Algoritmos Basados en Cúmulos de Partículas Para la Resolución de Problemas Complejos

Autor: José Manuel García Nieto
Directores: Enrique Alba Torres
y
Gabriel Jesús Luque Polo

septiembre de 2006

Índice general

Prólogo

El uso de métodos exactos y heurísticos para la resolución de problemas de optimización es un dominio típico en la Informática. Actualmente, esas tareas de optimización imponen una serie de requisitos que dificultan y en muchos casos impiden la utilización de métodos exactos para encontrar las soluciones óptimas. Estos impedimentos son debidos principalmente a la alta complejidad de estas tareas (generalmente NP-duras) y también al tiempo limitado que existe para resolverlas (aplicaciones de tiempo real). Todo ello ha llevado al desarrollo de las llamadas técnicas metaheurísticas [24], que son capaces de encontrar muy buenas soluciones (e incluso en muchos casos la solución óptima) con un tiempo y consumo de recursos razonables. Entre estas técnicas se encuentran los *algoritmos basados en cúmulos de partículas*, concretamente el método denominado *Particle Swarm Optimization* [31] (PSO en adelante) es una técnica bastante novedosa perteneciente a la familia de los algoritmos Bioinspirados [47] (donde también están incluidos los Algoritmos Evolutivos [6] y las Colonias de Hormigas [16]), que está resolviendo de manera muy efectiva y eficiente un gran conjunto de problemas de optimización de alta complejidad.

PSO fue originalmente desarrollado por al psicólogo-sociólogo Jammes Kennedy y por el ingeniero electrónico Russell Eberhart en 1995 fundamentándose en experimentos con algoritmos que modelaban el "comportamiento en vuelo" observado en algunas especies de pájaros, o el comportamiento de los bancos de peces, incluso las tendencias sociales en el comportamiento humano. La *"metáfora social"* que plantea este tipo de algoritmos se puede resumir de la siguiente forma: los individuos que son parte de una sociedad tienen una opinión influenciada por la creencia global compartida por todos los posibles individuos. Cada individuo, puede modificar su opinión (estado) dependiendo de tres factores: el conocimiento del entorno (su adaptación), los estados en la historia por los que ha pasado el individuo (su memoria) y estados en la historia de los individuos cercanos (memoria del vecindario).

En el algoritmo PSO, cada individuo, llamado partícula, se va moviendo en un espacio multidimensional que representa su espacio social. Cada partícula tiene memoria mediante la que conserva parte de su estado anterior. Cada movimiento de una partícula es la composición de una velocidad inicial alea-

toria y dos valores ponderados aleatoriamente: individual (la tendencia de las partículas de preservar su mejor estado anterior) y social (la tendencia a moverse hacia vecinos con mejor posición). Debido a su planteamiento, este tipo de algoritmo se adapta muy bien a problemas matemáticos de carácter continuo, aunque también se ha aplicado con gran éxito en tareas de naturaleza discreta [43].

En una primera fase de este proyecto, se ha diseñado e implementado el algoritmo PSO (tanto versión continua como discreta), para la resolución de problemas bien conocidos en la literatura. Una parte muy interesante consiste en hacer este tipo de algoritmos de acuerdo a patrones software (esqueletos de código) para mejorar la fiabilidad, reutilización y calidad del algoritmo resultante.

Para el diseño del algoritmo PSO en sus diferentes versiones se ha seguido la arquitectura de la biblioteca MALLBA [4] desarrollada en el proyecto TIC1999-0754-C03 (Spanish CICYT project 1999-2002), con la intención de extenderla añadiendo el algoritmo desarrollado en este proyecto. Del mismo modo, la implementación se realizará en el lenguaje de programación C++ [40] para facilitar la incorporación a esta biblioteca. Además, se han desarrollado distintas versiones del PSO (secuencial, distribuido en LAN y en WAN) siguiendo el modelo de MALLBA.

Como segunda parte de este proyecto, se ha aplicado el conocimiento obtenido del estudio e implementación del algoritmo PSO para su utilización a la hora de resolver aplicaciones de gran complejidad y reconocido impacto internacional. Durante esta fase se han abarcado diversas aplicaciones para mostrar la corrección de la implementación y la capacidad resolutiva de la técnica. Inicialmente, se han tratado problemas tanto del campo de la Bioinformática, con el problema Gene Ordering in Microarray Data (GOMAD) [37], como del campo de las Telecomunicaciones, abordando el problema denominado Location Area Management (LA) [48]. Estos problemas se han demostrado que tienen una gran dificultad (pertenecen a la clase NP-duros) y que tienen un gran interés práctico en sus dominios respectivos.

