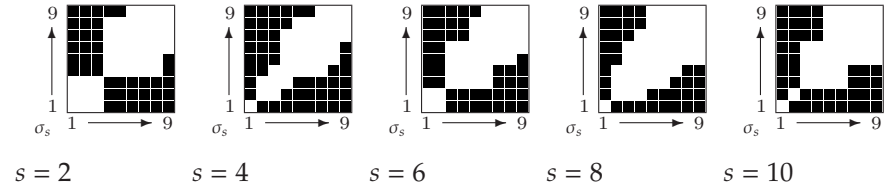


Figura B.31: Técnica misionera (soluciones guardadas): influencia de σ_s en la longitud de las trazas de error (Tabla A.14).

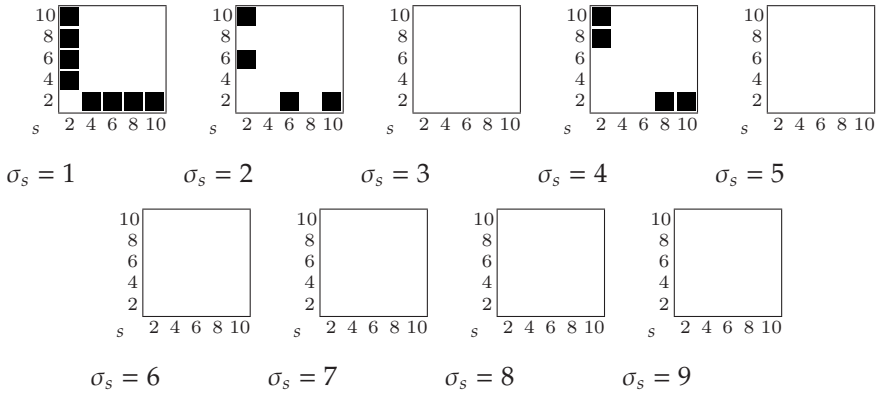


Figura B.32: Técnica misionera (soluciones guardadas): influencia de s en la longitud de las trazas de error (Tabla A.14).

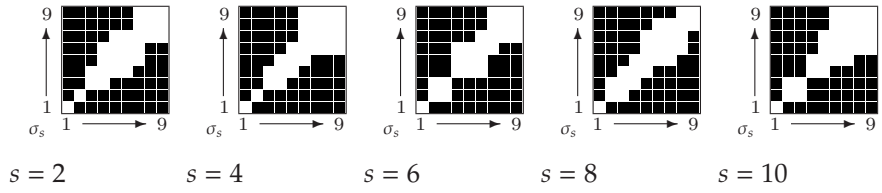


Figura B.33: Técnica misionera (soluciones guardadas): influencia de σ_s en la memoria requerida (Tabla A.15).

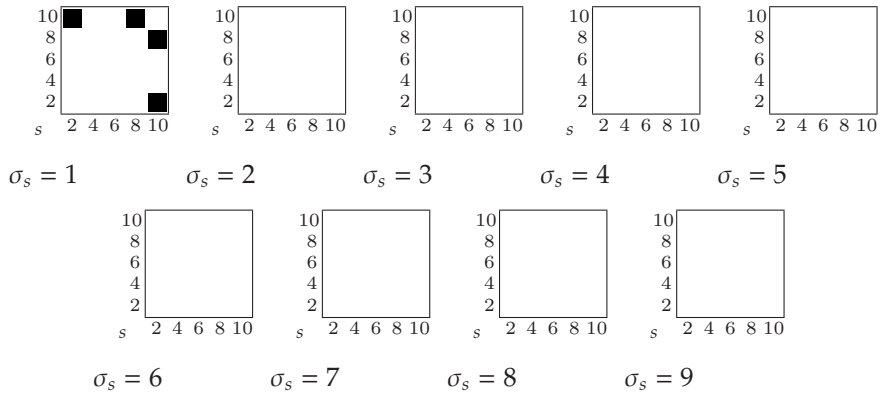


Figura B.34: Técnica misionera (soluciones guardadas): influencia de s en la memoria requerida (Tabla A.15).

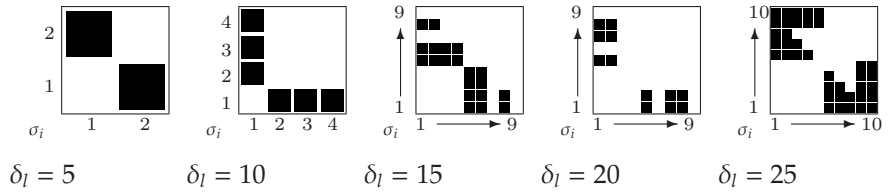


Figura B.35: Técnica de expansión: influencia de σ_i en la longitud de las trazas de error (Tabla A.18).

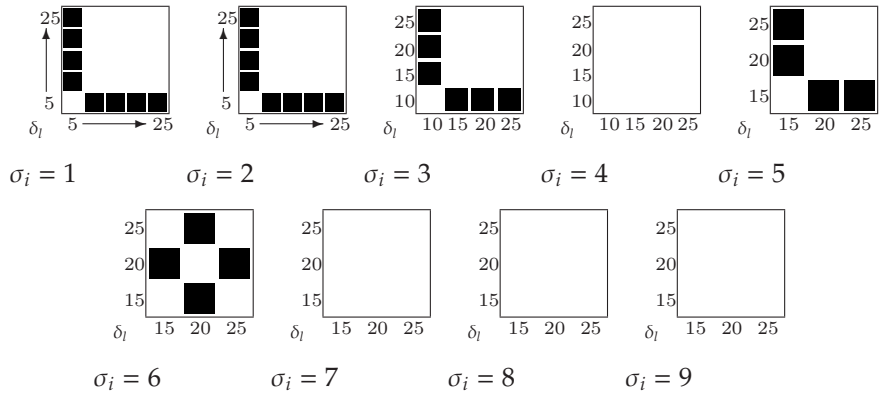


Figura B.36: Técnica de expansión: influencia de δ_l en la longitud de las trazas de error (Tabla A.18).

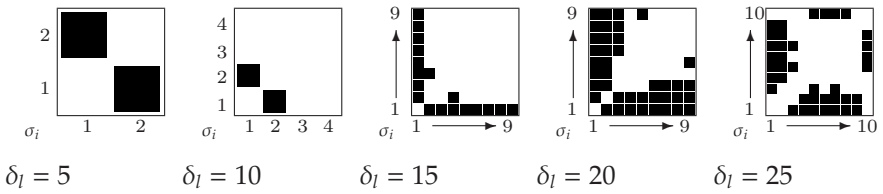


Figura B.37: Técnica de expansión: influencia de σ_i en la memoria requerida (Tabla A.19).

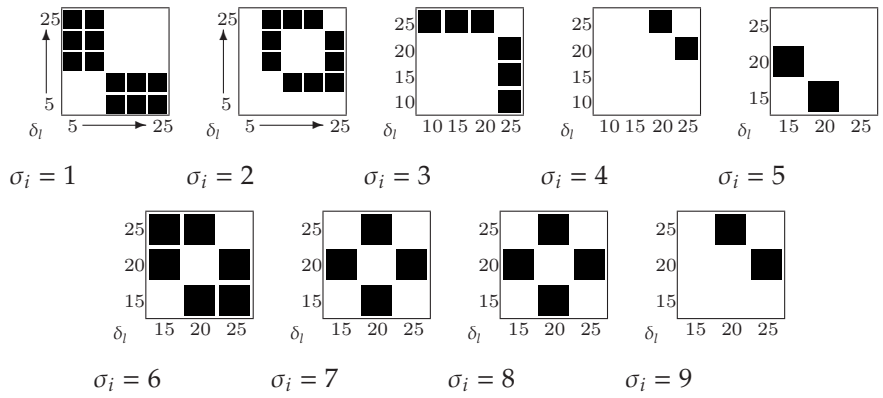


Figura B.38: Técnica de expansión: influencia de δ_l en la memoria requerida (Tabla A.19).

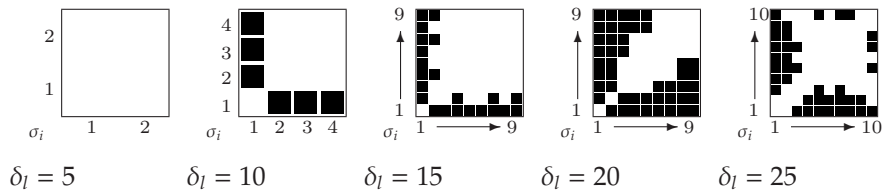


Figura B.39: Técnica de expansión: influencia de σ_i en el tiempo de CPU requerido (Tabla A.20).

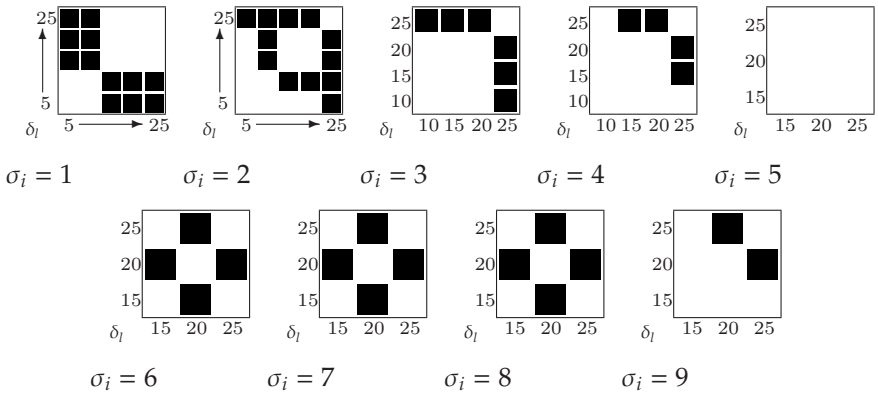


Figura B.40: Técnica de expansión: influencia de δ_l en el tiempo de CPU requerido (Tabla A.20).

Tabla B.6: Comparación entre la técnica misionera y la de expansión. Longitud de las trazas de error (Figura A.9).

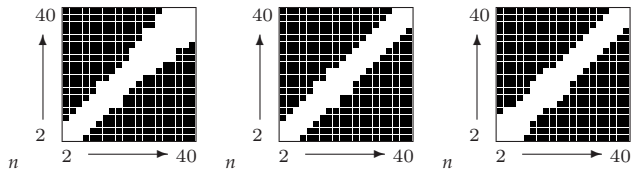
$\sigma_i = \sigma_s$	$\delta_l = \lambda_{ant}$				
	5	10	15	20	25
1	+	+	+	+	+
2	-	+	+	+	+
3	-	-	+	+	+
4	-	-	-	-	+
5	-	-	-	-	-
6	-	-	-	-	+
7	-	-	-	-	-
8	-	-	-	-	-
9	-	-	-	-	+
10	-	-	-	-	-

Tabla B.7: Comparación entre la técnica misionera y la de expansión. Memoria requerida (Figura A.10).

$\sigma_i = \sigma_s$	$\delta_l = \lambda_{ant}$				
	5	10	15	20	25
1	+	+	+	+	+
2	+	+	+	+	+
3	-	+	+	+	+
4	-	+	+	+	+
5	-	-	+	+	+
6	-	-	+	+	+
7	-	-	+	+	+
8	-	-	+	+	+
9	-	-	+	+	-
10	-	-	-	-	-

Tabla B.8: Comparación entre la técnica misionera y la de expansión. Tiempo de CPU (Figura A.11).

$\sigma_i = \sigma_s$	$\delta_l = \lambda_{ant}$				
	5	10	15	20	25
1	+	+	+	+	+
2	+	+	+	+	-
3	-	+	+	+	-
4	-	+	+	-	+
5	-	-	+	+	+
6	-	-	+	+	+
7	-	-	+	+	+
8	-	-	+	+	+
9	-	-	+	+	+
10	-	-	-	-	+



Longitud Memoria Tiempo de CPU

Figura B.41: Influencia del número de filósofos, n, en los resultados (Tabla A.21).

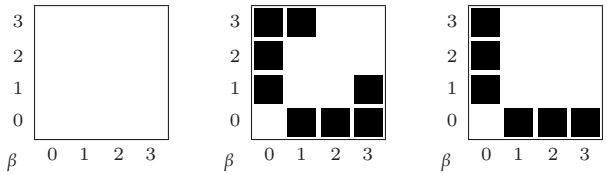


Figura B.42: Influencia de la potencia de la heurística, β, en los resultados (Tabla A.22).

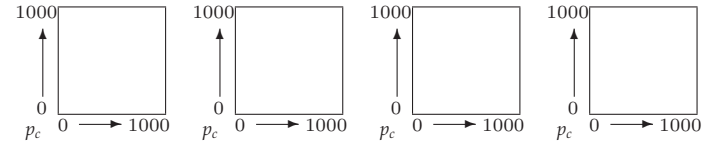


Figura B.43: Penalizaciones: influencia de p_c en la longitud de las trazas de error (Tabla A.24).

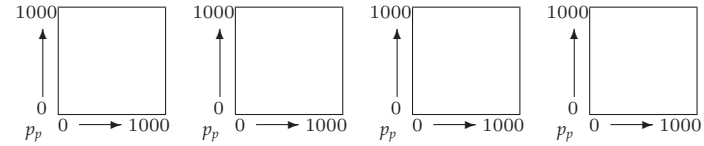


Figura B.44: Penalizaciones: influencia de p_p en la longitud de las trazas de error (Tabla A.24).

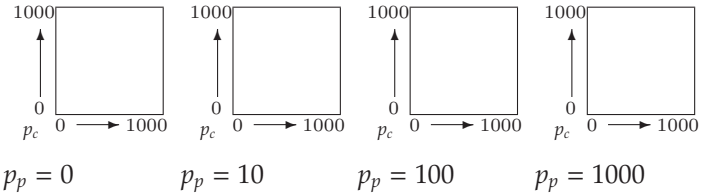


Figura B.45: Penalizaciones: influencia de p_c en la memoria requerida (Tabla A.25).

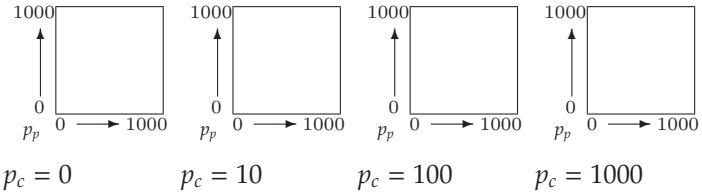


Figura B.46: Penalizaciones: influencia de p_p en la memoria requerida (Tabla A.25).

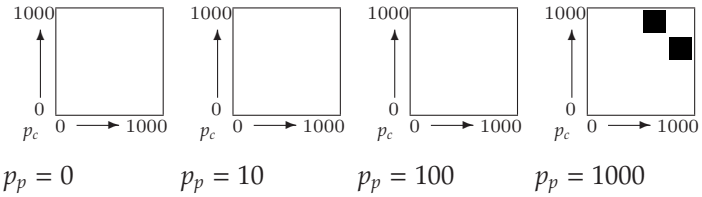


Figura B.47: Penalizaciones: influencia de p_c en el tiempo de CPU requerido (Tabla A.26).

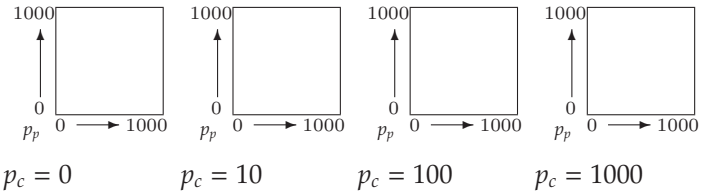


Figura B.48: Penalizaciones: influencia de p_p en el tiempo de CPU requerido (Tabla A.26).

Apéndice C

Bibliotecas

En este apéndice presentamos brevemente las dos principales bibliotecas en las que se han implementado los algoritmos metaheurísticos que utilizamos a lo largo de la tesis. La primera de ellas, JEAL (*Java Evolutionary Algorithms Library*) está implementada en Java y fue desarrollada por el doctorando para trabajar con algoritmos evolutivos tanto secuenciales como paralelos. La segunda, MALLBA, está implementada en C++ y es el resultado de un proyecto de investigación nacional desarrollado entre los años 1999 a 2002.

La biblioteca JEAL se ha usado para los problemas de planificación de proyectos software y de generación de casos de prueba. En ella se encuentran implementados los algoritmos genéticos y las estrategias evolutivas. La implementación del PSO es del Dr. Stefan Janson y fue incorporada a JEAL durante una estancia de investigación del doctorando en la Universidad de Leipzig. La biblioteca MALLBA se usó exclusivamente para resolver el problema de búsqueda de violaciones de propiedades de seguridad en sistemas concurrentes. Se hizo la elección de MALLBA en este caso para poder incorporar el código al de HSF-SPIN, un *model checker* experimental programado en C++, basado en SPIN y desarrollado por Stefan Edelkamp, Stefan Leue y Alberto Lluch Lafuente. La implementación de ACOhg tomó como base una implementación preliminar de ACO debida a Guillermo Ordóñez.

C.1. La biblioteca JEAL

El diseño de esta biblioteca se ha realizado teniendo siempre en mente el desarrollo de un software flexible y fácilmente ampliable que sea capaz de englobar cualquier algoritmo evolutivo [12]. En su diseño orientado a objetos se reflejan los componentes necesarios para resolver un problema de optimización con un algoritmo evolutivo. Estos componentes, cada uno de los cuales se encuentra representado en JEAL mediante una clase, son:

- El *problema*. Necesitamos evaluar las soluciones que encuentra el algoritmo para conocer su calidad. Esto se hace a través de la función de evaluación o función de aptitud (*fitness*) que depende del problema que se desea resolver.
- Los *individuos*. Las posibles soluciones del problema son representadas mediante individuos. Estos individuos son los que manipula el algoritmo evolutivo para realizar la búsqueda en el espacio de soluciones.

- La *población*. Está formada por un conjunto de individuos e información estadística.
- Los *operadores*. Son los encargados de manipular individuos para realizar la exploración del espacio de búsqueda.
- La *condición de parada*. Un algoritmo evolutivo sigue un proceso iterativo que debe parar en algún momento. El algoritmo se detendrá cuando cierta condición se cumpla.
- El *algoritmo*. El problema, los individuos, los operadores y la condición de parada deben colaborar de alguna forma para resolver el problema. La forma de colaborar la decide el algoritmo.

Como vimos en el Capítulo 4, los algoritmos evolutivos están formados por un bucle gobernado por la condición de parada y una serie de operadores que se aplican a los individuos de la población. En cada generación se aplican los mismos operadores. Lo que diferencia un algoritmo, como la estrategia evolutiva, de otro, como el algoritmo genético, desde el punto de vista de JEAL, son los operadores y los individuos utilizados. Así pues, para hacer flexible nuestro software nos hemos asegurado de que es fácil cambiar ambos componentes mediante archivos de configuración y hemos implementado una única clase para representar a los algoritmos evolutivos. De esta forma, mediante la elección adecuada de individuos, operadores y condición de parada, podemos obtener cualquier algoritmo evolutivo. A continuación veremos con más detalle cada uno de los componentes arriba identificados.

C.1.1. El problema

Los problemas a resolver son representados por subclases de la clase `Problem`. Esta clase posee un método que debe implementar el usuario para evaluar las soluciones (función de *fitness*). En JEAL se asume, sin pérdidad de generalidad, que este valor es positivo y que una solución es mejor cuando tiene mayor aptitud (maximización).

C.1.2. Individuos y población

Los distintos tipos de individuos son representados por subclases de la clase `Individual` (Figura C.1). Los individuos siguen el patrón de diseño fábrica (*factory*). Existe un nombre para cada tipo de individuo implementado. La relación entre estos nombres y las clases que los implementan se encuentra almacenada en un archivo que puede ser editado para incorporar nuevos tipos de individuos sin necesidad de compilar la biblioteca de nuevo.