Asimismo, los resultados obtenidos tras la evaluación de estos problemas con los algoritmos desarrollados tienen una repercusión directa en el proyecto nacional financiado TIC2002-04498-C05-02 (The TRACER project, `http://tracer.lcc.uma.es`) y en el proyecto TIN2005-08818-C04-01 (the OPLINK project, `http://oplink.lcc.uma.es`).

Cabe reseñar que han sido necesarias varias fases iniciales para el estudio preliminar de resultados (sistemas similares) así como realizar una serie de prototipos mejorados progresivamente del algoritmo hasta alcanzar una serie de versiones para la resolución óptima de problemas complejos. Se han estudiado parámetros tales como el número de iteraciones, el resultado óptimo encontrado (fitness), el tiempo de ejecución y los recursos de sistema emplea-

dos. Asimismo se han realizado observaciones sobre la bondad del algoritmo en la resolución de los tipos de problemas comentados y su comparación con otros algoritmos relacionados.

Con todo esto, los **objetivos** principales de este proyecto incluyen diseñar e implementar en C++ el esqueleto del algoritmo PSO siguiendo la arquitectura de MALLBA, lo cual permitirá su incorporación en esta biblioteca para la resolución de problemas de optimización. Además, se resolverán las distintas instancias de cada uno de estos problemas citados con un posterior análisis y obtención de estadísticas y conclusiones. Los objetivos secundarios incluyen estudiar las relaciones de este enfoque con otras técnicas de optimización, así como la difusión de los logros obtenidos en Internet y en informes escritos.

Esta memoria está estructurada en 6 capítulos, 3 apéndices y una bibliografía. En el primer capítulo se realiza una introducción sobre el proyecto y se introduce al lector en el campo de las técnicas de optimización metaheurísticas, para centrarse a continuación en los algoritmos basados en cúmulos de partículas. Los capítulos 2 y 3 describen detalles del diseño e implementación de los esqueletos de la biblioteca MALLBA así como los desarrollados en este proyecto. En los capítulos 4 y 5 se realiza una descripción de los problemas tratados y se exponen una serie de experimentos realizados con los algoritmos implementados sobre estos problemas, además de comentar las conclusiones derivadas de los experimentos. En el último capítulo se expresan las conclusiones del proyecto. Finalmente, en los apéndices, se describe el entorno de trabajo, se detallan otras clases no comentadas de la biblioteca MALLBA, para terminar con un manual de instalación y uso.

A continuación, resumimos brevemente capítulo a capítulo el contenido de esta memoria.

En el Capítulo 1 se introduce al lector en las técnicas metaheurísticas de optimización y se realiza una descripción detallada de los algoritmos basados en cúmulos de partículas. Se describen varias versiones del algoritmo PSO, las diferentes topologías de cúmulos y aspectos avanzados.

En el Capículo 2 se define el concepto de esqueleto de código y se describe la estructura e implementación de la biblioteca MALLBA.

El Capítulo 3 presenta el esqueleto desarrollado para los algoritmos basados en cúmulos de partículas, objetivo de este proyecto. Además, se introducen las diferentes versiones implementadas. Finalizando con la descripción del diseño, la implementación y la configuración del algoritmo en general.

En el Capítulo 4 se presentan los problemas resueltos LA y GOMAD, con la descripción de las instancias utilizadas, el modelo empleado para su representación y detalles de implementación en el esqueleto.

En el Capítulo 5 se describen los experimentos realizados para evaluar el esqueleto con los problemas. Se muestra una serie de estadísticas y resultados

obtenidos y se finaliza con conclusiones extraídas de estos experimentos.

Finalmente, en el Capítulo 6 se exponen las conclusiones realizadas sobre la realización de este proyecto. Se comentan las dificultades enfrentadas durante el desarrollo y las futuras extensiones que realizar.

Capítulo 1

Algoritmos Basados en Cúmulos de Partículas

En este capítulo se realiza en primer lugar una introducción a las técnicas metaheurísticas de optimización, estableciendo una clasificación de las más populares. En segundo lugar, se introduce un tipo concreto de estas técnicas metaheurísticas como son los algoritmos basados en cúmulos de partículas. Por último, se realiza una descripción más detallada de este tipo de algoritmos y se presentan sus principales características y aspectos avanzados.

1.1. Introducción a las Técnicas Metaheurísticas de Optimización

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. Constantemente estamos resolviendo pequeños problemas de optimización, como el camino más corto de ir un lugar a otro, la organización de una agenda, etc. En general éstos son lo suficientemente pequeños y pueden ser resueltos sin recurrir a elementos externos a nuestro cerebro. Pero conforme se hacen más grandes y complejos, el uso de los ordenadores para su resolución es inevitable.

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 1.1. Inicialmente, las técnicas las podemos clasificar en exactas (o enumerativas, exhaustivas, etc.) y técnicas 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-Completos. Esto provoca en muchos casos que el tiempo necesario para la resolución del problema sea inabordable (cientos 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 a lo largo de los últimos 30 años. Estos métodos sacrifican la garantía de encontrar el óptimo a cambio de encontrar una "buena" solución en un tiempo "razonable".

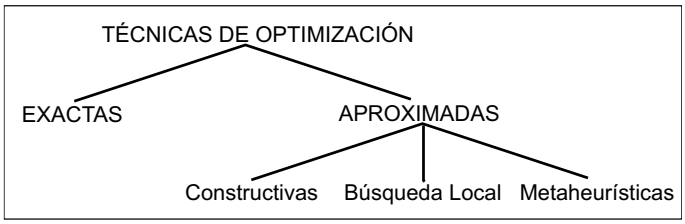

Figura 1.1: Clasificación de las técnicas de optimización

Dentro de los algoritmos no exactos se pueden encontrar tres tipos: los heurísticos constructivos (también llamados voraces), los métodos de búsqueda local (o métodos de seguimiento del gradiente) y las metaheurísticas (en las que nos centramos en esta sección).

Los heurísticos constructivos suelen ser los métodos más rápidos. Generan una solución partiendo de una vacía a la que se les va añadiendo componentes hasta tener 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, y encontrar métodos de esta clase que produzcan buenas soluciones es muy difícil ya que dependen mucho del problema, y para su planteamiento se debe tener un conocimiento muy extenso del mismo. Además, en muchos problemas es casi imposible, ya que, por ejemplo, en aquellos con muchas restricciones puede que la mayoría de las soluciones parciales sólo conduzcan a soluciones no factibles.

Los métodos de búsqueda local o de seguimiento del gradiente parten de una solución ya completa junto con el uso del concepto de vecindario, recorren parte del espacio de búsqueda hasta encontrar un óptimo local. En esa definición han surgido diferentes conceptos, como el de vecindario y óptimo local que ahora pasamos a definir. El vecindario de una solución s, que notamos como $N(s)$, 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 eligen 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 no exactos, 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 en [23] por Glover. Antes de que este término fuese aceptado completamente por la comunidad científica estos métodos eran denominados como heurísticos modernos [45]. Se pueden encontrar revisiones de metaheurísticas en [3, 7, 24].

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 del espacio de búsqueda eficiente para encontrar soluciones (casi) óptimas.
- Las metaheurísticas son algoritmos no exactos y generalmente son no deterministas.
- Pueden incorporar mecanismos para evitar las áreas del espacio de búsqueda no óptimas.
- El esquema básico de cualquier metaheurística es general y no depende del problema a resolver.
- Las metaheurísticas hacen uso de conocimiento del problema que se trata resolver en forma de heurísticos específicos que son controlados de manera estructurada por una estrategia de más alto nivel.
- Las metaheurísticas utilizan funciones de bondad (funciones de *fitness*) para cuantificar el grado de adecuación de una determinada solución.

Resumiendo esos 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 permite abordar problemas con espacios de búsqueda de gran tamaño (por ejemplo, 2^{1000} posibles soluciones). Por lo tanto es de especial interés el correcto equilibrio (generalmente dinámico) que haya entre *diversificación* e *intensificación*.

El termino diversificación se refiere a la exploración del espacio de búsqueda, mientras que intensificación se refiere a la explotación de algún área concreta de ese espacio. El equilibrio entre estos dos aspectos contrapuestos es de gran importancia, ya que por un lado deben identificarse rápidamente la regiones prometedoras del espacio de búsqueda global y por el otro lado no se debe malgastar tiempo en las regiones que ya han sido exploradas o que no contienen soluciones de alta calidad.