Para evitar la invocación frecuente del recolector de basura, se reutilizan los objetos que representan a los individuos. Esto se hace con ayuda de una clase auxiliar llamada `VirtualHeap`. Esta clase guarda los individuos que no se usan en un momento dado. Cuando es necesario emplear nuevos individuos, son solicitados a este objeto. Si tiene alguno disponible lo devuelve, en otro caso debe crearlo. El objeto que representa la población (objeto de la clase `Population`) es el encargado de controlar este objeto `VirtualHeap` y ofrece entre sus métodos algunos para crear y desechar individuos[1].

[1]Usamos el verbo desechar para referimos a que el objeto pasa a disposición del `VirtualHeap` para ser reutilizado posteriormente si se solicita un nuevo individuo.

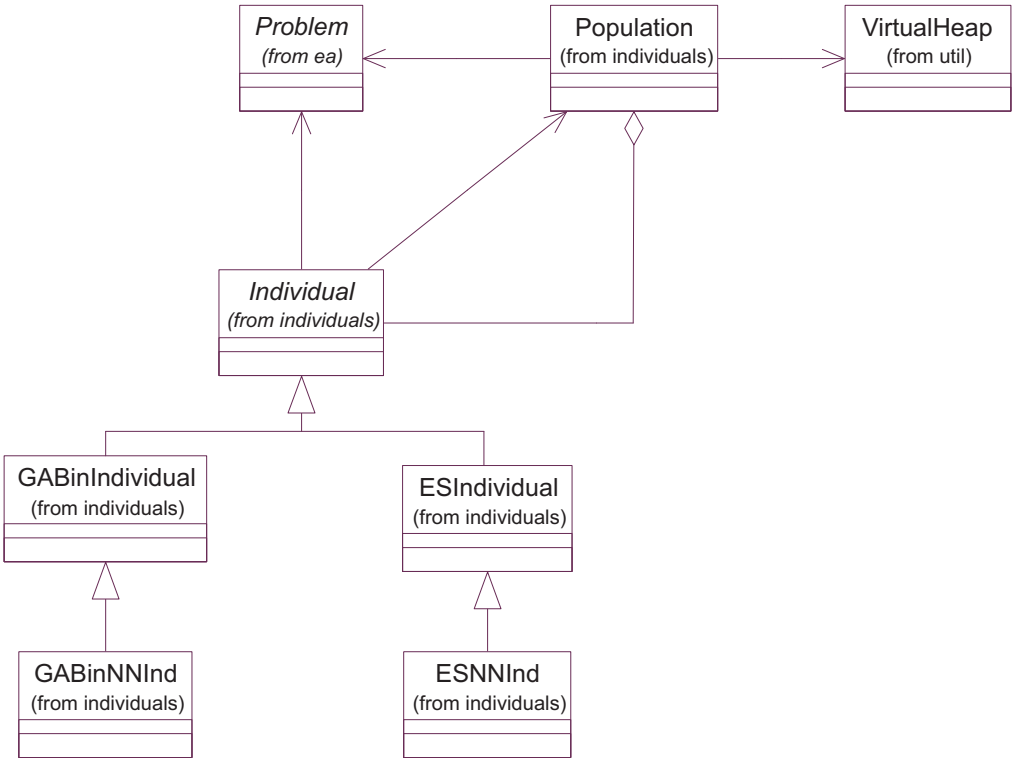


Figura C.1: Clases para representar los individuos y la población.

C.1.3. Operadores

La cantidad de operadores en el campo de la computación evolutiva es enorme. Para poder emplear el patrón de fábrica con los operadores, hemos diseñado una clase, denominada `EAOperator`, que declara la funcionalidad básica de todos los operadores o, al menos, de la mayoría. Hay operadores que actúan sobre un individuo (como la mutación), otros actúan sobre dos (como los operadores de cruce), otros sobre la población (como la selección), etc. El único nexo entre todos ellos es que toman individuos, los transforman (posiblemente) y los devuelven transformados. Esos individuos los pueden tomar de la población o de la salida de otro operador. Por esto, las subclases de `EAOperator` deben redefinir un método que acepta un array de individuos y devuelve otro array de individuos: `operation(Individual [] ind)`. La relación entre los nombres y las clases de los operadores se encuentra en un fichero de configuración que, al igual que ocurría con los tipos de individuo, puede ser editado para que los usuarios añadan nuevos operadores. Hemos implementado una gran cantidad de operadores. Algunos son específicos de un problema e incluso de un tipo de individuo, otros en cambio, actúan sobre cualquier operador (como es el caso de los operadores de selección). En la Figura C.2 presentamos un diagrama de clases con todos

los operadores implementados.

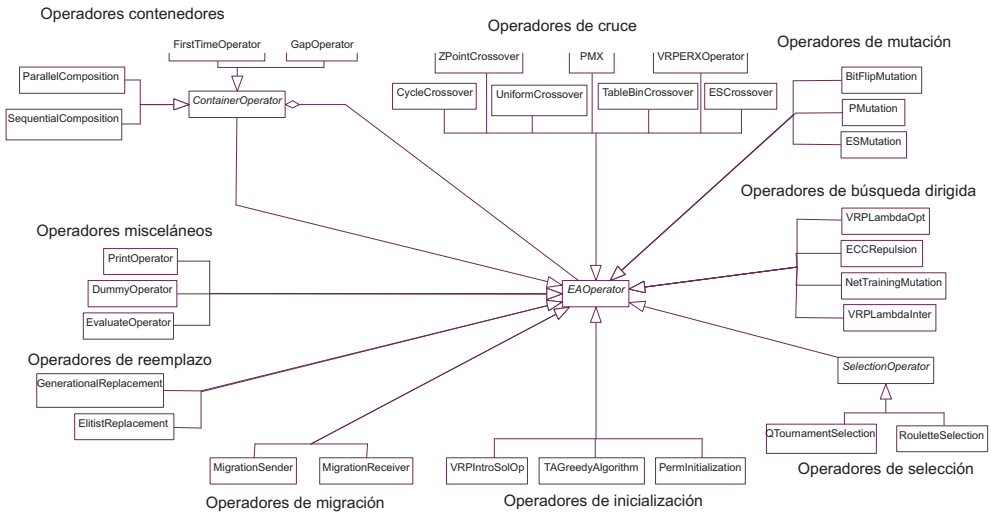


Figura C.2: Clases para representar los operadores.

C.1.4. Algoritmo

Como mencionamos anteriormente, existe una única clase para representar a los algoritmos evolutivos, se trata de `EvolutionaryAlgorithm` (Figura C.3). Esta clase posee un método que implementa el algoritmo evolutivo. En cada paso, el algoritmo parte de un array nulo de individuos y aplica los operadores secuencialmente, esto es, al vector nulo le aplica el primero, la salida de ese operador es la entrada del siguiente y así procede hasta acabar con todos los operadores. La salida del último operador es entregada al heap. El procedimiento anterior se repite hasta que se cumple la condición de parada.

Existe una clase denominada `MessageManager` que se utiliza en los algoritmos evolutivos paralelos. La ejecución en paralelo está supervisada por un proceso central que se encarga de sincronizar al resto y recoger los resultados de cada subalgoritmo. Este proceso está implementado en la clase `DistributionManager` y cuando se inicia, se pone a la escucha de conexiones. Los subalgoritmos del algoritmo paralelo emplean la clase `MessageManager` como interfaz con este proceso central. Existe un solo objeto `MessageManager` en cada subalgoritmo, el cual se conecta al proceso central cuando comienza la ejecución del subalgoritmo.

C.1.5. Condición de parada

Las clases que representan condiciones de parada deben ser descendientes de `StopCondition`. Las condiciones de parada siguen el patrón fábrica, existiendo, como en los casos anteriores, un fichero que establece la relación entre los nombres y las clases que implementan las condiciones de parada (Figura C.4).

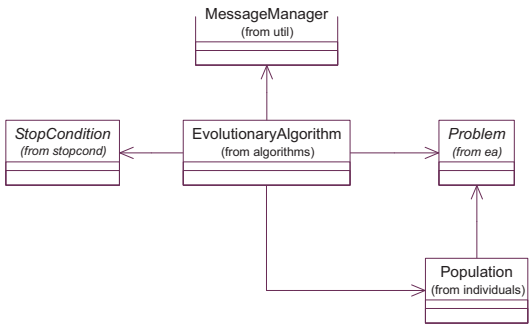


Figura C.3: Clases para representar el algoritmo evolutivo.

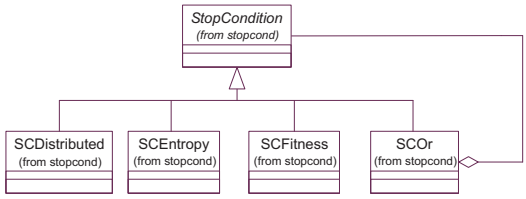


Figura C.4: Clases para representar la condición de parada.

C.1.6. Puesta en marcha

Una vez que tenemos todos lo componentes de un algoritmo evolutivo, debemos ponerlo en marcha para que realice su trabajo. Esto lo hace la clase `Driver` que, en su método `main()`, lee el fichero de configuración que se le pasa como parámetro y crea los objetos correspondientes al problema, la población, los operadores, el gestor de mensajes, la condición de parada y el algoritmo. Una vez que todos los objetos están creados, se ejecuta el algoritmo. Cuando concluye, la clase `Driver` escribe en uno o varios ficheros de texto el resultado de la ejecución.

C.2. La biblioteca MALLBA

El proyecto MALLBA[2] tuvo como objetivo el desarrollo de una biblioteca de esqueletos para la optimización combinatoria que incluye técnicas exactas, heurísticas e híbridas. Soporta de forma amigable y eficiente tanto plataformas secuenciales como paralelas. Respecto a los entornos paralelos, son considerados tanto las redes de área local (LAN) como las de área extensa (WAN).

Las tres principales características que se pueden destacar de la biblioteca MALLBA son la integración de todos los esqueletos bajo unos mismos principios de diseño, la facilidad de cambiar entre entornos se-

[2] `http://neo.lcc.uma.es/mallba/easy-mallba/index.html`

cuenciales y paralelos y, por último, la cooperación entre los esqueletos para proporcionar nuevos híbridos. En los siguientes párrafos, describiremos en mayor detalle cada uno de los aspectos de esta biblioteca.

C.2.1. Arquitectura de MALLBA

Desde un punto de vista del hardware, inicialmente la infraestructura de MALLBA estaba compuesta por computadores y redes de comunicación de las Universidades de Málaga (UMA), La Laguna (ULL) y Barcelona (UPC), todas ellas de España. Estas tres universidades están conectadas a través de RedIRIS, la red española de la comunidad académica y científica que conecta a las principales universidades y centros de investigación en España. Esta red está controlada por el CSIC (*Consejo Superior de Investigaciones Científicas*). RedIRIS es una WAN que usa tecnología ATM de 34/155 Mbps. Aunque para los primeros experimentos se utilizó esa tecnología, la biblioteca es independiente de la plataforma donde se ejecuta; de hecho, la versión actual de la biblioteca MALLBA funciona sobre cualquier *cluster* de PCs con Linux.

Respecto al software, todos los esqueletos que se han desarrollado en MALLBA han sido implementados siguiendo el esquema de *esqueletos software* (similar al patrón de estrategia [102]) con un interfaz interno común y otro público. Un esqueleto es una unidad algorítmica que, en forma de plantilla, implementa un algoritmo genérico. Un algoritmo se instancia para resolver un problema concreto, rellenando los requisitos especificados en su interfaz. Esta característica permite el rápido prototipado de aplicaciones y el acceso transparente a plataformas paralelas. En la biblioteca MALLBA, cada esqueleto implementa una técnica de los campos de la optimización exacta, heurística e híbrida. Toda la biblioteca está desarrollada en C++. Se eligió este lenguaje debido a que proporciona las características de un lenguaje de alto nivel orientado a objetos y al mismo tiempo genera código ejecutable muy eficiente.

Un aspecto clave es la separación conceptual de la parte propia del algoritmo y la parte correspondiente al problema. El diseño basado en esqueletos soluciona este aspecto, ya que define un conjunto de clases para cada parte: unas contienen el comportamiento del algoritmo y son independientes del problema mientras que las otras encapsulan las características del problema. Ambos conjuntos de clases cooperan entre sí a través de una pequeña interfaz fija, lo suficientemente genérica para no limitar las características del problema. El usuario es el encargado de implementar las clases asociadas al problema.

C.2.2. Interfaz de usuario

Para cada algoritmo de optimización se proporciona un conjunto de clases que, en función de su dependencia al problema objetivo, pueden agruparse en dos categorías: clases *provistas* y clases *requeridas* (véase la Figura C.5(a)). A continuación comentamos sus características.

Las *clases provistas* son las responsables de implementar toda la funcionalidad básica del algoritmo correspondiente. En primer lugar, la clase `Solver` encapsula el motor de optimización del algoritmo. Este motor de optimización es plenamente genérico, interactuando con el problema a través de las clases que el usuario debe proporcionar, y que serán descritas más adelante. En segundo lugar, la clase `SetupParams` contiene los parámetros propios de la ejecución del algoritmo como, por ejemplo, el número de iteraciones, el tamaño de la población en un algoritmo genético, el mecanismo de gestión de la cola de subproblemas en un algoritmo de ramificación y poda, etc. Otra clase provista es `Statistics`, cuya finalidad es la recolección de estadísticas propias del algoritmo empleado. Finalmente, las clases `StateVariable` y `StateCenter` facilitan la hibridación de algoritmos (gestión del estado).

Las *clases requeridas* son las responsables de proporcionar al esqueleto detalles sobre todos los aspectos dependientes del problema de optimización. Estas clases poseen una interfaz única y prefijada,

permitiendo, de esta manera, que las clases provistas puedan usarlas sin necesidad de relacionarse directamente con los detalles del problema. Entre las clases requeridas encontramos: la clase `Problem` (que debe proporcionar los métodos necesarios para manipular los datos propios del problema), la clase `Solution` (que encapsulará el manejo de soluciones parciales del problema tratado), la clase `UserStatistics` (que permitirá al usuario recoger aquella información estadística de su interés que no estuviera siendo monitorizada por la clase provista `Statistics`), así como otras clases que dependen del esqueleto algorítmico elegido. Así pues, el usuario de MALLBA sólo necesita implementar las estructuras de datos dependientes del problema, así como dotar de un comportamiento específico a los métodos incluidos en las interfaces de las clases requeridas.

En relación a los esqueletos de código podemos distinguir tres perfiles de usuarios (Figura C.5(b)), aunque nada impide que los tres perfiles se den en una misma persona o en varias. Los perfiles son:

- **Programador del esqueleto:** es el experto en el dominio del algoritmo desarrollado y se encarga de diseñar e implementar el esqueleto a partir del conocimiento del algoritmo. El programador debe decidir qué clases son proporcionadas por el esqueleto y cuáles deben ser facilitadas por el experto del problema. También debe decidir el mecanismo de comunicación existente para permitir la ejecución paralela.

- **Experto en el dominio del problema:** es el encargado de instanciar el esqueleto con los datos adecuados del problema del cual es experto. Aunque no tiene por qué conocer la implementación del algoritmo, debe saber cómo evoluciona desde un punto de vista numérico, ya que tiene que conocer la finalidad de las clases y métodos que debe rellenar para el problema de optimización.

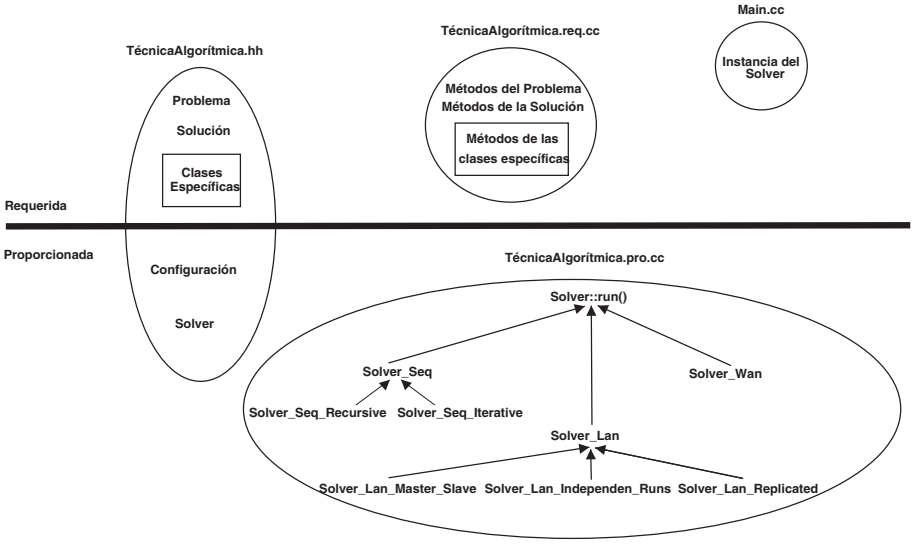
- **Usuario final:** es el que utiliza el esqueleto de código una vez que ha sido instanciado por el usuario correspondiente al perfil anterior. El usuario final será el encargado de dar la configuración a los parámetros del esqueleto (fichero `<esqueleto>.cfg`) de acuerdo con la ejecución que desee llevar a cabo. Este usuario tendrá a su disposición toda la información sobre la ejecución del algoritmo, permitiéndole hacer un seguimiento, obtener estadísticas, conocer el tiempo consumido hasta la mejor solución, etc.

C.2.3. Interfaz de comunicación

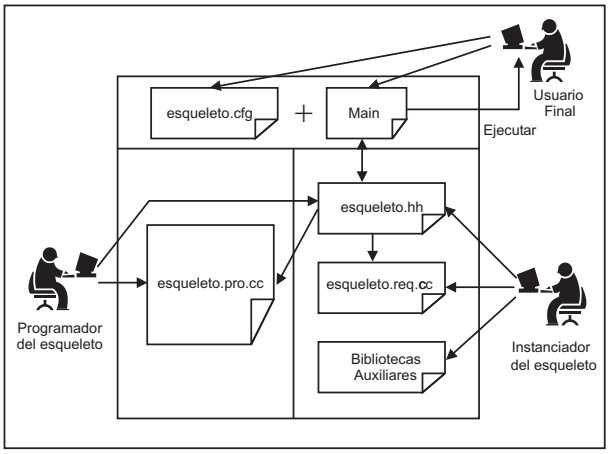
Las redes de área local son actualmente muy baratas y gozan de gran popularidad en los laboratorios y departamentos. Más aún, Internet nos permite comunicar esas redes para poder explotar los recursos de forma conjunta de numerosos sitios geográficamente distribuidos. Para este fin, es necesario disponer de un mecanismo que permita comunicar los esqueletos entre sí, tanto en redes de área local (LAN) como de área extensa (WAN). Como los esqueletos los hemos desarrollado a un alto nivel, sería deseable desarrollar un mecanismo de comunicación también de alto nivel pero sin pérdida de rendimiento.

Para permitir el diseño de algoritmos que se ejecuten de forma distribuida en redes de computadores, se ha diseñado en MALLBA un mecanismo de comunicación de alto nivel implementado sobre MPI. MPI reúne varias características muy interesantes: en primer lugar, es un estándar y posee varias implementaciones muy eficientes; en segundo lugar, su uso se está volviendo muy frecuente en la comunidad de software paralelo y, por último, está integrado con los sistemas más modernos y prometedores para computación grid como, por ejemplo, Condor y Globus.

Aunque no existe ningún inconveniente en usar MPI directamente, se prefirió desarrollar una capa intermedia muy ligera a la que se denominó `NetStream` (véase la Figura C.6), que permite un uso más



(a) Arquitectura de un esqueleto MALLBA. La línea horizontal establece la separación entre las clases C++ que el usuario debe rellenar (parte superior) y las clases del esqueleto que ya están incluidas y operativas (parte de abajo).



(b) Interacción entre los diferentes tipos de usuarios y ficheros de MALLBA que utilizan.

Figura C.5: Esqueletos MALLBA: Estructura e interacción.

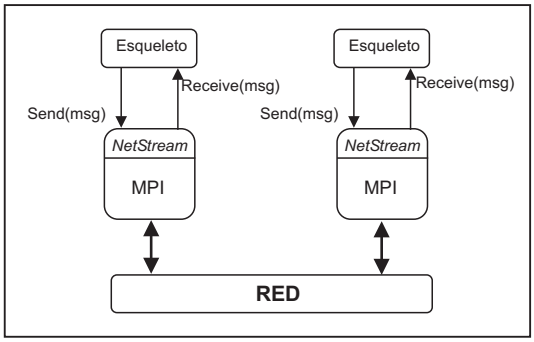


Figura C.6: Sistema de comunicación NetStream sobre MPI.

sencillo de MPI sin pérdida de eficiencia. Con esta capa, un programador de esqueletos de MALLBA puede evitar la gran cantidad de parámetros necesarios en MPI e interactuar con la red usando simplemente *modificadores de flujo*, lo que permite realizar operaciones complejas de paso de mensaje mediante el uso de los operadores habituales de flujo `<<` y `>>` de C++.

C.2.4. Interfaz de hibridación

En esta sección discutimos los mecanismos disponibles en MALLBA para combinar esqueletos en busca de técnicas híbridas más eficientes. En un sentido amplio, la hibridación [69] consiste en la inclusión de conocimiento del problema en el método de búsqueda general. La hibridación se suele clasificar en dos clases según la forma en que el conocimiento del problema se incluye en el algoritmo [64]:

- **Hibridación fuerte**: el conocimiento se incluye usando una representación u operadores específicos.
- **Hibridación débil**: el algoritmo resultante es una combinación de varios algoritmos claramente distinguibles.

MALLBA ofrece soporte para la hibridación débil, es decir, permite la combinación de varios algoritmos. Para esto, se define el *estado del esqueleto*. En este estado se almacena de forma genérica toda la información necesaria para influir en su comportamiento. Estos valores son manejados de forma abstracta por las clases `StateVariable` y `StateCenter`. Mediante estas clases, un algoritmo puede obtener información de otro o influir en su comportamiento de forma genérica.

Apéndice D

Relación de publicaciones que sustentan la tesis doctoral

En este apéndice se presenta el conjunto de trabajos que han sido publicados como fruto de las investigaciones desarrolladas a lo largo de esta tesis doctoral. Estas publicaciones avalan el interés, la validez, y las aportaciones de esta tesis doctoral en la literatura, ya que estos trabajos han aparecido publicados en foros de prestigio y, por tanto, han sido sometidos a procesos de revisión por reconocidos investigadores especializados. En la Figura D.1 se muestra un esquema de estas publicaciones. A continuación se muestran las referencias de todas las publicaciones.

[1] Enrique Alba y Francisco Chicano. Software project management with GAs. *Information Sciences*, 177(11):2380–2401, Junio 2007.

[2] Enrique Alba y Francisco Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers & Operations Research* (en imprenta).

[3] Enrique Alba y Francisco Chicano. On the behavior of parallel genetic algorithms for optimal placement of antennae in telecommunications. *International Journal of Foundations of Computer Science*, 16(2):343–359, Abril 2005.

[4] Enrique Alba y J. Francisco Chicano. Training neural networks with GA hybrid algorithms. En *Genetic and Evolutionary Computation Conference (GECCO04), LNCS 3102*, páginas 852–863, Seattle, EEUU, Junio 2004. Springer-Verlag.

[5] Enrique Alba y J. Francisco Chicano. Software testing with evolutionary strategies. En *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques, LNCS 3943*, páginas 50–65, Heraklion, Grecia, Septiembre 2005.

[6] Enrique Alba, Guillermo Molina y Francisco Chicano. Optimal placement of antennae using metaheuristics. En *Sixth International Conference on Numerical Methods and Applications (NM&A 06), LNCS 4310*, páginas 214–222, Borovets, Bulgaria, Agosto 2006.

[7] Enrique Alba y Francisco Chicano. Ant colony optimization for model checking. En *EUROCAST 2007, LNCS 4739*, páginas 523–530, Gran Canaria, España, Febrero 2007.

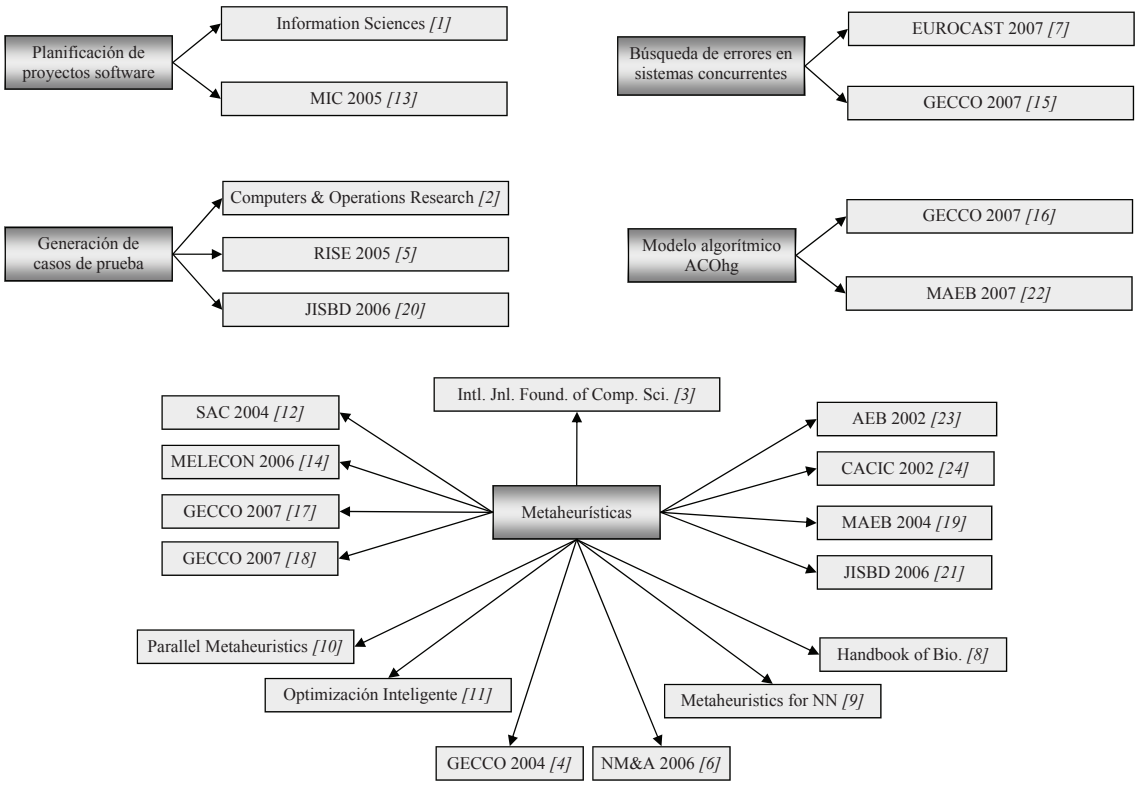


Figura D.1: Esquema de las publicaciones que avalan el trabajo realizado en esta tesis doctoral.

[8] Enrique Alba, J. Francisco Chicano, Francisco Luna, Gabriel Luque y Antonio J. Nebro. *Handbook of Bioinspired Algorithms and Applications*, volumen 6 de *Chapman & Hall/CRC Computer & Information Science Series*, capítulo 26, Advanced evolutionary algorithms for training neural networks. CRC Press, 2005.

[9] Enrique Alba y Francisco Chicano. *Metaheuristic Procedures for Training Neural Networks*, volumen 36 de *Operations Research/Computer Science Interfaces Series*, capítulo 6, Genetic Algorithms. Springer, 2006.

[10] Sergio Nesmachnow, Héctor Cancela, Enrique Alba y Francisco Chicano. *Parallel Metaheuristics: A New Class of Algorithms*, capítulo 20, Parallel Metaheuristics in Telecommunications, páginas 495–515. John Wiley & Sons, 2005.

[11] Enrique Alba, J. Francisco Chicano, Carlos Cotta, Bernabé Dorronsoro, Francisco Luna, Gabriel Luque y Antonio J. Nebro. *Optimización Inteligente. Técnicas de Inteligencia Computacional para Optimización*, capítulo 5, Metaheurísticas secuenciales y paralelas para optimización de problemas complejos Universidad de Málaga. Servicio de Publicaciones e Intercambio Científico, Julio 2004.

[12] Enrique Alba y J. Francisco Chicano. Solving the error correctring code problem with parallel

hybrid heuristics. En *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC04)*, páginas 985–989, Nicosia, Chipre, Marzo 2004.

[13] Enrique Alba y J. Francisco Chicano. Management of software projects with GAs. En *Metaheuristics International Conference (MIC-2005)*, páginas 13–18, Viena, Austria, Agosto 2005.

[14] Enrique Alba y Francisco Chicano. Evolutionary algorithms in telecommunication. En *Proceedings of the 13th IEEE Mediterranean Electrotechnical Conference (MELECON 2006)*, Benalmádena, Málaga, España, Mayo 2006.

[15] Enrique Alba y Francisco Chicano. Finding safety errors with ACO. En *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, páginas 1066–1073, London, Reino Unido, Julio 2007. ACM Press. **Nominado a mejor artículo**.

[16] Enrique Alba y Francisco Chicano. ACOhg: dealing with huge graphs. En *Proceedings of the Genetic and Evolutionary Conference*, páginas 10–17, London, Reino Unido, Julio 2007. ACM Press.

[17] Antonio J. Nebro, Enrique Alba, Guillermo Molina, Francisco Chicano, Francisco Luna y Juan J. Durillo. Optimal antenna placement using a new multi-objective CHC algorithm. En *Genetic and Evolutionary Computation Conference (GECCO07)*, páginas 876–883, London, Reino Unido, Julio 2007. ACM Press.

[18] José García-Nieto, Enrique Alba y Francisco Chicano. Using metaheuristic algorithms remotely via ROS. En *Genetic and Evolutionary Computation Conference (GECCO07)*, página 1510, London, Reino Unido, Julio 2007. ACM Press.

[19] Enrique Alba, J. Francisco Chicano, Bernabé Dorronsoro y Gabriel Luque. Diseño de códigos correctores de errores con algoritmos genéticos. En *Actas del Tercer Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados (MAEB04)*, páginas 51–58, Córdoba, España, Febrero 2004.

[20] Enrique Alba, Francisco Chicano y Stefan Janson. Testeo de software con dos técnicas metaheurísticas. En *XI Jornadas de Ingeniería del Software y Bases de Datos*, páginas 109–118, Sitges, Barcelona, España, Octubre 2006.

[21] Enrique Alba, Jose G. Nieto y Francisco Chicano. ROS: Servicio de optimización remota. En *XI Jornadas de Ingeniería del Software y Bases de Datos*, páginas 508–513, Sitges, Barcelona, España, Octubre 2006.

[22] Enrique Alba y Francisco Chicano. Una versión de ACO para problemas con grafos de muy gran extensión. En *Actas del Quinto Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, MAEB 2007*, páginas 741–748, Puerto de la Cruz, Tenerife, España, Febrero 2007.

[23] Enrique Alba, Antonio J. Nebro y Francisco Chicano. Heterogeneidad, WAN y nuevas aplicaciones de los algoritmos evolutivos paralelos. En *Actas del Primer Congreso Español de Algoritmos Evolutivos y Bioinspirados (AEB02)*, páginas 402–409, Mérida, España, Febrero 2002.

[24] Enrique Alba, Carlos Cotta, Francisco Chicano y Antonio J. Nebro. Parallel evolutionary algorithms in telecommunications: Two case studies. En *Actas del Congreso Argentino de Ciencias de la Computación (CACIC02)*, Buenos Aires, Argentina, 2002.

Apéndice E

Summary of this thesis in English

The design of efficient algorithms to solve complex problems has been one of the more important aspects in computer science research. The aim is the development of new methods that are able to solve complex problems with a low computational effort, outperforming the current algorithms. In this context, the research activity in exact, *ad hoc* heuristics, and metaheuristic algorithms is increasing nowadays.

The main advantage of using exact algorithms is that they always find the global optimum for any problem. However, they have an important drawback: in real problems (NP-hard in most cases) their execution time grows in an exponential way with the size of the problem. On the other hand, *ad hoc* heuristic algorithms usually are quite fast, but the quality of the solutions found is far from the optimum. Another drawback of *ad hoc* heuristics is that they are difficult to design for some problems. Metaheuristic algorithms offer an appropriate balance between both extremes: they are generic methods that offer good quality solutions (global optimum in many cases) in a reasonable time.

Software Engineering, despite its short age, is an important source of optimization problems. Engineers and software project managers have to tackle many optimization problems that can not be solved with exact techniques. Thus, it is possible to apply metaheuristics to these problems in order to provide the experts with a good balance between quality and efficiency.

In this PhD dissertation we apply metaheuristic techniques to optimization problems of the Software Engineering domain, analyzing different alternatives for exploiting these techniques. In concrete, the problems are software project scheduling, automatic test case generation, and search for safety property violations in concurrent systems. In addition, these metaheuristics have been designed and implemented using methods and tools coming from the Software Engineering field in order to develop good quality software. In this way, we close the loop: Software Engineering for developing tools that solve Software Engineering problems.

E.1. Organization of this thesis

This thesis is organized in four parts. In the first one, the preliminaries of Software Engineering and metaheuristics are presented. In the second part, we present the methodology and the results obtained after the application of metaheuristics to Software Engineering problems. In the third part, we show the main conclusions and future work of the thesis. Finally, in the forth part, the appendices and several secondary results can be found.

Next, we detail the content of the chapters.

- **Chapter 1: Introduction.** This chapter presents an outline of the dissertation including objectives, phases, contributions, and organization.
- **Chapter 2: Software Engineering optimization problems.** This is a review chapter. We identify and classify works in which Software Engineering optimization problems are solved. From the optimization problems we select three to be solved with metaheuristic techniques in this PhD dissertation: software project scheduling, test case generation, and search for safety property violations in concurrent systems.
- **Chapter 3: Tackled problems.** In this chapter, we present and formalize in detail the three Software Engineering optimization problems selected.
- **Chapter 4: Metaheuristics.** This chapter introduces the metaheuristic techniques, classifies them, and offers a formal definition. The metaheuristic algorithms used in the thesis are described in detail: evolutionary algorithms, particle swarm optimization, and ant colony optimization.
- **Chapter 5: Methodological issues in this thesis**. This chapter describes the proposed techniques for solving the selected problems. The most important contribution in this chapter is a new model of ACO, called ACOhg (ACO for huge graphs), that is able to solve problems with a very large underlying graph.
- **Chapter 6: Application of genetic algorithms to software project scheduling.** This chapter analyzes the results obtained after the application of genetic algorithms to the problem of software project scheduling.
- **Chapter 7: Application of metaheuristics to the test case generation.** This chapter analyzes the results obtained after the application of parallel decentralized and sequential centralized versions of genetic algorithms and evolutionary strategies to test case generation. Particle swarm optimization is also applied to this problem.
- **Chapter 8: Application of ACOhg to the problem of searching for safety property violations in concurrent systems.** This chapter analyzes the results obtained after the application of ACOhg (the new ACO model proposed) to the search for safety property violations in concurrent systems.
- **Appendix A: Analyzing the configuration of the algorithms.** In this appendix, we study the results of some secondary experiments performed in order to analyze the influence of several algorithmic parameters on the results.
- **Appendix B: Statistical validation of the results.** This appendix presents the results of all the statistical tests performed in the thesis for supporting the conclusions obtained from the results of the experiments.
- **Appendix C: Metaheuristic libraries.** This appendix gives a brief presentation of the metaheuristic libraries used for the experiments: JEAL and MALLBA. JEAL was developed for this thesis and MALLBA was the result of a Spanish research project.
- **Appendix D: Publications.** This appendix presents the publications of the PhD student that are related to the PhD dissertation and support its quality.

E.2. Software Engineering optimization problems

In the second chapter of this thesis we present a brief description of a set of optimization problems of the Software Engineering domain. It is not an exhaustive review but a significant sample in which the diversity of the optimization problems that appear in Software Engineering can be observed. In order to present the problems in a sorted way, we propose a classification of them according to the phase in which the problems appear in the software development process. The phases considered are: requirement analysis, design, implementation, testing, deployment, maintenance, and management. In Figure E.1 we show the number of works for each category.

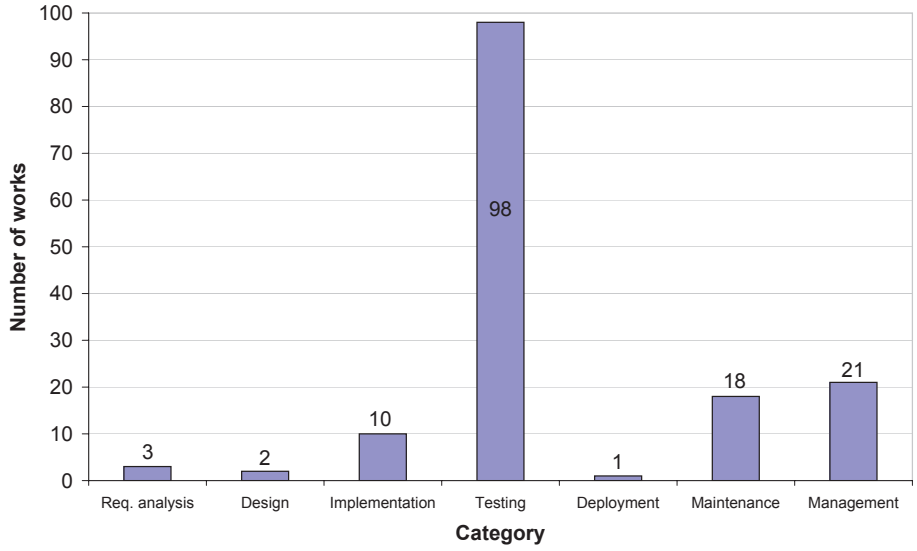


Figure E.1: Number of works for each category.

The number of researchers and software engineers that apply optimization techniques to their problems is increasing, and the same happens with the number of related research articles in the last years. There is a phase of the software development cycle that has received more attention: the testing phase. Two thirds of the papers revised for our brief review belong to this category. The second category with more articles is the management phase. This sheds some light on the main concerns in Software Engineering.

In this thesis we tackle three Software Engineering optimization problems: software project management, test case generation, and search for safety property violations in concurrent systems. These problems belong to the two categories that are more interesting for the software community: testing and management.

E.3. Tackled problems

In the third chapter we present in detail the three Software Engineering problems that we have selected to solve with metaheuristic techniques. These problems are the project scheduling problem, the test case

generation problem, and the search for safety property violations in concurrent systems. In the following sections we present a brief description of the three problems.

E.3.1. Project scheduling problem

The high complexity of existing software projects justifies the research into computer aided tools to properly plan the project development. Current software projects usually demand complex management involving scheduling, planning, and monitoring tasks. There is a need to control people and processes, and to efficiently allocate resources in order to achieve specific objectives while satisfying a variety of constraints. In a general way, the project scheduling problem consists in defining what resources are used to perform each task and when it should be carried out. The tasks can be very diverse: from maintaining documents to writing programs. The resources include people, machines, time, etc. The objectives are usually to minimize the project duration, to minimize the project cost, and to maximize the product quality [54]. In an actual project, the manager wants an automatic plan reconciling as far as possible these three conflicting goals.