Hay diferentes formas de clasificar y describir las técnicas metaheurísticas [13]. Dependiendo de las características que se seleccionen se pueden obtener diferentes taxonomías: basadas en la naturaleza o no basadas en la naturaleza, basadas en memoria o sin memoria, con función objetivo estática o dinámica, etc. En este proyecto hemos elegido clasificarlas de acuerdo a si en cada paso manipulan un único punto del espacio de búsqueda o trabajan sobre un conjunto (población) de ellos, es decir, esta clasificación divide a las metaheurísticas en basadas en trayectoria y basadas en población. Elegimos esta clasificación porque es ampliamente utilizada en la comunidad científica, además de ser muy coherente. Esta taxonomía se muestra de forma gráfica en la Figura 1.2 en la que se pueden observar las principales metaheurísticas que pasamos a describir brevemente en las siguientes subsecciones.

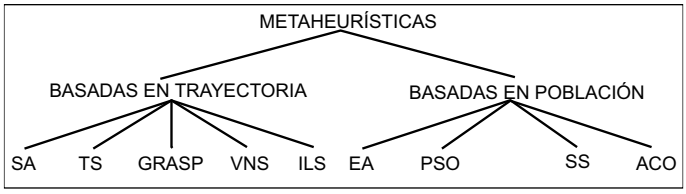

Figura 1.2: Clasificación de las metaheurísticas

1.1.1. Metaheurísticas Basadas en Trayectoria

La principal característica de estos métodos es que parten de un punto y mediante la exploración del vecindario van actualizando la solución actual, formando una trayectoria. La mayoría de estos algoritmos surgen como extensiones de los métodos de búsqueda local simples a los que se les añade alguna característica para escapar de los mínimos locales. Esto implica la necesidad de una condición de parada más compleja 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.

- El Enfriamiento Simulado o *Simulated Annealing* (SA) es una de las más antiguas entre las metaheurísticas y seguramente es el primer algoritmo con una estrategia explícita para escapar de los óptimos locales. El SA fue inicialmente presentado en [35]. La idea del SA es simular el proceso de recocido del metal y del cristal. Para evitar quedar atrapado en un óptimo local, el algoritmo permite elegir una solución peor que 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 variación en la función de fitness, $f(s') - f(s)$ (caso de minimización). Esta probabilidad generalmente se calcula siguiendo la distribución de Boltzmann:

$$p(s'|T, s) = e^{-\frac{f(s')-f(s)}{T}}. \tag{1.1}$$

- 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 [23]. La idea básica de la búsqueda tabú es el uso explícito de un historial de la búsqueda (una memoria de corto plazo), tanto para escapar de los óptimos locales como para implementar su estrategia de exploración y evitar buscar varias veces en la misma región. Esta memoria de corto plazo es implementada en esta técnica como 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.

- El Procedimiento de Búsqueda Miope Aleatorizado y Adaptativo o *The Greedy Randomized Adaptive Search Procedure* (GRASP) [20] es una metaheurística simple que combina heurísticos constructivos con búsqueda local. GRASP es un procedimiento iterativo compuesto de dos fases: primero una construcción de una solución y después un proceso de mejora. La solución mejorada es el resultado del proceso de búsqueda.

- La Búsqueda en Vecindario Variable o *Variable Neighborhood Search* (VNS) es una metaheurística propuesta en [38], que aplica explícitamente una estrategia para cambiar entre diferentes estructuras de vecindario de entre un conjunto de ellas definidas al inicio del algoritmo. Este algoritmo es muy general y con muchos grados de libertad a la hora de diseñar variaciones e instanciaciones particulares.

- La Búsqueda Local Iterada o *Iterated Local Search* (ILS) [51] 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 óptimo local obtenido por el método de mejora puede ser aceptado como nueva solución actual si pasa un test de aceptación.

1.1.2. Metaheurísticas Basadas en Población

Los métodos basados en población se caracterizan por trabajar con un conjunto de soluciones (población) en cada iteración, a diferencia de los métodos que vimos antes que únicamente utilizan un punto del espacio de búsqueda por iteración. El resultado final proporcionado por este tipo de algoritmos depende fuertemente de la forma en que manipula la población.