In Project Scheduling Problem (PSP), the resources considered are people with a set of skills, and a salary. These employees have a maximum degree of dedication to the project. Formally, each person (employee) is denoted with e_i, where i goes from 1 to E (the number of employees). Let SK be the set of skills, and s_i the i-th skill with i varying from 1 to $S = |SK|$. The skills of the employee e_i will be denoted with $e_i^{skills} \subseteq SK$, the monthly salary with e_i^{salary}, and the maximum dedication to the project with e_i^{maxded}. The salary and the maximum dedication are both real numbers. The former is expressed in fictitious currency units, while the latter is the ratio between the amount of hours dedicated to the project and the full working day length of the employee. The tasks of the project are denoted with t_k, where k goes from 1 to T (the number of tasks). Each task t_k has a set of required skills associated with it that we denote with t_k^{skills} and an effort t_k^{effort} expressed in person-month (PM). The tasks must be performed according to a Task Precedence Graph (TPG). It indicates which tasks must be completed before a new task is started. The TPG is an acyclic directed graph $G(V, A)$ with a vertex set $V = \{t_1, t_2, \ldots, t_T\}$ and an arc set $A \in V \times V$, where $(t_i, t_j) \in A$ if the task t_i must be completed, with no other intervening tasks, before task t_j can start.

The goals of the problem are to minimize the cost and the duration of the project. In addition, any solution must fulfil three constraints:

- **R1**: each task must be performed by at least one person.
- **R2**: the set of required skills of a task must be included in the union of the skills of the employees performing the task.
- **R3**: no employee must exceed her/his maximum dedication to the project.

E.3.2. Test case generation

From the very beginning of computer research, computer engineers have been interested in techniques allowing them to know whether a software module fulfils a set of requirements (the specification). Modern software is very complex and these techniques have become a necessity in most software companies. One of these techniques is *formal verification*, in which some properties of the software can be checked much like a mathematical theorem defined on the source code. Another well-known and fully automatic formal

method is *model checking*. Both in formal verification and model checking a model of the program is required in order to prove (or refute) the properties we want to check. Nevertheless, the most popular technique used to check software requirements is *software testing*. With this technique, the engineer selects a set of program inputs (test cases) and tests the program with them. If the program behaviour is as expected, s/he assumes that it is correct. Since the size of the test cases set is the engineer's decision, s/he can control the effort dedicated to the testing task. This is a very important, time consuming, and hard task in software development [21, 208, 217]. *Automatic test case generation* consists of automatically proposing a suitable set of test cases for a program to be tested (the *test program*). It comes as a way of releasing engineers from the task of finding the best set of test cases to check program correctness. The automatic generation of test cases for computer programs has been dealt with in the literature for many years [59, 207]. We can distinguish four large paradigms in search-based software testing that differ in the kind of information they use to generate the test cases: structural testing, functional testing, grey-box testing, and non-functional testing [196].

In *structural testing* [112, 190, 199, 206, 249] the test case generator uses the structural information of the program to guide the search for new test cases (for this reason it is also called white-box testing). Usually, this structural information is gathered from the *control flow graph* of the program. Structural testing seeks to execute every testable element under consideration (whether statements, branches, conditions, etc.). We can find several alternatives in the literature for structural testing. In the so-called *random test case generation*, the test cases are created randomly until the objective is satisfied or a maximum number of test cases are generated [36, 206]. *Symbolic test case generation* [59] involves using symbolic rather than concrete values in order to get a symbolic execution. Some algebraic constraints are obtained from this symbolic execution and these constraints are used for finding test cases [221]. A third (widespread) approach is *dynamic test case generation*. In this case, the program is instrumented to pass information to the test generator. The test generator checks whether the test adequacy criterion is fulfilled or not. If the criterion is not fulfilled it creates new test cases to serve as input for the program. The test case generation process is translated into a minimization problem, where the objective function is some kind of "distance" to a desirable execution where the adequacy criterion is fulfilled. This paradigm was presented in [207] and much work has been based on it [151, 164, 206, 287]. Some hybrid techniques combining symbolic and concrete execution have been explored with very good results. This is the case of the tools DART [112] and CUTE [249].

E.3.3. Search for safety property violations in concurrent systems

Model checking [58] is a well-known and fully automatic formal method in which all the possible states of a given model are analyzed (in an explicit or implicit way) in order to prove (or refute) that the model satisfies a given property. This property is specified using a temporal logic like Linear Temporal Logic (LTL) or Computation Tree Logic (CTL). One of the best known explicit state model checkers is SPIN [145], which takes a software model codified in Promela and a property specified in LTL as inputs. SPIN transforms the model and the negation of the LTL formula into Büchi automata in order to perform their intersection. The resulting automaton is explored to search for a path starting in the initial state and including a cycle of states containing an *accepting state* (Figure E.2). If such a path is found, then there exists at least one execution of the model not fulfilling the LTL property (see [145] for more details). If such kind of path does not exist, then the model fulfils the property and the verification ends with success. In SPIN, this exploration is performed with Nested-DFS [144], an exhaustive algorithm. When the property to check is a safety property [189], the verification is reduced to a search for one path from

the initial state to one accepting state in the Büchi automaton. This path represents an execution of the concurrent model in which the given safety property is violated. Finding such a path is the case in which we are interested.

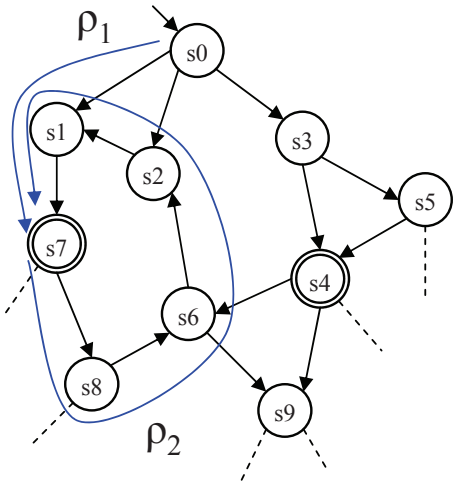


Figure E.2: Intersection Büchi automaton with an accepting sequence.

The foundations of this problem are based on Büchi automata theory, however, we can formalize the problem in the following reduced way. Let $G = (S, T)$ be a directed graph where S is the set of nodes and $T \subseteq S \times S$ is the set of arcs. Let $q \in S$ be the *initial node* of the graph and $F \subseteq S$ a set of distinguished nodes that we call *final nodes* (accepting states). We denote with $T(s)$ the successors of node s. A finite path over the graph is a sequence of nodes $\pi = s_1 s_2 \dots s_n$ where $s_i \in S$ for $i = 1, 2, \dots, n$. We denote with π_i the ith node of the sequence and we use $|\pi|$ to refer to the length of the path, that is, the number of nodes of π. We say that a path π is a *starting path* if the first node of the path is the initial node of the graph, that is, $\pi_1 = q$. We will use π_* to refer to the last node of the sequence π, that is, $\pi_* = \pi_{|\pi|}$. The problem at hands consists in finding a starting path π ending in a final node. That is, find π subject to $\pi_1 = q \wedge \pi_* \in F$

The graph G used in the problem is derived from the intersection Büchi automaton B of the model and the negation of the LTL formula of the property. The set of nodes S in G is the set of states in B, the set of arcs T in G is the set of transitions in B, the initial node q in G is the initial state in B, and the set of final nodes F in G is the set of accepting states in B.

E.4. Metaheuristics

A *metaheuristic* can be defined as a high level strategy using different methods for exploring the search space. It is a general template that must be filled with problem-specific knowledge (solution representation, operators, etc.) and can tackle problems with very large search spaces. We can classify metaheuristics in two classes depending on the number of solutions they manipulate in one step: *trajectory based* and *population based* metaheuristics (Figure E.3). The techniques in the first class work with one solution that is modified in each step. On the other hand, population based metaheuristics work with

a set of solutions. Some examples of trajectory based metaheuristics are: Simulated Annealing (SA), Tabu Search (TS), GRASP, Variable Neighbourhood Search (VNS), and Iterated Local Search (ILS). Some examples of population based metaheuristics are: Evolutionary Algorithms (EA), Estimation of Distribution Algorithms (EDA), Scatter Search (SS), Ant Colony Optimization (ACO), and Particle Swarm Optimization (PSO).

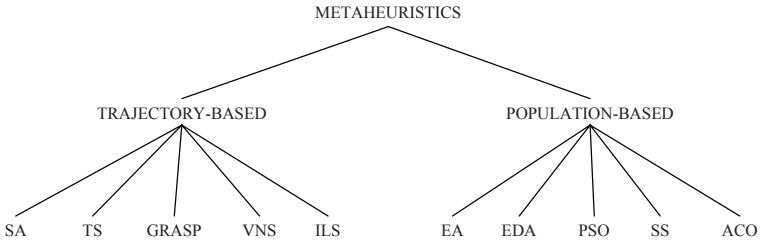


Figure E.3: Classification of metaheuristics.

In some cases, the time required to find a solution in an optimization problem can be very long even for metaheuristic algorithms. Parallelism can help not only to reduce this required time, but also to find better solutions. There exist different parallelization schemes that can be classified according to the class of metaheuristic algorithm: trajectory based and population based.

In the trajectory based metaheuristics we can identify the following three parallelization schemes:

- **Multiple executions:** several subalgorithms (homogeneous or heterogeneous) are executed in parallel. In general, each subalgorithm utilizes a different initial solution. We can distinguish different cases depending on the collaboration of the subalgorithms. If there is no collaboration among the subalgorithms the model is similar to the sequential execution, but the execution time is reduced. On the other hand, if there is collaboration among the subalgorithms, the behaviour of the parallel algorithm is different from the sequential one. The user must specify some parameters of the parallel algorithm: which information is exchanged among the subalgorithms, the frequency of the exchange, the kind of synchronization, etc.

- **Parallel movements:** the trajectory based metaheuristics explore in each step the neighbourhood of a solution and select from it the next solution. This step can be hard in terms of computational effort, since the exploration of the neighbourhood implies multiple computations of the fitness function. The parallel movements model speeds up the neighbourhood exploration by doing it in parallel. It follows a master-slave scheme in which the master (executing the algorithm) sends each slave the current solution. Each slave explores a part of the neighbourhood returning back to the master the most promising one. Among all the returned solutions the master selects one for the next step. This model does not change the way in which the search is performed but it speeds up its execution if it is executed in a parallel platform.

- **Movement speed up:** in many cases, the most time-consuming task of the algorithm is the fitness function evaluation. This computation can be broken down into several independent computations. In this model, each of these subcomputations is performed in parallel. This model does not change the way in which the search is performed.

In the case of population based metaheuristics, the parallelism appears in a natural way, since each solution of the population can be manipulated in an independent way. For this reason, the performance of the population based algorithms is usually better when they are executed in parallel. We can divide the parallelization strategies for population based metaheuristics into two categories: parallelization of the computation and parallelization of the population.

One of the most popular parallelization model following the first category is *master-slave* (or *global parallelization*). In this strategy, a central process performs the operations that affect the population (e.g., the selection in evolutionary algorithms) while the slave processes perform the operations that only affect one individual (e.g., mutation and evaluation). With this model, the search space exploration does not change with respect to the sequential algorithm but the total execution time is reduced.

Unlike the master-slave paradigm, most of the parallel population based metaheuristics in the specialized literature utilize a structured population. This scheme is especially widely spread in the case of evolutionary algorithms. Among the most popular ways of structuring the population we find the *distributed* model (coarse grain) and the *cellular* model (fine grain) [10]. In the case of distributed algorithms, the population is divided into a set of islands executing a sequential metaheuristic. The islands cooperate among them by exchanging information (usually individuals). This cooperation introduces diversity in the subpopulations, what prevents the algorithm from getting stuck in local optima. In order to completely define the algorithm the user must specify some parameters: topology of the islands (where the information is sent), migration gap (when the information is sent), migration rate (how much information is sent), migration selection (which solutions will be sent), migration replacement (how is the received information incorporated into the islands), and synchronization (whether the information exchange is synchronous or asynchronous). On the other hand, the cellular scheme of parallelization is based on the concept of neighbourhood. Each individual has a set of neighbours where the exploitation of the solutions is performed. The exploration and the diffusion of the solutions to the rest of the population are performed due to the overlapping among neighbourhoods.

E.4.1. Utilized metaheuristics

For the resolution of the problems tackled in this thesis we have selected three families of metaheuristics: evolutionary algorithms, particle swarm optimization, and ant colony optimization. In the following sections we present a brief summary of these metaheuristics.

Evolutionary algorithms

EAs [27] are metaheuristic search techniques loosely based on the principles of natural evolution, namely, adaptation and survival of the fittest. These techniques have been shown to be very effective in solving hard optimization tasks. Initially, the algorithm creates a population (set of individuals) of μ individuals randomly or by using a seeding algorithm. At each step, some individuals of the population are selected, usually according to their fitness values, and used for creating new λ individuals by means of *variation operators*. Some of these operators only affect one individual (mutation), but others generate a new individual by combining components of several of them (recombination). These last operators are able to put together good solution components that are distributed in the population, while the first one is the source of new different components. The individuals created are evaluated according to the fitness function. At the end of one step of the algorithm, a replacement operator decides which individuals will form part of the new population. This process is repeated until a stop criterion, such

as a maximum number of evaluations, is fulfilled. Depending on the representation of the solutions and the variation operators used we can distinguish four main types of EA: genetic algorithm, evolutionary strategy, evolutionary programming, and genetic programming. From these four types of evolutionary algorithms we will detail in the following the evolutionary strategy and the genetic algorithm, which are the ones we utilize in this thesis.

An evolutionary strategy (ES) is an EA in which each individual is composed of a vector of real numbers representing the problem variables ($\mathbf{x}$), a vector of standard deviations (σ) and, optionally, a vector of angles (ω). These two last vectors are used as parameters of the main operator of this technique: the Gaussian mutation. The additional parameters evolve together with the program variables to allow the search strategy to adapt itself to the landscape of the search space. For the selection operator many alternatives exist: roulette wheel, random selection, q-tournament selection, and so on (see [29] for a good review of selection operators). All the selection operators can be used with any EA since the only information needed to select the individuals is the fitness value (the selection operator does not depend on the representation used). In the recombination operator of an ES each of the three real vectors of an individual can be recombined in a different way. Some of the several possibilities for the recombination of the real vectors are discrete, intermediate, and generalized intermediate recombination [27]. The recombination operator is not so important as the mutation. In fact, we do not use recombination in our algorithm. With respect to the replacement operator, there is a special notation to indicate whether the old population is taken into account or not to form the new population. When only the new individuals are used, we have a (μ, λ) replacement; otherwise, we have a $(\mu + \lambda)$ replacement, where the best μ individuals from the union of the old population and the offspring are selected to form the new population. As it was the case in the selection operator, the replacement operator only needs the fitness value of the individuals to form the new population. Thus, we can use the two mentioned operators in any kind of EA.

A genetic algorithm (GA) is an EA that usually represents the solutions by means of binary strings (chromosome). However, other representations have been used with GA, so we can not use this feature as a distinguishing one. Unlike the ES, the recombination operator is of a great importance in GA. Some of the most popular recombination operators are the single point crossover (SPX), the double point crossover (DPX), and the uniform crossover (UX) [11]. The first one selects a random position in the chromosome and the two parents exchange their slices located before and after this point to generate two offspring. In DPX, two points are stochastically defined and the two parents exchange the slices bounded by these two limits creating again two offspring. Finally, in UX each bit of the offspring is randomly selected from the two parents. All these recombination operators can also be applied to real vectors by replacing the bits by vector components. The traditional mutation operator works on binary strings by probabilistically changing every position to its complementary value. However, for representations based on vectors of numeric values the mutation could add some random value to the components of the vector.

Particle swarm optimization

PSO maintains a set of solutions (real vectors), called *particles*, that are randomly initialized in the search space. The movement of one particle is influenced by its velocity and the positions where good solutions were found in the past. In the standard PSO, the velocity of a particle depends on the global best solution found. However, there exists another variant in which the velocity is updated according to the best solution found by a particle in the neighbourhood. The position and velocity of the particles are

updated according to the following equations:

$$v_j^i(t+1) = w \cdot v_j^i(t) + c_1 \cdot r_1 \cdot (p_j^i - x_j^i(t)) + c_2 \cdot r_2 \cdot (n_j^i - x_j^i(t)) \ , \quad \text{(E.1)}$$

$$x_j^i(t+1) = x_j^i(t) + v_j^i(t+1) \ , \quad \text{(E.2)}$$

where w is the *inertia* and controls the influence of the previous velocity value on the new one, c_1 and c_2 allows to adjust the influence of the best personal solution ($\mathbf{p}^i$) and the best neighbourhood solution ($\mathbf{n}^i$), and r_1 and r_2 values are random real numbers in the interval $[0,1]$ that are generated in each iteration. The best personal solution $\mathbf{p}^i$ of a particle i is the position with the best fitness value found by particle i. On the other hand, the best neighbourhood solution $\mathbf{n}^i$ of particle i is the position with the best fitness value found by a particle in the neighbourhood of particle i. The value w must be below 1.0 in order for the algorithm to converge. A high value for w increases the exploration of the algorithm while a low value increases the exploitation.

Ant colony optimization

ACO is inspired by the foraging behaviour of real ants. The main idea consists of simulating the ants' behaviour in a graph (the so-called *construction graph*) in order to search for the lowest cost path from an initial node to an objective one. The cooperation among the different simulated ants is a key factor in the search. This cooperation is performed indirectly by means of *pheromone trails*, which is a model of the chemicals the real ants use for their communication.

The pseudo-code of ACO consists of two main procedures executed during the search: the construction of solutions and the pheromone update. They are executed until a given stopping criterion is fulfilled, such as finding a solution or reaching a given number of steps. In the first procedure each artificial ant follows a path in the construction graph. The ant starts in an initial node and then it stochastically selects the next node according to the pheromone and the heuristic value associated with each arc (or the node itself). The ant appends the new node to the traversed path and selects the next node in the same way. This process is iterated until a candidate solution is built. In our case, when ant k is in node i it selects node j with probability

$$p_{ij}^k = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in L(i)} [\tau_{il}]^\alpha [\eta_{il}]^\beta}, \text{ if } j \in L(i) \ , \quad \text{(E.3)}$$

where $L(i)$ is the set of successor nodes for node i, and α and β are two parameters of the algorithm determining the relative influence of the pheromone trail and the heuristic value on the path construction, respectively.

In the pheromone update procedure, pheromone trails associated to arcs are modified. In our case, the pheromone trails associated to the arcs that ants traverse are updated during the construction phase (like in *Ant Colony System*) using the expression

$$\tau_{ij} \leftarrow (1-\xi)\tau_{ij} \ , \quad \text{(E.4)}$$

where ξ controls the evaporation of the pheromone during the construction phase. This mechanism increases the exploration of the algorithm, since it reduces the probability that an ant follows the path of a previous ant. After the construction phase, pheromone trails are evaporated using the expression

$$\tau_{ij} \leftarrow (1-\rho)\tau_{ij} \ , \forall (i,j) \in L \ , \quad \text{(E.5)}$$

where ρ is the *pheromone evaporation rate* and it holds $0 \leq \rho \leq 1$. Finally, the best-so-far ant deposits pheromone on the arcs it traverses:

$$\tau_{ij} \leftarrow \tau_{ij} + \Delta\tau_{ij}^{bs}, \ \forall (i,j) \in L \ , \tag{E.6}$$

where $\Delta\tau_{ij}^{bs}$ is the amount of pheromone that the ant deposits on arc (i,j). In a minimization problem (like ours) this amount is the inverse of the fitness value of the solution.

In addition, we adopt here the idea introduced in $\mathcal{MAX}-\mathcal{MIN}$ Ant Systems ($\mathcal{MM}$AS) of keeping the value of pheromone trails in a given interval $[\tau_{min}, \tau_{max}]$ in order to maintain the probability of selecting one node above a given threshold. The values of the trail limits are

$$\tau_{max} = \frac{Q}{\rho} \ , \tag{E.7}$$

$$\tau_{min} = \frac{\tau_{max}}{a} \ , \tag{E.8}$$

where Q is the inverse of the minimum fitness value. The parameter a controls the size of the interval. When one pheromone trail is greater than τ_{max} it is set to τ_{max} and, in a similar way, when it is lower than τ_{min} it is set to τ_{min}. Each time a new better solution is found the interval limits are updated consequently and all pheromone trails are checked in order to keep them inside the interval.

E.5. Methodological issues in this thesis

In this section we summarize the methodological issues performed in this thesis. We divide the description of them into three sections that are related to the three Software Engineering problems tackled.

E.5.1. Software project scheduling

In this problem our aim is to study the advantages of applying a basic genetic algorithm with binary representation to different instances of the problem. For this reason we have developed an instance generator for this problem. In the following we briefly describe this generator and the details of the genetic algorithm used.

Instance generator

In order to perform a meaningful study we must analyze several instances of the scheduling problem instead of focusing on only one, which could bias the conclusions. In order to do this, we have developed an instance generator which creates fictitious software projects after setting a set of parameters such as the number of tasks, the number of employees, etc. An instance generator is an easily parameterizable task which derives instances with growing difficulty at will. Also, using a problem generator removes the possibility of hand-tuning algorithms to a particular instance, therefore allowing greater fairness when comparing algorithms. Consequently, the predictive power of the results for the problem class as a whole is increased.

The components of an instance are: employees, tasks, skills, and the task precedence graph (TPG). Each of these components has several parameters which must be determined by the instance generator. There are two kinds of values to be generated: single numeric values and sets. For the numeric values a

probability distribution is given by the user and the values are generated by sampling this distribution. In the case of sets, the user provides a probability distribution for the cardinality (a numeric value) and then, the elements of the set are randomly chosen from its superset.

All the probability distributions are specified in a configuration file. This file is a plain text file containing attribute-value pairs. The instance generator reads the configuration file and then it generates the skills, the tasks, the TPG, and the employees, in that order. For each task, it generates the effort value and the required skill set. For each employee it generates the salary, the maximum dedication, and the set of skills.

In this thesis, we use the instance generator to study instances with different parameterizations, that is, different number of tasks, employees, and skills. The difficulty of the instances depends on these parameters. For example, we expect the instances with a larger number of tasks to be more difficult than those with a smaller set of tasks, as in real world projects. This is common sense since it is difficult to do more work with the same number of employees (without working overtime). Following this reasoning, when we increase the number of employees while maintaining the number of tasks we expect easier instances to emerge from the generator. However, sometimes these rules of thumb are not fulfilled in complex software projects, because there exist other parameters which have an influence on the difficulty of an instance. One of these parameters is the TPG: with the same number of tasks, one project can be tackled by fewer employees in the same time as another project with a different TPG.

GA details

In this section we discuss the solution representation and the fitness function utilized in the genetic algorithm. A solution can be represented with a matrix $\mathbf{X} = (x_{ij})$ of size $E \times T$ where $x_{ij} \geq 0$. The element x_{ij} is the degree of dedication of employee e_i to task t_j. If employee e_i performs task t_j with a 0.5 dedication degree s/he spends half of her/his working day on the task. If an employee does not perform a task s/he will have a dedication degree of 0 for that task. Here we have to decide how these elements are encoded. We consider that no employee works overtime, so the maximum dedication of all the employees is 1. For this reason, the maximum value for x_{ij} is 1 and therefore $x_{ij} \in [0, 1]$. On the other hand, we use a GA with binary string chromosomes to represent problem solutions. Hence we need to discretize the interval $[0, 1]$ in order to encode the dedication degree x_{ij}. We distinguish eight values in this interval which are equally distributed. Therefore, three bits are required for representing them. The matrix $\mathbf{X}$ is stored into the chromosome $\mathbf{x}$ in row major order[1]. The chromosome length is $3 \cdot E \cdot T$. To compute the fitness of a chromosome $\mathbf{x}$ we use the next expression:

$$f(\mathbf{x}) = \begin{cases} 1/q & \text{if the solution is feasible} \\ 1/(q+p) & \text{otherwise} \end{cases} , \tag{E.9}$$

where

$$q = w_{cost} \cdot p_{cost} + w_{dur} \cdot p_{dur} \ , \tag{E.10}$$

and

$$p = w_{penal} + w_{undt} \cdot undt + w_{reqsk} \cdot reqsk + w_{over} \cdot p_{over} \ . \tag{E.11}$$

The fitness function has two terms: the cost of the solution (q) and the penalty for unfeasible solutions (p). The two terms appear in the denominator because the goal is to minimize them, i.e., maximize $f(\mathbf{x})$.

[1]We use $\mathbf{x}$ to refer to the chromosome (binary string) which represents the matrix solution $\mathbf{X}$.

The first term is the weighted sum of the project cost and duration. In this term, w_{cost} and w_{dur} are values weighting the relative importance of the two objectives. These weights allow the fitness to be adapted according to the needs of the project manager. For example, if the cost of the project is a primary concern, the corresponding weight must be high. However, we must take into account the order of magnitude of both the project cost and duration. This can be done by setting all the weights to one initially and then executing the GA several times. Next, the cost weight is divided by the average project cost and the duration weight is divided by the average project duration. In this way, the weighted terms related to project cost and duration are in the same order of magnitude. At this point the project manager can try different weight values in order to adapt the solutions proposed by the GA to her/his requirements.

The penalty term p is the weighted sum of the parameters of the solution that make it unfeasible, that is: the overwork of the project (p_{over}), the number of tasks with no employee associated ($undt$), and the number of skills still required in order to perform all project tasks ($reqsk$). Each of these parameters is weighted and added to the penalty constant w_{penal}. This constant is included in order to separate the fitness range of the feasible solutions from that of the unfeasible ones.

E.5.2. Test case generation

In this section we describe the proposed test case generator and the whole test case generation process. We must define a test adequacy criterion in order to formalize the objective of the generator, that is, we need a condition which any test cases set should fulfil in order to be considered an adequate testing set. In this thesis we use the *condition coverage* test adequacy criterion. This criterion requires that all the atomic predicates of the test program be evaluated to the two boolean values: *true* and *false*. Other well-known test adequacy criteria are *branch coverage*, requiring all the branches to be taken, and *statement coverage*, in which all the program statements must be executed. It is important to note that the condition coverage criterion is harder than branch and statement coverage in the case of programs with short-circuit evaluation of boolean expressions (this is our case). That is, if we find a set of test cases that makes all the atomic conditions of a C program take the possible boolean values, then we can ensure that all the feasible branches will be taken and, in consequence, all the reachable statements will be executed. However, the opposite is not true, i.e., executing all the reachable statements or taking all the feasible branches does not ensure that all the atomic predicates will take the feasible boolean values. This fact makes condition coverage equivalent to *condition-decision coverage* [206] in programs with short-circuit evaluation of boolean expressions and this is the reason why it was selected here.

Our test case generator breaks down the global objective (the condition coverage) into several partial objectives consisting of making one atomic predicate take one boolean value. Then, each partial objective can be treated as an optimization problem in which the function to be minimized is a distance between the current test case and a test case satisfying the partial objective. For this reason we call this function the *distance function*.

Distance function

Following on from the discussion in the previous section, we have to solve several minimization problems: one for each atomic predicate and boolean value. The distance function to be minimized depends on the expression of the particular atomic predicate and the values of the program variables when the condition is reached. In Table E.1 we show the distance functions for all kinds of atomic predicates and boolean values.

Tabla E.1: Distance functions for different kinds of predicates and boolean values. The variables a and b are numeric variables (integer or real)

Atomic predicates	*true* expression	*false* expression
$a < b$	$a - b$	$b - a$
$a <= b$	$a - b$	$b - a$
$a == b$	$(b - a)^2$	$(1 + (b - a)^2)^{-1}$
$a != b$	$(1 + (b - a)^2)^{-1}$	$(b - a)^2$
a	$(1 + a^2)^{-1}$	a^2

The distance function of a particular partial objective can only be computed if the program flow reaches the associated atomic condition, because it depends on the values of the program variables at that point of the program execution. For this reason, when the condition is not reached, the distance function takes the maximum value for a real number in a machine using 64-bit IEEE 754 representation (that is, $2^{1024} - 2^{971}$).

Program instrumentation

We instrument the source code of the program in order to get information about the value of the distance function and the conditions traversed in a program execution. The instrumentation must be done carefully to avoid a change in the program behaviour. This step is performed automatically (not manually) by our application that parses the C source program and generates a modified C source program with the same original behaviour. This application transforms each atomic predicate into an expression that is evaluated to the same value as the original predicate. This expression has a (neutral) side effect: it informs about the boolean value it takes, and the distance value related to the predicate. If `<cond>` is an atomic predicate in the original program, the associated expression used instead of the original predicate in the modified program is:

```
((<cond>)?
    (inform(<ncond>,1),(distance(<ncond>,<true_expr>,<false_expr>),1)):
    (inform(<ncond>,0),(distance(<ncond>,<true_expr>,<false_expr>),0)))
```

where `<ncond>` is the number of the atomic predicate in the program, `<true_expr>` and `<false_expr>` are the fitness expressions for the true and false values of the predicate, `inform` is a function that informs the test case generator about the condition reached and its value, and `distance` is a function that informs about the distance value. This transformation does not modify the functional behaviour of the program unless the original atomic predicate has side effects.

In the evaluation of a test case, when the test case generator executes the modified test program with such a test case, a report of the atomic predicates reached and the distance values are computed and transmitted to the generator. With this information the generator builds a coverage table where it stores, for each predicate, the set of test cases that makes the predicate true and false throughout the process. That is, for each predicate the table stores two sets: the *true* set and the *false* set. This table is an important internal data structure that is consulted during the test case generation. We say that a predicate is *reached* if at least one of the sets associated with the predicate is non-empty. On the other hand, we say that a predicate is *covered* if the two sets are non-empty.

Test case generation process

Once we have presented the distance functions and the instrumentation details we can now focus on the test case generator itself. At the beginning of the generation process some random test cases (10 in this thesis) are generated in order to reach some predicates. Then, the main loop of the generator begins and the generator selects a partial objective not covered. The test case generator does not select the partial objectives in a random way. As we said before, the distance function of each partial objective depends on the expression of the particular atomic predicate and the values of the program variables when the predicate is reached. This means that the distance can only be computed if the program flow reaches the atomic predicate. Thus, the test case generator always selects a partial objective with an associated predicate reached by a previous test case.

When the partial objective is selected, it utilizes the optimization algorithm to search for test cases making that predicate take the value not yet covered. The optimization algorithm is seeded with at least one test case reaching the mentioned predicate. The algorithm tries different test cases and uses the distance value to guide the search. During this search, test cases covering other conditions can be found. These test cases are also used for updating the condition table. In fact, we can set the stop condition of the optimization algorithm to cover a partial objective not yet covered (we study this alternative in the experimental chapter). As a result, the optimization algorithm can or can not find a solution. In any case, when the algorithm stops, the main loop starts again and the test generator selects a different partial objective. This scheme is repeated until total condition coverage is obtained or a maximum number of consecutive failures of the optimization algorithm is reached. When this happens the test case generator exits the main loop and stops.

When we use a decentralized optimization algorithm, such as a distributed evolutionary algorithm, we have several subalgorithms working independently of each other with some sparse interconnections among them. In this case we can assign a different partial objective to each subalgorithm. If all the partial objectives are solved approximately at the same time, the search could be accelerated. This alternative is also analyzed in the experimental part of this thesis.

Coverage measures

We must discuss the coverage measure used to report the generator results. The simplest coverage measure is the ratio between the covered partial objectives and the total number of partial objectives which, expressed as a percentage, gives the *coverage percentage*. Although the coverage percentage is the simplest way of reporting generator efficacy, it is not the more appropriate measure. The reason is that it is impossible for some programs to reach total coverage, because of the presence of unreachable partial objectives. In this case a global coverage loss is produced irrespectively of the techniques used for testing. For example, an infinite loop has a predicate that is always true and never false. Another example is the condition `(sign(x)>2)`, where the function `sign` can only return three values: -1, 0, +1. In these cases there are pairs (predicate, boolean) that are *unreachable* and no test case generator can reach 100 % of condition coverage due to the test program itself. In this situation we say that there is a *code-dependent coverage loss*. However, there is another factor which may produce an unavoidable coverage loss: the environment in which the program is executed. One example of this situation is related to dynamic memory allocation. Let us suppose that a program allocates some dynamic memory and then it checks if this allocation has failed. Most probably it succeeds for all the tests run on the program, and the corresponding predicate gets only one value. In this case we talk about an *environment-dependent coverage loss*. When one of these situations appears in a program no test case generator can get total

coverage and it may appear to be ineffective when, in fact, it is not. For example, we can get a low coverage percentage in one program but this low percentage might happen to be the maximum coverage that it is possible to obtain.

We search for a coverage measure taking into account the coverage loss, a measurement that gets its known maximum value when it is impossible to get a new test case covering more partial objectives. For this reason we have introduced another measure that we call *corrected coverage*. This measure is defined as the ratio between the number of covered and reachable partial objectives. In this measure the unreachable partial objectives are not taken into account, without any loss of information or drawback for the testing task. This measure is useful for comparing the performance of the test case generator in different programs. In this way, we can sort the programs in order of difficulty. If we use the simple condition coverage to compare the programs we can classify a program as difficult when it has a lot of unreachable partial objectives but the remaining partial objectives can be covered easily. However, the computation of the corrected coverage requires knowing the unreachable partial objectives. In small programs these partial objectives can be easily determined, but it can be a difficult task in larger programs (it could be a NP-hard problem itself). In these cases, the corrected coverage measure is not practical. In this thesis, we decide by human observation whether a partial objective is reachable or not and the unreachable partial objectives are then communicated to the test case generator via configuration files. We use the corrected coverage in the experiments in order to avoid code-dependent coverage loss. The environment-dependent coverage loss is more difficult to avoid and the new measure does not take this loss into account.

At this point we need to modify the way in which the partial objective selection is performed at the beginning of the main loop of the test case generator in order to count the correct number of evaluations required for the coverage measured. As a matter of fact, in order to avoid unnecessary work, the unreachable partial objectives should not be taken into account in the partial objective selection.

Details of the metaheuristic algorithms utilized

The original contributions in this problem have been, first, the application of metaheuristic techniques never used before in test case generation, and, second, a detailed study of the performance of distributed evolutionary algorithms. The two new applied metaheuristics are ES and PSO, in which the solutions are represented by means of real vectors. This representation allows us to completely explore the search space. This contrasts with other works in which the values of the input variables are constrained to bounded integer values, reducing the search space [70, 244, 261]. Since GA has been the most popular proposal in evolutionary testing, we include in the study a GA with real representation of the solutions. Concerning the study of distributed metaheuristics we selected ES, GA and their distributed counterparts. In the following we present some details of the selected metaheuristics.

- **ES**: the input arguments of a program can be integer values. In this case, the value in the solution vector associated to an integer is rounded before the solution is evaluated (the program executed).

- **PSO**: as in the previous case, when a program input argument is an integer value it must be rounded. This algorithm can converge very fast. For this reason we decided to add a mutation operator that adds a random value to the velocities of half of the swarm (randomly selected) if the best solution is not improved after one step. The magnitude of this perturbation increases in an exponential-like way (it is multiplied by 10) as the number of steps without improvement increases.

- **GA**: the solutions are represented with a vector of numbers that can be integer or real so, in this case, it is not required to round the numbers. The most popular recombination operators (SPX, DPX, and UX) can be applied to this representation using vector components instead of bits. The mutation operator utilized adds a random number to each component following a normal distribution with mean 0. The standard deviation is a parameter of the operator.

Representation and fitness function

The fitness function used in the evolutionary search is not exactly the distance function. We want to avoid negative fitness values in order to be able to apply selection operators that depend on the value of the fitness function such as the Roulette Wheel selection. For this reason we transform the distance value by using an `arctan` function that maps the whole real number set into a bounded interval. The resulting fitness function is:

$$f(\mathbf{x}) = \pi/2 - \arctan(\mathit{distance}(\mathbf{x})) + 0.1 \ . \tag{E.12}$$

In the previous expression we multiply the `arctan` result by -1 since our EA software is designed to maximize the fitness function. Thus, we need to add the value $\pi/2$ to the expression in order to always get a positive result. Finally, the 0.1 value is added so that negative values are not obtained when there is precision loss in the difference calculation.

E.5.3. Search for safety property violations in concurrent systems

The problem of searching for safety property violations in concurrent systems can be translated into a search of a node in a graph. For this search we can use heuristic information. From the metaheuristics, the one that performs a search in a graph is ant colony optimization. For this reason, we propose to apply ACO to this problem. However, the state explosion affects also to the ACO algorithms. The ACO models we can find in the literature are not suitable in problems in which the construction graph has more than 10^6 nodes (i.e. 10^{12} arcs). They are also not suitable when the amount of nodes is not known beforehand because they are dynamically generated as the search progresses. We try to solve here such a kind of problem. In effect, the number of states of a concurrent system is usually very large even in small models. In order to solve all these obstacles that arise when working with large graphs, we have developed a new ACO model called ACOhg (ACO for huge graphs) that is able to tackle combinatorial optimization problems with an underlying construction graph of unknown size that is built as the search progresses. The main ideas this model introduces are related to the length of the ant paths, the fitness function, and the memory consumption of pheromone trails. We tackle these points in the following paragraphs.

Length of the ant paths

In order to avoid the, in general unviable, construction of complete candidate solutions, we limit the length of the paths traversed by ants in the construction phase. That is, when the path of an ant reaches a given maximum length λ_{ant}, the ant is stopped. In this way, the construction phase can be performed in a bounded time and with a bounded amount of memory. However, the limitation of the ant path length implies that most (if not all) of the paths are partial solutions and therefore we need a fitness function that is able to evaluate partial solutions.

The limitation in the ant path length solves the problem of the "wandering ants" but introduces a new one. There is a new parameter for the algorithm (λ_{ant}) whose optimal value is not easy to establish *a priori*. If we select a value smaller than the depth[2] of all the objective nodes, the algorithm will not find any solution to the problem. Thus, we must select a value larger than the depth of one objective node (if known). This is not difficult when we know where the objective node is, but the usual scenario is the opposite one. In the last case, two alternatives are proposed.

The first consists in dynamically increasing λ_{ant} during the search if no objective node is found. At the beginning, a low value is assigned to λ_{ant} and it is increased in a given quantity δ_l every certain number of steps σ_i. In this way, the length will be hopefully high enough to reach at least one objective node. This is called *expansion technique*. This mechanism is similar to the one used by IDA* algorithm and can be useful when the depth of the objective nodes is not very high. Otherwise, the length of the ant paths will increase too much and the same happens with the time and the memory required to build the paths, since it will approach the behaviour of a regular ACO incrementally.

The second alternative consists in repeatedly re-starting the path construction of the ants from different nodes during the search. That is, at the beginning, the ants are placed on the initial nodes of the graph and the algorithm is executed during a given number of steps σ_s (called *stage*). If no objective node is found, the last nodes of the paths constructed by the ants are used as starting nodes for the next ants. In the next steps (the second stage) of the algorithm, the new ants traverse the graph starting in the last nodes of paths computed in the first stage. In this way, the new ants start at the end nodes of previous ant paths trying to go beyond in the graph. This mechanism is called *missionary technique*. The length of the ant paths (λ_{ant}) is kept always constant and the pheromone trails can be discarded from one stage to another in order to keep almost constant the amount of computational resources (memory and CPU time) in all the stages. The assignment of ants to starting nodes at the beginning of one stage is performed in several phases. First, we need to select the paths of the previous stage whose last visited nodes will be used as starting points in the new stage. For this, we store the best paths (according to the fitness value) found in the previous stage. We denote with s the number of stored paths. Once we have the set of starting nodes, we need to assign the new ants to those nodes. For each new ant we select its starting node using roulette selection; that is, the probability of selecting one node is proportional to the fitness value of the solution associated with it. The number of stored paths between two stages, s, is a parameter of the algorithm set by the user. It must hold $1 \leq s \leq \sigma_s \cdot colsize$, that is, there must be at least one stored path (the ants of the following stage need at least one starting node) and the upper bound is the maximum number of paths the algorithm is able to construct in σ_s steps: $\sigma_s \cdot colsize$.