- Los Algoritmos Evolutivos o *Evolutionary Algorithms* (EA) [6] están inspirados en la capacidad de la naturaleza para evolucionar seres para adaptarlos a los cambios de su entorno. Esta familia de técnicas siguen un proceso iterativo y estocástico que opera sobre una población de individuos. Cada individuo representa una solución potencial al problema que se está resolviendo. Inicialmente, la población es generada aleatoriamente (quizás con ayuda de un heurístico de construcción). Cada individuo en la población tiene asignado, por medio de una función de aptitud (fitness), una medida de su bondad con respecto al problema bajo consideración. Este valor es la información cuantitativa que el algoritmo usa para guiar su búsqueda. En los métodos que siguen el esquema de los algoritmos evolutivos, la modificación de la población se lleva a cabo mediante tres operadores: selección, recombinación y mutación.
  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), basados sobre el hecho que la política de reemplazo permite la aceptación de nuevas soluciones que no mejoran necesariamente la existentes.
  En la literatura se han propuesto diferentes algoritmos basados en este esquema general. Básicamente, estas propuestas se pueden clasificar en tres categorías que fueron desarrolladas de forma independiente. Estas categorías son la Programación Evolutiva o *Evolutionary Programming* (EP) desarrollada por Fogel [21], las Estrategias Evolutivas o *Evolution Strategies* (ES) propuestas por Rechenberg en [44], y los Algoritmos Genéticos o *Genetic Algorithms* (GA) introducidos por Holland en [28].

- La Búsqueda Dispersa o *Scatter Search* (SS) [24] es una metaheurística cuyos principios fueron presentados en [22] y que actualmente está recibiendo una gran atención por parte de la comunidad científica. El algoritmo se basa en mantener un conjunto relativamente pequeño de soluciones tentativas (llamado *"conjunto de referencia"*) que se caracteriza tanto por contener buenas soluciones como soluciones diversas. Este conjunto se divide en subconjuntos de soluciones a las cuales se les aplica una operación de recombinación y mejora. Para realizar la mejora o refinamiento de soluciones se suelen utilizar mecanismos de búsqueda local.

- Los sistemas basados en Colonias de Hormigas o *Ant Colony Optimization* (ACO) [15] son unas metaheurísticas inspiradas en el comportamiento en la de las hormigas reales cuando realizan la búsqueda de 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 la 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. Esta comunicación indirecta entre las hormigas mediante el rastro de feromona las capacita para encontrar el camino más corto entre el nido y la comida. Esta funcionalidad es la que intenta simular este método para resolver problemas de optimización. En esta técnica, el rastro de feromona es simulado mediante un modelo probabilístico.

- Los Algoritmos Basados en Cúmulos de Partículas o *Particle Swarm Optimization* (PSO) [33] son técnicas metaheurísticas inspiradas en el comportamiento social del vuelo de las bandadas de aves o el movimiento de los bancos de peces. Se fundamenta en los factores que influyen en la toma de decisión de un agente que forma parte de un conjunto de agentes similares. La toma de decisión por parte de cada agente se realiza conforme a una componente social y una componente individual, mediante las que se determina el movimiento (dirección) de este agente para alcanzar una nueva posición en el espacio de soluciones. Simulando este modelo de comportamiento se obtiene un método para resolver problemas de optimización.

En esta sección hemos ofrecido una introducción al campo de las metaheurísticas. Hemos incluido una clasificación y un breve repaso por las técnicas más importantes y populares dentro de este campo.

En este proyecto nos centramos en las técnicas comentadas anteriormente de los algoritmos basados en cúmulos de partículas que pasamos a describir con más detalle en la siguiente sección.

1.2. Algoritmos Basados en Cúmulos de Partículas (PSO)

Un "Algoritmo Basado en Cúmulos de Partículas" [1] o *Particle Swarm Optimization* es una técnica metaheurística basada en poblaciones e inspirada en el comportamiento social del vuelo de las bandadas de aves o el movimiento de los bancos de peces. PSO fue originalmente desarrollado por el psicólogo-sociólogo Jammes Kennedy y por el ingeniero electrónico Russell Eberhart en 1995, basándose en un enfoque conocido como la *"metáfora social"* [33], que describe a este algoritmo y que se puede resumir de la siguiente forma: los individuos que conviven en una sociedad tienen una opinión que es parte de un "conjunto de creencias" (el espacio de búsqueda) compartido por todos los posibles individuos. Cada individuo puede modificar su propia opinión basándose en tres factores:

- Su conocimiento sobre el entorno (su valor de fitness).
- Su conocimiento histórico o experiencias anteriores (su memoria).
- El conocimiento histórico o experiencias anteriores de los individuos situados en su vecindario.

Siguiendo ciertas reglas de interacción, los individuos en la población adaptan sus esquemas de creencias al de los individuos con más éxito de su entorno. Con el tiempo, surge una cultura cuyos individuos tienen un conjunto de creencias estrechamente relacionado.