Fitness function

The objective of ACOhg is to find a low cost path between an initial node and an objective one. For the problem tackled here the cost of a solution is its length, but, in general, cost and length (number of components) of a solution can be different, and for this reason it is safer to talk about cost. Considering a minimization problem, the fitness function of a complete solution can be the cost of the solution. However, as we said above, the fitness function must be able to evaluate partial solutions. In this case, the partial solution cost is not a suitable fitness value since a low cost partial solution can be considered better than one high cost complete solution. This means that low cost partial solutions are awarded and the fitness function will not suitably represent the quality of the solutions. In order to avoid this problem we penalize the partial solutions by adding a fixed quantity p_p to the cost of such solutions.

[2]The *depth* of a node in the construction graph is the length of the shortest path from an initial node to it.

In our case, for the problem at hands, we compute the fitness function in the following way. Firstly, given a (partial) solution (path), we compute a lower bound of the length of a complete solution that is an extension of the solution. This lower bound is computed as the sum of the length of the partial solution and the heuristic value of the last node of the path. Secondly, we add a penalty term when the solution is not complete.

In some problems, solutions (paths) with cycles are not desirable. This happens in our problem. One path with a cycle can be replaced by another shorter path without cycles. For this reason, we want to avoid cycles in partial solutions. In ACOhg we avoid cycles in the following way. During the construction phase, when an ant builds its path, it can stop due to three reasons: the maximum ant path length λ_{ant} is reached, the last node of the ant path is an objective node, or all the successors nodes are in the ant path (visited nodes). This last condition, which is used in order to avoid the construction of paths with cycles, has an undesirable side effect: it awards paths which form a cycle. In effect, the premature stop of the ant during the construction phase produces a shorter partial solution; which are favored by the algorithm, since it awards shorter paths. In order to avoid this situation we penalize those partial solutions whose path length is shorter than λ_{ant}.

The total penalty expression we use for taking into account the above discussed issues is

$$p = p_p + p_c \frac{\lambda_{ant} - l}{\lambda_{ant} - 1} \ , \tag{E.13}$$

where p_p is the penalty due to the incompleteness of the solutions, p_c is a penalty constant related to the cycle formation, and l is the ant path length. The second term in (E.13) makes the penalty higher in shorter cycles. The intuition behind this is that longer cycles are nearer of a path without a cycle. For this reason, we add to p_p the maximum cycle penalty (p_c) when the ant length is the minimum ($l = 1$) and no cycle penalty is added when there is no cycle ($l = \lambda_{ant}$). In conclusion, the final fitness function we use in the experiments for evaluating partial solutions is:

$$f(x) = l + H(j) + p_p + p_c \frac{\lambda_{ant} - l}{\lambda_{ant} - 1} \ , \tag{E.14}$$

where j is the last state of the path and $H(j)$ is the heuristic value for state j. The fitness value for a complete solution is just the ant path length l.

Pheromone trails

In ACOhg, the pheromone trails are stored in a hash table where only the pheromone values of the edges traversed by the ants are stored. This is true for both, expansion and missionary techniques. However, as the search progresses, the memory required by pheromone trails can increase until inadmissible values. We can avoid this by removing from the hash table the pheromone trails with a low influence on the ant construction. This is done in the missionary technique when a new stage begins. In a new stage, the region of the graph explored by the ants is different with respect to previous stages[3]. This means that pheromone trails used in previous stages are not useful in the current one and can be discarded without a negative influence on the results.

[3]It is possible to find some overlap among the regions explored in different stages, but this overlap can be very small.

Construction graph

One last issue to take into account in ACOhg is the generation of the construction graph. The graph is generated during the search itself, and this means that more memory is required as the search progresses. We need to be careful with the implementation in order to save memory. For example, if the size of one node in memory is larger than one pointer we can reduce the memory required by keeping only one copy of the node in memory and using pointers to it wherever they are required in the future. In this case, we have to avoid duplicated nodes. However, if the node size is equal to or less than one pointer, the previous mechanism is not useful and makes the implementation inefficient. On the other hand, if pheromone trails are discarded after one stage of the missionary technique, the nodes adjacent of the removed arcs can be also removed if they are not referenced by any other arc or ant.

The above mentioned ideas are mechanisms for memory reduction that are used to maximize the portion of the graph that can be stored in memory, since it is there where the algorithm can work with high efficacy to maximize the quality of the partial solutions.

E.6. Application of genetic algorithms to software project scheduling

For the experimental study we have generated a total of 48 different instances with the instance generator and we have solved them with a genetic algorithm. We performed 100 independent runs for each instance in order to get statistically meaningful conclusions. We have grouped the instances into five benchmarks. In the first three groups we change only one parameter of the problem. With these studies we want to analyze how sensitive the results obtained are to the variation of these parameters. In the last two groups we change several parameters at the same time. In this way, we study whether the results change in the way suggested by the studies of the first three groups.

The results show that the instances with more tasks are more difficult to solve and their solutions are more expensive. In the same way, the projects with a larger number of employees are easier to tackle and can be driven to a successful end in a shorter time. However, the relationship between employees and cost is not that simple: in some cases it is direct and in other cases it is inverse.

This problem is essential for the Software Engineering industry nowadays and automatically finding "good" solutions to it can save software companies lots of time and money. A software manager can study different scenarios with such an automatic tool to take decisions on the best project for her/his company. Furthermore, in our approach, s/he can adjust the fitness weights to better represent particular real world projects.

With the study we present, we show that GAs can be a very useful tool for project management in Software Engineering.

GAs can assign people to the project tasks in a nearly optimal way and they allow us to try different configurations changing the relative importance of the cost and duration of the project with no risk for the actual project because it is performed inside a computer. Although the project model is very simple it can serve as a first step in the application of evolutionary algorithms to *in silico* experiments in Software Engineering.

E.7. Application of metaheuristics to test case generation

In this chapter we analyze, first, the application of evolutionary algorithms with centralized and decentralized population to the problem of test case generation. In particular, we have compared a distributed ES and a distributed GA against their panmictic versions using a benchmark of twelve test programs implementing some fundamental algorithms in computer science. The results show that the decentralized versions have no statistically significant advantage over the panmictic versions, neither in terms of the coverage nor in effort. This is an unexpected observation since much research exists reporting a higher degree of accuracy for the decentralized approach. The conclusion is that the decentralized algorithms should maybe focus on cooperating in a different way.

In order to check that the conclusions obtained in the distributed approach are not the result of a naive configuration, we have studied the influence of several parameters of the dES-based test case generator. These parameters are the search mode, the stop condition, the number of seeds used in the dES islands, and the migration gap. The results state that, by searching for the same partial objective in all the islands, we can outperform the results with respect to the version that searches for different partial objectives in the islands. On the other hand, the stop condition that involves stopping when any new partial objective is covered does not have a clear advantage over the one in which only the coverage of the searched partial objective is used to stop the algorithm. Analyzing the number of seeds used in the initial population of dES, we discovered that the best results are obtained with one single seed. Finally, a high migration gap seems to benefit the search and this confirms that the distributed approach, as utilized in this thesis, is not good for this problem.

In a second stage, we show the results obtained with particle swarm optimization for the test case generation problem and compare them with the ones of ES, GA, and a random generator. The results show that PSO and ES have a similar efficacy in general. In some programs PSO is better than ES and in others ES obtains the best results. Anyway, we conclude that both algorithms are better than GA, which is the one utilized in many works of the specialized literature.

This fact opens a promising research line related to the application of PSO and ES to the evolutionary testing, where GA has been up to the moment a standard *de facto*. Furthermore, the number of parameters of an ES is smaller than the one of the other algorithms and, for this reason, we think that it can be more appropriate for automatic tools that must be used, in general, by people with no knowledge about evolutionary algorithms.

E.8. Application of ACOhg to the search for safety property violations in concurrent systems

In this chapter we show the results obtained after the application of ACOhg to the problem of searching for safety property violations in concurrent systems. For the experiments we have selected 10 Promela models implementing faulty concurrent systems previously reported in the literature by Edelkamp et al. [94]. All these models violate a safety property.

First, we have studied the influence on the results of the maximum ant path length λ_{ant}. The experiments show that the hit rate (probability of finding an error trail) increases with λ_{ant}. The same happens with the length of the error trails found. In addition, the memory and CPU time required for finding an error trail decrease when the ants are allowed to build longer paths.

With respect to the missionary and expansion techniques we conclude that both techniques are tied when the length of the error trails is compared. If we consider the memory required for finding an error trail, we can observe that the expansion technique requires more memory than the missionary technique. Thus, we conclude that the missionary technique is more efficient than the expansion technique with respect to the memory consumption. The hit rate is similar in both techniques when $\sigma_i = \sigma_s$ is below 10. However, for larger values, the efficacy of the missionary technique is higher.

We have compared the results of ACOhg against the ones of well known exhaustive algorithms used for this problem in the literature. From these exact algorithms we distinguish two groups according to their behaviour: the ones that require a low amount of computational resources obtaining long error trails, and the ones that obtain optimal error trails with a very high resource consumption. Our proposal achieves an optimal trade-off between those two extremes for the concurrent systems analyzed: it obtains almost optimal solutions with a low amount of memory. ACOhg is also the most robust algorithm of the experiments. It has a similar behaviour for all the models. On the contrary, the behaviour of the exhaustive state-of-the-art algorithms depends to a large extent on the model they solve. For this reason, we point out the ACOhg algorithm as a very promising technique for finding errors in concurrent programs.

We also have combined ACOhg with partial order reduction (POR), a technique that exploits the commutativity of asynchronous systems in order to reduce the size of the state space in model checking. We can notice in the results that the POR technique combined with ACOhg reduces in all the cases the average amount of computational resources (memory and CPU time) required to find a solution. We conclude that ACOhg is not only compatible with POR, but they together can greatly reduce the computational effort.

Finally, we have compared ACOhg with a GA utilized by Godefroid and Khurshid in the specialized literature for this task [111]. For these experiments, we used the same protocols (dining philosophers and Needham-Schroeder) as in [111] and the results of ACOhg are much better than the ones of the GA published in the mentioned work. ACOhg is able to find always an error (100 % hit rate) in both protocols while GA finds an error only in 52 % of the cases in the dining philosophers and 3 % in Needham-Schroeder. With respect to the execution time, the results reveal a large difference between the algorithms: ACOhg is between two or three order of magnitude faster than GA, even taking into account the different machines used in the experiments. However, although it seems that ACOhg is better than GA, we cannot confirm this statement due to the differences in the protocol implementation and the model checkers utilized. Hence, we just notice here the results obtained by Godefroid and Kurshid with their GA and we defer for a future work a fair comparison between ACOhg and GA.

E.9. Conclusions and future work

In this PhD dissertation we study the application of metaheuristic techniques to some optimization problems from the Software Engineering domain. The problems are: software project scheduling, test case generation, and search for safety property violations in concurrent systems. The metaheuristic techniques utilized for solving these problems are: genetic algorithm, evolutionary strategy, particle swarm optimization, and ant colony optimization. The problem of searching for safety property violations in concurrent systems suffers from an important drawback: the Büchi automata that must be explored in order to solve this problem usually have a very large size and cannot be completely stored in the memory of one computer. In order to overcome this drawback, we have developed a new algorithmic model based on ant colony optimization, called ACOhg, that is able to tackle the problem using only a reduced amount of computational resources.

We have utilized a genetic algorithm with binary representation for solving the software project scheduling problem. The algorithm proposes scheduling solutions for different software projects that are automatically generated by an instance generator. This generator, developed along the research, allows us to analyze the solutions proposed by the genetic algorithm for projects with very different features, offering results that help the project manager to take decisions about the scheduling. The experiments show that the genetic algorithm can be a very useful technique for software project scheduling.

In the test case generation problem we have studied the application of parallel algorithms with decentralized population. The results show that the decentralization model used does not improve the search for test cases in the instances tackled. We have also utilized two algorithms that have never been used for this problem in the past: evolutionary strategies and particle swarm optimization. In addition to the novelty, the application of these algorithms to the test case generation has revealed that both techniques outperform in efficacy the genetic algorithms, widely utilized for this task during many years.

The algorithmic proposal selected for searching for safety property violations in concurrent systems, ACOhg, is able to find short error trails (good quality) with a very reduced amount of memory and time. We have performed a study of the different alternatives of the algorithmic model in order to select a suitable configuration for the problem. The results show that ACOhg is the best trade-off between solution quality and memory required. We have combined ACOhg with a technique for reducing the number of states of the automaton to explore: partial order reduction. The results show an advantage of the combination of both techniques in the analyzed models.

As future work, there are some research lines concerning the three problems and the algorithms that appear after this PhD dissertation.

With respect to the software project scheduling, the model of the problem can be extended to include some aspects of real world projects, such as other resources, staff change, etc. This problem can also be solved using multi-objective algorithms in order to avoid the weights that project managers have to adjust in the current mono-objective formulation.

In the test case generation problem, a first future research line is the study in depth of the reasons of the unexpected low efficiency of the distributed approach presented in this dissertation. This study can help in the design of a suitable distributed technique. Another future research line (in progress) is the application of metaheuristic techniques to extract test cases from high level models of the software, like software usage models.

For the problem of searching for safety property violations in concurrent systems, we can extend the approach to liveness properties. We can also combine ACOhg with other techniques for reducing the required memory, such as state compression and symmetry reduction. On the other hand, we plan to include ACOhg in other model checkers, like Java PathFinder.

Finally, in the domain of metaheuristics, we can export the ideas used in ACOhg for solving problems with very large underlying graphs to other metaheuristics. This way, we can apply other metaheuristics to model checking.

Bibliografía

[1] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference (GECCO 2004), LNCS 3103*, volume 3103, pages 1338–1349. Springer, 2004.

[2] J. Aguilar-Ruiz, I. Ramos, J. C. Riquelme, and Miguel Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875–882, 2001.

[3] J. S. Aguilar-Ruiz, J. C. Riquelme S., and Isabel R. Natural evolutionary coding: An application to estimating software development projects. In *GECCO Late Breaking Papers*, pages 1–8, 2002.

[4] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms*. John Wiley & Sons, October 2005.

[5] E. Alba and F. Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers & Operations Research*, 177(11):2380–2401, June 2007.

[6] E. Alba and J. F. Chicano. Management of software projects with GAs. In *Metaheuristics International Conference (MIC-2005)*, pages 13–18, Viena, Austria, August 2005.

[7] E. Alba and C. Cotta. Optimización en entornos geográficamente distribuidos. Proyecto MALLBA. In *Actas del Primer Congreso Español de Algoritmos Evolutivos y Bioinspirados (AEB'02)*, pages 38–45, Mérida, 2002.

[8] E. Alba, G. Leguizamon, and G. Ordonez. Analyzing the behavior of parallel ant colony systems for large instances of the task scheduling problem. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, April 2005.

[9] E. Alba, G. Luque, and S. Khuri. Assembling DNA Fragments with Parallel Algorithms. In B. McKay, editor, *CEC-2005*, pages 57–65, Edinburgh, UK, 2005.

[10] E. Alba and M. Tomassini. Parallelism and Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, October 2002.

[11] E. Alba and J. M. Troya. A survey of parallel distributed genetic algorithms. *Complexity*, 4:31–52, 1999.

[12] E. Alba and J. M. Troya. Gaining new fields of application for OOP: the parallel evolutionary algorithm case. *Journal of Object Oriented Programming*, December (web version only) 2001.

[13] E. Alba and J.M. Troya. Genetic Algorithms for Protocol Validation. In *Proceedings of the PPSN IV International Conference*, pages 870–879, Berlin, 1996. Springer.

[14] Enrique Alba. *Análisis y Diseño de Algoritmos Genéticos Paralelos Distribuidos.* PhD thesis, University of Málaga, 1999.

[15] Enrique Alba and Francisco Chicano. ACOhg: Dealing with huge graphs. In *Proceedings of the Genetic and Evolutionary Conference*, pages 10–17, London, UK, July 2007. ACM Press.

[16] Enrique Alba and Francisco Chicano. Ant colony optimization for model checking. In *EUROCAST 2007 (LNCS)*, volume 4739 of *Lecture Notes in Computer Science*, pages 523–530, Gran Canaria, Spain, February 2007.

[17] Enrique Alba and Francisco Chicano. Finding safety errors with ACO. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, pages 1066–1073, London, UK, July 2007. ACM Press.

[18] Enrique Alba and Francisco Chicano. Software project management with GAs. *Information Sciences*, 177(11):2380–2401, June 2007.

[19] Enrique Alba, Francisco Chicano, and Stefan Janson. Testeo de software con dos técnicas metaheurísticas. In *XI Jornadas de Ingeniería del Software y Bases de Datos*, pages 109–118, Sitges, Barcelona, Spain, October 2006.

[20] Enrique Alba and J. Francisco Chicano. Software testing with evolutionary strategies. In *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques*, volume 3943 of *LNCS*, pages 50–65, Heraklion, Crete, Greece, September 2005.

[21] D. Alberts. The economics of software quality assurance. In *Proceedings of the 1976 National Computer Conference*, volume 45, pages 433–442. AFIPS Press, 1976.

[22] Bowen Alpern and Fred B. Schneider. Defining liveness. *Inform. Proc. Letters*, 21:181–185, 1985.

[23] P. Ammann, P. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods*, pages 46–54, Brisbane, Australia, December 1998. IEEE Computer Society Press.

[24] Giuliano Antoniol, Massimiliano Di Penta, and Mark Harman. A robust search-based approach to project management in the presence of abandonment, rework, error and uncertainty. In *IEEE METRICS*, pages 172–183. IEEE Computer Society, 2004.

[25] Giulio Antoniol, Massimiliano Di Penta, and Mark Harman. Search–based techniques for optimizing software project resource allocation. In *Genetic and Evolutionary Computation Conference (GECCO 2004)*, 2004.

[26] Giulio Antoniol, Massimiliano Di Penta, and Mark Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *IEEE International Conference on Software Maintenance*, pages 240–249, 2005.

[27] T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, New York, 1996.

[28] T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York NY, 1997.

[29] T. Bäck, D. B. Fogel, and Z. Michalewicz, editors. *Evolutionary Computation 1. Basic Algorithms and Operators*, volume 1. IOP Publishing Lt, 2000.

[30] Paul Baker, Mark Harman, Kathleen Steinhöfel, and Alexandros Skaliotis. Search based approaches to component selection and prioritization for the next release problem. In *ICSM*, pages 176–185. IEEE Computer Society, 2006.

[31] André Baresel, David Wendell Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, 2004.

[32] Andre Baresel and Harmen Sthamer. Evolutionary testing of flag conditions. In *Genetic and Evolutionary Computation (GECCO-2003)*, volume 2724, pages 2442–2454. Springer-Verlag, 2003.

[33] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1329–1336. Morgan Kaufmann Publishers, 2002.

[34] André Baresel, Harmen Sthamer, and Joachim Wegener. Applying evolutionary testing to search for critical defects. In *GECCO (2)*, volume 3103, pages 1427–1428. Springer, 2004.

[35] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[36] D. Bird and C. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[37] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[38] Barry Boehm and Rony Ross. Theory-W Software Project Management: Principles and Examples. *IEEE Transaction on Software Engineering*, 15(7):902–916, July 1989.

[39] Barry W. Boehm. A spiral model of software development. *Computer*, 21(5):61–72, May 1988.

[40] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithms. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1337–1342. Morgan Kaufmann Publishers, 2002.

[41] Leonardo Bottaci. Predicate expression cost functions to guide evolutionary search for test data. In *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 2455–2464. Springer-Verlag, 2003.

[42] Salah Bouktif, Giuliano Antoniol, Ettore Merlo, and Markus Neteler. A novel approach to optimize clone refactoring activity. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1885–1892. ACM Press, 2006.

[43] Salah Bouktif, Houari Sahraoui, and Giuliano Antoniol. Simulated annealing for improving software quality prediction. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1893–1900. ACM Press, 2006.

[44] H.J. Bremermann. *Self-Organizing Systems*, chapter Optimization Trough Evolution and Resombination, pages 93–106. Spartan Books, Washington DC, 1962.

[45] Lionel C. Briand, Jie Feng, and Yvan Labiche. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 43–50. ACM Press, 2002.

[46] Lionel C. Briand, Yvan Labiche, and Marwa Shousha. Stress testing real-time systems with genetic algorithms. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1021–1028. ACM Press, 2005.

[47] Renée C. Bryce and Charles J. Colbourn. Constructing interaction test suites with greedy algorithms. In *ASE*, pages 440–443. ACM, 2005.

[48] Renée C. Bryce, Charles J. Colbourn, and Myra B. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE*, pages 146–155. ACM, 2005.

[49] Oliver Buehler and Joachim Wegener. Evolutionary functional testing of an automated parking system. In *Proceedings of the International Conference on Computer, Communication and Control Technologies*, Orlando, Florida, August 2003.

[50] Oliver Buehler and Joachim Wegener. Evolutionary functional testing of a vehicle brake assistant system. In *Proceedings of the 6th Metaheuristic International Conference*, pages 157–162, Vienna, Austria, August 2005.

[51] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4), April 1994.

[52] Colin J. Burgess and Martin Lefley. Can genetic programming improve software effort estimation? A comparative evaluation. *Information & Software Technology*, 43(14):863–873, 2001.

[53] E. Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms*, chapter 7. Migration, Selection Pressure, and Superlinear Speedups, pages 97–120. Kluwer, 2000.

[54] Carl K. Chang, Mark J. Christensen, and Tao Zhang. Genetic Algorithms for Project Management. *Annals of Software Engineering*, 11:107–139, 2001.

[55] Yoonsik Cheon and Myoung Kim. A specification-based fitness function for evolutionary testing of object-oriented programs. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1953–1954. ACM Press, 2006.

[56] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[57] E. M. Clarke, R. Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1-2):77–104, 1996.

[58] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, January 2000.

[59] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976.

[60] Myra Cohen, Shiu Beng Kooi, and Witawas Srisa-an. Clustering the heap in multi-threaded applications for improved garbage collection. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1901–1908. ACM Press, 2006.

[61] Myra B. Cohen, Peter B. Gibbons, Warwick B. Mugridge, and Charles J. Colbourn. Constructing test suites for interaction testing. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 38–48. IEEE Computer Society, 2003.

[62] Séverine Colin, Bruno Legeard, and Fabien Peureux. Preamble computation in automated test case generation using constraint logic programming. *Software Testing, Verification and Reliability*, 14(3):213–235, 2004.

[63] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proceedings of the ACM Sigplan 1999 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES'99)*, volume 34.7, pages 1–9. ACM Press, 1999.

[64] C. Cotta and J. M. Troya. On decision-making in strong hybrid evolutionary algorithms. In A. P. Del Pobil, J. Mira, and M. Ali, editors, *Tasks and Methods in Applied Artificial Intelligence*, volume 1416 of *LNCS*, pages 418–427. Springer-Verlag, Berlin Heidelberg, 1998.

[65] T.G. Crainic and M. Toulouse. *Handbook of Metaheuristics*, chapter Parallel Strategies for Metaheuristics, pages 475–513. Kluwer Academic Publishers, 2003.

[66] N.L. Cramer. A representation for the adaptive generation of simple sequential programs. In J.J. Grefenstette, editor, *Proceedings of the First International Conference on Genetic Algorithms*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, July 24–26 1985.

[67] Maria Klawe Danny Dolev and Michael Rodeh. An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, September 1982.

[68] C. Darwin. *On the Origin of Species by Means of Natural Selection*. Londres, 1859.

[69] L. Davis, editor. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[70] Eugenia Díaz, Raquel Blanco, and Javier Tuya. Applying tabu and scatter search to automated software test case generation. In *Proceedings of the 6th Metaheuristic International Conference*, pages 290–297, Vienna, Austria, August 2005.

[71] Eugenia Díaz, Javier Tuya, and Raquel Blanco. A modular tool for automated coverage in software testing. In *Proceedings of the 11th Annual International Workshop on Software Technology and Engineering Practice*, pages 241–246, Amsterdam, The Netherlands, September 2003.

[72] Concettina Del Grosso, Giuliano Antoniol, Massimiliano Di Penta, Philippe Galinier, and Ettore Merlo. Improving network applications security: a new heuristic to generate stress testing data. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1037–1043. ACM Press, 2005.

[73] Concettina Del Grosso, Giulio Antoniol, Ettore Merlo, and Philippe Galinier. Detecting buffer overflow via automatic test input data generation. *Computers and Operations Research (COR) focused issue on Search Based Software Engineeering.*

[74] Christian Del Rosso. Reducing internal fragmentation in segregated free lists using genetic algorithms. In *WISER '06: Proceedings of the 2006 international workshop on Workshop on interdisciplinary software engineering research*, pages 57–60. ACM Press, 2006.

[75] E. Demeulemeester and W. Herroelen. A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. *Management Science*, 38:1803–1818, 1992.

[76] K. Derderian, R. Hierons, M. Harman, and Q. Guo. Automated Unique Input Output sequence generation for conformance testing of FSMs. *The computer Journal*, 49(3):331–344, 2006.

[77] Karnig Agop Derderian. *Automated test sequence generation for Finite State Machines using Genetic Algorithms.* PhD thesis, School of Information Systems, Computing and Mathematics, Brunel University, 2006.

[78] E. Díaz, J. Tuya, and R. Blanco. Automated software testing using a metaheuristic technique based on tabu search. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 310–313, Montreal, Quebec, Canada, October 2003.

[79] Eileen Dillon. Hybrid Approach for the Automatic Determination of Worst Case Execution Time for Embedded Systems Written in C. Master's thesis, Institute of Technology, Carlow, 2005.

[80] Karl Doerner and Walter J. Gutjahr. Representation and optimization of software usage models with non-markovian state transitions. *Information and Software Technology*, 42:873–887, 2000.

[81] Karl Doerner and Walter J. Gutjahr. Extracting test sequences from a markov software usage model by ACO. In *Genetic and Evolutionary Computation, GECCO*, volume 2724 of *LNCS*, pages 2465 – 2476. Springer-Verlag, 2003.

[82] Tadashi Dohi, Yasuhiko Nishio, and Shunji Osaki. Optimal software release scheduling based on artificial neural networks. *Annals of Software Engineering*, 8:167–185, 1999.

[83] J. J. Dolado and L. Fernandez. Genetic programming, neural networks and linear regression in software project estimation. In *International Conference on Software Process Improvement, Research, Education and Training*, pages 157–171. British Computer Society, 1998.

[84] Jose J. Dolado. On the problem of the software cost function. *Information and Software Technology*, 43(1):61–72, 2001.

[85] José Javier Dolado. A validation of the component-based method for software size estimation. *IEEE Trans. Software Eng*, 26(10):1006–1021, 2000.

[86] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Dipartamento di Elettronica, Politecnico di Milano, 1992.

[87] M. Dorigo and T. Stützle. *Handbook of Metaheuristics*, volume 57 of *International Series In Operations Research and Management Science*, chapter 9.-The Ant Colony Optimization Metaheuristic: Algorithms, Applications, and Advances, pages 251–285. Kluwer Academic Publisher, 2003.

[88] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. The MIT Press, 2004.

[89] Bernabé Dorronsoro. *Diseño e implementación de algoritmos genéticos celulares para problemas complejos*. PhD thesis, University of Málaga, 2007.

[90] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Intl. Conference on Software Tools and Engineering Practice*, 1999.

[91] Grzegorz Dudek. Genetic algorithm with integer representation of unit start-up and shut-down times for the unit commitment problem. *European Transactions on Electrical Power*, 17(5):500–511, 2006.

[92] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed Explicit Model Checking with HSF-SPIN. In *Lecture Notes in Computer Science, 2057*, pages 57–79. Springer, 2001.

[93] Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Protocol Verification with Heuristic Search. In *AAAI-Spring Symposium on Model-based Validation Intelligence*, pages 75–83, 2001.

[94] Stefan Edelkamp, Stefan Leue, and Alberto Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal of Software Tools for Technology Transfer*, 5:247–267, 2004.

[95] Maria Cláudia Figueiredo Pereira Emer and Silvia Regina Vergilio. GPTesT: A testing tool based on genetic programming. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1343–1350. Morgan Kaufmann Publishers, 2002.

[96] Deji Fatiregun, Mark Harman, and Rob Hierons. Search-based amorphous slicing. In 12^{th} *International Working Conference on Reverse Engineering (WCRE 05)*, pages 3–12, 2005.

[97] Deji Fatiregun, Mark Harman, and Robert Hierons. Search based transformations. In *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724, pages 2511–2512. Springer-Verlag, 2003.

[98] Deji Fatiregun, Mark Harman, and Robert Mark Hierons. Evolving transformation sequences using genetic algorithms. In *Fourth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 65–74. IEEE Press, 2004.

[99] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optmization*, 6:109–133, 1999.

[100] L.J. Fogel. Autonomous automata. *Industrial Research*, 4:14–19, 1962.

[101] A.S. Fraser. Simulation of genetic systems by automatic digital computers II: Effects of linkage on rates under selection. 10:492–499, 1957.

[102] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[103] Moheb R. Girgis. Automatic test data generation for data flow testing using a genetic algorithm. *J. UCS*, 11(6):898–915, 2005.

[104] F. Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8:156–166, 1977.

[105] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13:533–549, 1986.

[106] F. Glover. A template for Scatter Search and Path Relinking. In J.-K-Hao et al., editor, *Altificial Evolution*, number 1363 in Lecture Notes in Computer Science, pages 13–54. Springer, 1998.

[107] F. Glover and G. Kochenberger. *Handbook of Metaheuristics*. Kluwer, Norwell, MA, 2002.

[108] F. Glover and M. Laguna. *Tabu search*. Kluwer, 1997.

[109] Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proc. of the 9th Conference on Computer Aided Verification, LNCS 1254*, pages 476–479, 1997.

[110] Patrice Godefroid and Sarfraz Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Lecture Notes in Computer Science, 2280*, pages 266–280. Springer, 2002.

[111] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. *International Journal on Software Tools for Technology Transfer*, 6(2):117–127, 2004 2004.

[112] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM Press.

[113] Nicolas Gold, Mark Harman, Zheng Li, and Kiarash Mahdavi. Allowing overlapping boundaries in source code using a search based approach to concept binding. In *International Conference on Software Maintenance (ICSM)*, pages 310–319. IEEE Computer Society, 2006.

[114] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[115] V. S. Gordon and D. Whitley. Serial and Parallel Genetic Algorithms as Function Optimizers. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 177–183. Morgan Kaufmann, 1993.

[116] Des Greer and Günther Ruhe. Software release planning: an evolutionary and iterative approach. *Information & Software Technology*, 46(4):243–253, 2004.

[117] A. Groce and W. Visser. Heuristics for model checking java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(4):260–276, 2004.

[118] Alex Groce and Willem Visser. Model checking java programs using structural heuristics. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 12–21, New York, NY, USA, 2002. ACM Press.

[119] H. G Gross. *Measuring evolutionary testability of real-time software.* PhD thesis, University of Glamorgan, 2000.

[120] H.-G. Groß and N. Mayer. Search-based execution-time verification in object-oriented and component-based real-time system development. In *WORDS*, page 113. IEEE Computer Society, 2003.

[121] Hans-Gerhard Groß. A prediction system for evolutionary testability applied to dynamic execution time analysis. *Information & Software Technology*, 43(14):855–862, 2001.

[122] Hans-Gerhard Groß, Bryan F. Jones, and David E. Eyres. Structural performance measure of evolutionary testing applied to worst-case timing of real-time systems. *IEE Proceedings - Software*, 147(2):25–30, 2000.

[123] Hans-Gerhard Groß and Nikolas Mayer. Evolutionary testing in component-based real-time system construction. In *GECCO Late Breaking Papers*, pages 207–214. AAAI, 2002.

[124] Qiang Guo. *Improving Fault Coverage and Minimising the Cost of Fault Identification When Testing from Finite State Machines.* PhD thesis, School of Information Systems, Computing and Mathematics, Brunel University, 2006.

[125] Qiang Guo, Mark Harman, Robert Hierons, and Karnig Derderian. Computing unique input/output sequences using genetic algorithms. In *3rd International Workshop on Formal Approaches to Testing of Software (FATES 2003)*, pages 164–177. LNCS 2931, 2003.

[126] Qiang Guo, Robert Mark Hierons, Mark Harman, and Karnig Derderian. Constructing multiple unique input/output sequences using evolutionary optimisation techniques. *IEE Proceedings — Software*, 152(3):127–140, 2005.

[127] H. R. Lourenço, O. Martin, and T. Stützle. *Handbook of Metaheuristics*, chapter Iterated local search, pages 321–353. Kluwer Academic Publishers, 2002.

[128] Mark Harman, Robert Hierons, and Mark Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358. Morgan Kaufmann Publishers, 2002.

[129] Mark Harman, Lin Hu, Robert Hierons, André Baresel, and Harmen Sthamer. Improving evolutionary testing by flag removal. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1359–1366. Morgan Kaufmann Publishers, 2002.

[130] Mark Harman, Lin Hu, Robert M. Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. Testability transformation. *IEEE Trans. Software Eng*, 30(1):3–16, 2004.

[131] Mark Harman, Lin Hu, Robert Mark Hierons, Chris Fox, Sebastian Danicic, André Baresel, Harmen Sthamer, and Joachim Wegener. Evolutionary testing supported by slicing and transformation. In *IEEE International Conference on Software Maintenance*, page 285, 2002.

[132] Mark Harman and Bryan F. Jones. Search based software engineering. *Information and Software Technology*, 43(14):833–839, December 2001.

[133] Mark Harman, Stephen Swift, and Kiarash Mahdavi. An empirical study of the robustness of two module clustering fitness functions. In *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 1, pages 1029–1036. ACM Press, 2005.

[134] Mark Harman and Laurence Tratt. Pareto optimal search based refactoring at the design level. In *GECCO 2007: Proceedings of the Genetic and Evolutionary Computation Conference*, 2007.

[135] Scott Henninger. Case-Based Knowledge Management Tools for Software Development. *Automated Software Engineering*, 4:319–340, 1997.

[136] Francisco Herrera, Manuel Lozano, and Jose L. Verdegay. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial Intelligence Review*, 12(4):265–319, 1998.

[137] Robert M. Hierons, Mark Harman, and Chris Fox. Branch-coverage testability transformation for unstructured programs. *Computer Journal*, 48(4):421–436, 2005.

[138] Robert Mark Hierons, Mark Harman, Qiang Guo, and Karnig Dederian. Input sequence generation for testing of communicating finite state machines (CFSMs). In *Genetic and Evolutionary Computation Conference (GECCO 2004)*, 2004.

[139] K. S. Hindi, H. Yang, and K. Fleszar. An evolutionary algorithm for resource-constrained project scheduling. *IEEE Transactions on Evolutionary Computation*, 6(5):512–518, October 2002.

[140] Charles Anthony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[141] J. H. Holland. *Adaptation in Natural and Artificial Systems.* The University of Michigan Press, Ann Arbor, Michigan, 1975.

[142] J.H. Holland. Outline for a logical theory of adaptive systems. *Journal of the ACM*, 9(3):297–314, 1962.

[143] G. J. Holzmann. An analysis of bitstate hashing. In *Proc. 15th Int. Conference on Protocol Specification, Testing, and Verification*, pages 301–314, Warsaw, Poland, 1995. Chapman & Hall.

[144] G. J. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.

[145] Gerald J. Holzmann. *The SPIN Model Checker.* Addison-Wesley, 2004.

[146] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):1–17, May 1997.

[147] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proc. of the 3th International SPIN Workshop*, 1997.

[148] Gerard J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *Journal International Journal on Software Tools for Technology Transfer*, 2(3):270–278, 1999.

[149] T.S. Hussain. An introduction to evolutionary computation. tutorial presentation. CITO Researcher Retreat, May 12-14, Hamilton, Ontario 1998.

[150] IEEE. IEEE standard glossary of software engineering terminology -description (610.12-1990), 1990.

[151] B. Jones, H.-H., and D. Eyres. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal*, 11(5):299–306, September 1996.

[152] B. F. Jones, D. E. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *Computer Journal*, 41:98–107, 1998.

[153] B.F. Jones, H.-H. Sthamer, and D.E. Eyres. Automatic structural testing using genetic algorithms. *The Software Engineering Journal*, 11:299–306, 1996.

[154] Audun Josang. Security protocol verification using spin. In *International SPIN Workshop*, 1995.

[155] M. Kamel and S. Leue. Vip: a visual editor and compiler for v-promela. In *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1785*, pages 471–486. Springer, 2000.

[156] Moatz Kamel and Stefan Leue. Validation of a remote object invocation and object migration in CORBA GIOP using Promela/Spin. In *International SPIN Workshop*, 1998.

[157] S. Katz and D. Peled. An efficient verification method for parallel and distributed programs. In *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency (LNCS 354)*, pages 489–507. Springer, 1988.

[158] J. Kennedy. Small worlds and mega-minds: effects of neighborhood topology on particle swarm performance. In *Proceedings of IEEE Congress on Evolutionary Computation (CEC 1999)*, pages 1931–1938, Piscataway, NJ, USA, 1999.

[159] T. M. Khoshgoftaar, Y. Liu, and N. Seliya. Module-order modeling using an evolutionary multi-objective optimization approach. In *IEEE METRICS*, pages 159–169. IEEE Press, 2004.

[160] Taghi Khoshgoftaar, Yi Liu, and Naeem Seliya. A Multiobjective Module-Order Model for Software Quality Enhancement. *IEEE Transactions on Evolutionary Computation*, 8(6):593–608, 2004.

[161] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[162] Colin Kirsopp, Martin Shepperd, and John Hart. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1367–1374. Morgan Kaufmann Publishers, 2002.

[163] Bodgan Korel and Ali M. Al-Yami. Assertion-oriented automated test data generation. In *Proceedings of the 18th International Conference on Software Engineering*, pages 71–80, Berlin, Germany, March 1996. IEEE Computer Society Press.

[164] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[165] Bogdan Korel, S. Chung, and P. Apirukvorapinit. Data dependence analysis in automated test generation. In *Proceedings: IASTED International Conference on Software Engineering and Applications*, pages 476–481, 2003.

[166] Bogdan Korel, Mark Harman, S. Chung, P. Apirukvorapinit, R. Gupta, and Q. Zhang. Data dependence based testability transformation in automated test generation. In *ISSRE 05: Proc. of Intl. Symposium on Software Reliability Engineering*, pages 245–254. IEEE Computer Society, 2005.

[167] Alberto Lluch Lafuente. Symmetry Reduction and Heuristic Search for Error Detection in Model Checking. In *Workshop on Model Checking and Artificial Intelligence*, August 2003.

[168] Manuel Laguna and Rafael Martí. *Scatter Search. Metodology and Implementations in C.* Kluwer, Boston, 2003.

[169] Frank Lammermann, André Baresel, and Joachim Wegener. Evaluating evolutionary testability with software-measurements. In *GECCO (2)*, volume 3103, pages 1350–1362. Springer, 2004.

[170] Frank Lammermann and Stefan Wappler. Benefits of software measures for evolutionary white-box testing. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1083–1084. ACM Press, 2005.

[171] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[172] Leslie Lamport. Basic concepts. In *Advanced Course on Distributed Systems - Methods and Tools for Specification*, volume 190 of *Lecture Notes in Computer Science*, 1984.

[173] P. Larrañaga, R. Etxeberria, J. A. Lozano, and J. M. Peña. Optimization by learning and simulation of Bayesian and Gaussian networks. Technical Report KZZA-IK-4-99, Department of Computer Science and Artificial Intelligence, University of the Basque Country, 1999.

[174] Huey-Ming Lee, Shu-Yen Lee, Tsung-Yen Lee, and Jan-Jo Chen. A new algorithm for applying fuzzy set theory to evaluate the rate of aggregative risk in software development. *Information Sciences*, 153:177–197, July 2003.

[175] Martin Lefley and Martin J. Shepperd. Using genetic programming to improve software effort estimation based on general data sets. In *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724, pages 2477–2487. Springer-Verlag, 2003.

[176] G. Leguizamón and Z. Michalewicz. A new version of Ant System for subset problems. In P.J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, editors, *Proceedings of the 1999 Congress on Evolutionary Computation*, pages 1459–1464, Piscataway, New Jersey, USA, 1999. IEEE Computer Society Press.

[177] Per Kristian Lehre and Xin Yao. Runtime analysis of (1+1) EA on computing unique input output sequences. In *Proceedings of 2007 IEEE Congress on Evolutionary Computation (CEC'07)*, 2007.

[178] Zheng Li, Mark Harman, and Robert M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[179] Lung-Chung Liu and Ellis Horowitz. A Formal Model for Software Management. *IEEE Transaction on Software Engineering*, 15(10):1280–1293, October 1989.

[180] Xiyang Liu, Hehui Liu, Bin Wang, Ping Chen, and Xiyao Cai. A unified fitness function calculation rule for flag conditions to improve evolutionary testing. In *ASE '05: Proc. of the 20th IEEE/ACM Intl. Conference on Automated Software Engineering*, pages 337–341. ACM Press, 2005.

[181] Alberto Lluch-Lafuente, Stefan Leue, and Stefan Edelkamp. Partial Order Reduction in Directed Model Checking. In *9th International SPIN Workshop on Model Checking Software*, Grenoble, April 2002. Springer.

[182] Francisco Luna, Antonio J. Nebro, and Enrique Alba. *Parallel Metaheuristics. A New Class of Algorithms*, chapter Parallel Heterogeneous Metaheuristics, pages 395–422. Wiley, 2005.

[183] Gabriel Luque. *Resolución de Problemas Combinatorios con Aplicación Real en Sistemas Distribuidos.* PhD thesis, University of Málaga, 2006.

[184] W. C. Lynch. Computer systems: Reliable full-duplex file transmission over half-duplex telephone line. *Communications of the ACM*, 11(6):407–410, 1968.

[185] Kiarash Mahdavi. *A clustering genetic algorithm for software modularisation with multiple hill climbing approach.* PhD thesis, Brunel University West London, 2005.

[186] Kiarash Mahdavi, Mark Harman, and Robert Hierons. Finding building blocks for software clustering. In *GECCO 2003: Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2724, pages 2513–2514. Springer-Verlag, 2003.

[187] Kiarash Mahdavi, Mark Harman, and Robert Mark Hierons. A multiple hill climbing approach to software module clustering. In *ICSM'03: Proceedings of the International Conference on Software Maintenance*, pages 315–324. IEEE Computer Society Press, 2003.

[188] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proceedings of the International Workshop on Program Comprehension.* IEEE Computer Society Press, 1998.

[189] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems.* Springer-Verlag New York, Inc., New York, NY, USA, 1992.

[190] Nashat Mansour and Miran salame. Data generation for path testing. *Software Quality Journal*, 12(2):121–136, June 2004.

[191] T. Mantere. *Automatic software testing by genetic algorithms.* PhD thesis, University of Vaasa, 2003.

[192] Timo Mantere and Jarmo T. Alander. Evolutionary software engineering, a review. *Applied Soft Computing*, 5(3):315–331, March 2005.

[193] Johannes Mayer. Towards effective adaptive random testing for higher-dimensional input domains. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1955–1956. ACM Press, 2006.

[194] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. Technical Report RSTR-018-97-01, RST Corporation, 1998.

[195] Kenneth L. McMillan. *Symbolic Model Checking. An approach to the state explosion problem.* PhD thesis, Carnegie Mellon University, 1992.

[196] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.

[197] Phil McMinn, David Binkley, and Mark Harman. Testability transformation for efficient automated test data search in the presence of nesting. In *UK Software Testing Workshop*, 2005.

[198] Phil McMinn, Mark Harman, David Binkley, and Paolo Tonella. The species per path approach to search-based test data generation. In *International Symposium on Software Testing and Analysis (ISSTA 06)*, pages 13–24, 2006.

[199] Phil McMinn and Mike Holcombe. The state problem for evolutionary testing. In Erik Cantú Paz et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2724, pages 2488–2498, Chicano, Illinois, USA, 2003. Springer-Verlag.

[200] Phil McMinn and Mike Holcombe. Hybridizing evolutionary testing with the chaining approach. In *GECCO (2)*, volume 3103 of *LNCS*, pages 1363–1374. Springer, 2004.

[201] Phil McMinn and Mike Holcombe. Evolutionary testing of state-based programs. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1013–1020. ACM Press, 2005.

[202] D. G. McVitie and L. B. Wilson. The stable marriage problem. *Com.. ACM*, 14(7):486–490, 1971.

[203] G. Mendel. *Versuche über Pflanzen-Hybriden.* Verhandlungen des Naturforschedes Vereines in Brünn 4, 1865.

[204] D. Merkle, M. Middendorf, and H. Schmeck. Ant colony optimization for resource-constrained project scheduling. *IEEE Transactions on Evolutionary Computation*, 6(4):333–346, 2002.

[205] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *Journal of Chemical Physics*, 21:1087–1092, 1953.

[206] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[207] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.

[208] R. De Millo, W. McCracken, R. Martin, and J. Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings, Menlo Park, California, 1987.

[209] A. Mingozzi, V. Maniezzo, S. Ricciardelli, and L. Bianco. An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation. *Management Science*, 44(5):714–729, 1998.

[210] Brian S. Mitchell and Spiros Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1375–1382. Morgan Kaufmann Publishers, 2002.

[211] Brian S. Mitchell, Spiros Mancoridis, and Martin Traverso. Search based reverse engineering. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 431–438. ACM Press, 2002.

[212] Brian S. Mitchell, Spiros Mancoridis, and Martin Traverso. Using interconnection style rules to infer software architecture relations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, 2004.

[213] N. Mladenovic and P. Hansen. Variable neighborhood search. *Com. Oper. Res*, 24:1097–1100, 1997.

[214] Yannick Monnier, Jean-Pierre Beauvais, and Anne-Marie Déplanche. A genetic algorithm for scheduling tasks in a real-time distributed system. In *EUROMICRO*, pages 20708–20714. IEEE Computer Society, 1998.

[215] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *IEEE Real Time Technology and Applications Symposium*, volume 21, pages 241 – 268, 2001.

[216] H. Mühlenbein. The equation for response to selection and its use for prediction. *Evolutionary Computation*, 5:303–346, 1998.

[217] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York, 1979.

[218] Tadashi Nakatani. Verification of group address registration protocol using promela and spin. In *International SPIN Workshop*, 1997.

[219] Roger M. Needham and Michael D. Schroeder. ACM president's letter: "wouldn't you rather live in a world where people cared". *Communications of the ACM*, 21(12):991–992, 1978.

[220] Andy Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *High-Performance Computing and Networking, International Conference and Exhibition, HPCN Europe 1998, Amsterdam, The Netherlands, April 21-23, 1998, Proceedings*, volume LNCS 1401, pages 987–989. Springer, 1998.

[221] Jeff Offutt. An integrated automatic test data generation system. *Journal of Systems Integration*, 1(3):391–409, November 1991.

[222] Mark O'Keeffe and Mel O'Cinneide. Search-based software maintenance. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 249–260, 2006.

[223] Mireille Palpant, Christian Artigues, and Philippe Michelon. LSSPER: Solving the Resource-Constrained Project Scheduling Problem with Large Neighbourhood Search. *Annals of Operations Research*, 131:237–257, 2004.

[224] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *The Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.

[225] Doron Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1):39–64, January 1996.

[226] M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. BOA: The Bayesian optimization algorithm. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, volume 1, pages 525–532. Morgan Kaufmann Publishers, San Francisco, CA, 1999. Orlando, FL.

[227] Amir Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60, 1981.

[228] Hartmut Pohlheim and Joachim Wegener. Testing the temporal behavior of real-time software modules using extended evolutionary algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, page 1795. Morgan Kaufmann, 1999.

[229] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C. The Art of Scientific Computing.* Cambridge University Press, 2002. Available online at http://www.library.cornell.edu/nr/bookcpdf.html.

[230] Connie L. Ramsey and Victor R. Basili. An Evaluation of Expert Systems for Software Engineering Management. *IEEE Transactions on Software Engineering*, 15(6):747–759, June 1989.

[231] Real Academia Española. *Diccionario de la lengua española*, 22 edition, 2001.

[232] I. Rechenberg. Cybernetic solution path of an experimental problem. Technical report, Royal Aircraft Establishment, Library translation No. 1122, Farnborough, Hants., UK, 1965.

[233] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Fromman-Holzboog Verlag, Stuttgart, 1973.

[234] C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publishing, Oxford, UK, 1993.

[235] Marek Reformat, Xinwei Chai, and James Miller. Experiments in automatic programming for general purposes. In *ICTAI*, pages 366–373. IEEE Computer Society, 2003.

[236] Marco Ronchetti, Giancarlo Succi, Witold Pedrycz, and Barbara Russo. Early estimation of software size in object-oriented environments a case study in a CMM level 3 software firm. *Information Sciences*, 176(5):475–489, March 2006.

[237] Marc Roper. CAST with GAs (genetic algorithms) - automatic test data generation via. evolutionary computation. In *IEE Colloquium on Computer Aided Software Testing Tools.* IEE, 1996.

[238] Marc Roper. Computer aided software testing using genetic algorithms. In *10th International Software Quality Week*, pages 9T1–1–17. Software Research Institute, 1997.

[239] Winston W. Royce. Managing the development of large software systems. In *Proceedings of the IEEE Western Electronic Show and Convention*, pages 328–338, LA, USA, 1970. IEEE Press.

[240] G. Rudolph. *Evolutionary Computation 1. Basic Algorithms and Operators*, volume 1, chapter 9, Evolution Strategies, pages 81–88. IOP Publishing Lt, 2000.

[241] Conor Ryan. *Automatic re-engineering of software using genetic programming.* Kluwer, 2000.

[242] R. Sagarna. *An Optimization Approach for Software Test Data Generation: Applications of Estimation of Distribution Algorithms and Scatter Search.* PhD thesis, Univ. del País Vasco, 2007.

[243] R. Sagarna and J. A. Lozano. Software metrics mining to predict the performance of estimation of distribution algorithms in test data generation. In *Knowledge-Driven Computing. Knowledge Engineering and Intelligent Computations.* Springer-Verlag, 2007. In press.

[244] R. Sagarna and J.A. Lozano. Variable search space for software testing. In *Proceedings of the International Conference on Neural Networks and Signal Processing*, volume 1, pages 575–578. IEEE Press, December 2003.

[245] Ramón Sagarna and Jose A. Lozano. On the performance of estimation of distribution algorithms applied to software testing. *Applied Artificial Intelligence*, 19(5):457–489, May-June 2005.

[246] Ramón Sagarna and José A. Lozano. Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms. *European Journal of Operational Research*, 169(2):392–412, March 2006. (available online).

[247] A. Schultz, J. Grefenstette, and K. De Jong. Test and evaluation by genetic algorithms. *IEEE Expert*, 8(5):9–14, 1993.

[248] H.-P. Schwefel. *Kybernetische Evolution als Strategie der Experimentellen Forschung in der Strömungstechnik.* PhD thesis, Technical University of Berlin, 1965.

[249] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM Press.

[250] Olaf Seng, Markus Bauer, Matthias Biehl, and Gert Pache. Search-based improvement of subsystem decompositions. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1045–1051. ACM Press, 2005.

[251] Olaf Seng, Johannes Stammel, and David Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1909–1916. ACM Press, 2006.

[252] Lijun Shan and Hong Zhu. Testing software modelling tools using data mutation. In *AST '06: Proc. of the 2006 Intl. workshop on Automation of software test*, pages 43–49. ACM Press, 2006.

[253] Y. Shan, R. I. McKay, C. J. Lokan, and D. L. Essam. Software project effort estimation using genetic programming. In *Proc. of the Intl. Conf. on Communications Circuits and Systems*, 2002.

[254] Alaa F. Sheta. Estimation of the COCOMO model parameters using genetic algorithms for NASA software projects. *Journal of Computer Science*, 2(2):118–123, 2006.

[255] K. K. Shukla. Neuro-genetic prediction of software development effort. *Information and Software Technology*, 42:701–713, 2000.

[256] C. L. Simons and I. C. Parmee. Single and multi-objective genetic operators in object-oriented conceptual software design. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1957–1958. ACM Press, 2006.

[257] Krzysztof Socha and Christian Blum. *Metaheuristic Procedures for Training Neural Networks*, chapter 8, Ant Colony Optimization, pages 153–180. Springer, 2006.

[258] M. Soto, A. Ochoa, S. Acid, and L. M. de Campos. Introducing the polytree aproximation of distribution algorithm. In *Second Symposium on Artificial Intelligence. Adaptive Systems. CIMAF 99*, pages 360–367, 1999. La Habana.

[259] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly. Meta optimization: improving compiler heuristics with machine learning. In *PLDI '03: Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 77–90. ACM Press, 2003.

[260] H. Sthamer, J. Wegener, and A. Baresel. Using evolutionary testing to improve efficiency and quality in software testing. In *Proceedings of the 2nd Asia-Pacific Conference on Software Testing Analysis & Review*, Melbourne, Australia, July 2002.

[261] Harmen-Hinrich Sthamer. *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD thesis, University of Glamorgan, November 1995.

[262] Andrew Sutton, Huzefa Kagdi, Jonathan I. Maletic, and L. Gwenn Volkert. Hybridizing evolutionary algorithms and clustering algorithms to find source-code clones. In *GECCO '05: Proc. of the 2005 conference on Genetic and evolutionary computation*, pages 1079–1080. ACM Press, 2005.

[263] T. Stützle. Local search algorithms for combinatorial problems analysis, algorithms and new applications. Technical report, DISKI Dissertationen zur Künstliken Intelligenz. Sankt Augustin, Germany, 1999.

[264] El-Ghazali Talbi and Hervé Meunier. Hierarchical parallel approach for gsm mobile network design. *Journal of Parallel and Distributed Computing*, 66(2):274–290, 2006.

[265] B. Talbot and J. Patterson. An efficient integer programming algorithm with network cuts for solving resource-constrained scheduling problems. *Management Science*, 24:1163–1174, 1978.

[266] Marouane Tlili, Stefan Wappler, and Harmen Sthamer. Improving evolutionary real-time testing. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1917–1924. ACM Press, 2006.

[267] N. Tracey. *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, University of York, 2000.

[268] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 73–81. ACM Press, 1998.

[269] Nigel Tracey, John Clark, and Keith Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180, 1998.

[270] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering*, pages 285–288, Hawaii, USA, October 1998.

[271] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 30(1):61–79, 2000.

[272] Nigel Tracey, John Clark, John McDermid, and Keith Mander. A search-based automated test-data generation framework for safety-critical systems. *Systems engineering for business process change: new directions*, pages 174–213, 2002.

[273] G. H. Travassos and M. O. Barros. Contributions of In Virtuo and In Silico experiments for the future of empirical studies in software engineering. In *Proc. of the ESEIW 2003 Workshop on Empirical Studies in Software Engineering*, pages 117–130, Roman Castles, Italy, 2003. Fraunhofer IRB Verlag.

[274] H. Turgut Uyar, A. Şima Uyar, and Emre Harmanci. Pairwise sequence comparison for fitness evaluation in evolutionary structural software testing. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1959–1960. ACM Press, 2006.

[275] Peter van Eijk. Verifying relay circuits using state machines. In *Intl.l SPIN Workshop*, 1997.

[276] Frank Vavak and Terence C. Fogarty. Comparison of steady state and generational genetic algorithms for use in nonstationary environments. In *International Conference on Evolutionary Computation*, pages 192–195, 1996.

[277] Rodrigo Vivanco and Nicolino Pizzi. Finding optimal software metrics to classify maintainability using a parallel genetic algorithm. In *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103, pages 1388–1399. Springer-Verlag, 2004.

[278] B.M. Wall. *A Genetic Algorithm for Resource-Constrained Scheduling.* PhD thesis, Massachusetts Institute of Technology, 1996.

[279] Stefan Wappler and Frank Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060. ACM Press, 2005.

[280] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1925–1932. ACM Press, 2006.

[281] Joachim Wegener, André Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology Special Issue on Software Engineering using Metaheuristic Innovative Algorithms*, 43(14):841–854, 2001.

[282] Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, December 2001.

[283] Joachim Wegener and Oliver Bühler. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In *Proceedings of GECCO 2004*, volume 3103, pages 1400–1412. Springer-Verlag Berlin Heidelberg, 2004.

[284] Joachim Wegener and M. Grochtmann. Verifying timing constraints of real-time systems by means of evolutionary testing. *Real-Time Systems*, 15(3):275 – 298, 1998.

[285] Joachim Wegener and F. Mueller. A comparison of static analysis and evolutionary testing for the verification of timing constraints. *Real-Time Systems*, 21(3):241–268, 2001.

[286] Joachim Wegener, Harmen Sthamer, Bryan F. Jones, and David E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality*, 6:127–135, 1997.

[287] Joachim Wegener, Harmen Sthamer, Bryan F. Jones, and David E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6:127–135, 1997.

[288] Weihrauch. *Computability.* Springer-Verlag, 1987.

[289] J. Whittaker. *Graphical models in applied multivariate statistics.* John Wiley & Sons, Inc., 1990.

[290] Kenneth Peter Williams. *Evolutionary Algorithms for Automatic Parallelization.* PhD thesis, University of Reading, UK, 1998.

[291] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing (Application des algorithmes génétiques au test des logiciels). In *5th International Conference on Software Engineering and its Applications*, pages 625–636, 1992.

[292] Man Xiao, Mohamed El-Attarand Marek Reformat, and James Miller. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, 12(2):183–239, 2007.

[293] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *ISSTA'07: Proceedings of the International Symposium on Software Testing and Analysis*, 2007.

[294] Yuan Yuan, Zhongjie Li, and Wei Sun. A graph-search based approach to BPEL4WS test generation. In *ICSEA*, page 14. IEEE Computer Society, 2006.

[295] Yuan Zhan and John A. Clark. Search-based mutation testing for simulink models. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1061–1068. ACM Press, 2005.

[296] Yuan Zhan and John A. Clark. The state problem for test generation in simulink. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1941–1948. ACM Press, 2006.

[297] Yuanyuan Zhang, Mark Harman, and S. Afshin Mansouri. The multi-objective next release problem. In *GECCO 2007: Proc. of the Genetic and Evolutionary Computation Conference*, 2007.

Índice de tablas

Índice de figuras

Índice de términos