El principio natural en el que se basa PSO es el comportamiento de una bandada de aves o de un banco de peces (Figura 1.3): supongamos que una de estas bandadas busca comida en un área y que solamente hay una pieza de comida en dicha área. Los pájaros no saben dónde está la comida pero sí conocen su distancia a la misma, por lo que la estrategia más eficaz para hallar la comida es seguir al ave que se encuentre más cerca de ella. PSO emula este escenario para resolver problemas de optimización. Cada solución (partícula) es un "ave" en el espacio de búsqueda que está siempre en continuo movimiento y que nunca muere.

El cúmulo de partículas (swarm) es un sistema multiagente, es decir, las partículas son agentes simples que se mueven por el espacio de búsqueda y que guardan (y posiblemente comunican) la mejor solución que han encontrado. Cada partícula tiene un *fitness*, una *posición* y un *vector velocidad* que dirige su "movimiento". El movimiento de las partículas por el espacio está guiado por las partículas óptimas en el momento actual.

[1] En la literatura lo podemos encontrar como población, nube, enjambre o colmena (swarm) de partículas. En este proyecto usaremos el término "cúmulo".

Figura 1.3: Ejemplos de *swarm* en la naturaleza

Los algoritmos basados en cúmulos de partículas se han aplicado con éxito en diferentes campos de investigación. Algunos ejemplos son: optimización de funciones numéricas [54], entrenamiento de redes neuronales [26], aprendizaje de sistemas difusos [42], registrado de imágenes [39], problema del viajante de comercio [46] e ingeniería química [41].

En las siguientes secciones se realiza una descripción formal del algoritmo PSO, junto sus principales factores, parámetros, topologías de vecindario y aspectos avanzados de su desarrollo.

1.3. Descripción del Algoritmo PSO

Un algoritmo PSO consiste en un proceso iterativo y estocástico que opera sobre un cúmulo de partículas. La posición de cada partícula representa una solución potencial al problema que se está resolviendo. Generalmente, una partícula p_i está compuesta de tres vectores y dos valores de fitness:

- El vector $x_i = \langle x_{i1}, x_{i2}, ..., x_{in} \rangle$ almacena la posición actual (localización) de la partícula en el espacio de búsqueda.
- El vector $pBest_i = \langle p_{i1}, p_{i2}, ..., p_{in} \rangle$ almacena la posición de la mejor solución encontrada por la partícula hasta el momento.
- El vector de velocidad $v_i = \langle v_{i1}, v_{i2}, ..., v_{in} \rangle$ almacena el gradiente (dirección) según el cual se moverá la partícula.
- El valor de fitness $fitness\_x_i$ almacena el valor de adecuación de la solución actual (vector x_i).
- El valor de fitness $fitness\_pBest_i$ almacena el valor de adecuación de la mejor solución local encontrada hasta el momento (vector $pBest_i$).

El cúmulo se inicializa generando las posiciones y las velocidades iniciales de las partículas. Las posiciones se pueden generar aleatoriamente en

el espacio de búsqueda (quizás con ayuda de un heurístico de construcción), de forma regular o con una combinación de ambas formas. Una vez generadas las posiciones, se calcula el fitness de cada una y se actualizan los valores de $fitness\_x_i$ y $fitness\_pBest_i$.

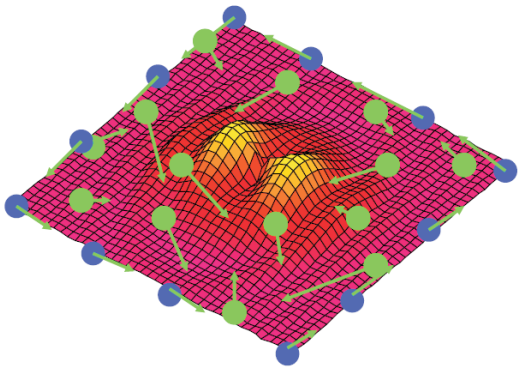

Figura 1.4: Inicialización del cúmulo en el espacio de búsqueda

Las velocidades se generan aleatoriamente, con cada componente en el intervalo $[-v_{max}, v_{max}]$, donde v_{max} será la velocidad máxima que pueda tomar una partícula en cada movimiento. No es conveniente fijarlas a cero pues no se obtienen buenos resultados [33].

Inicializado el cúmulo (Figura 1.4), las partículas se deben mover dentro del proceso iterativo. Una partícula se mueve desde una posición del espacio de búsqueda hasta otra, simplemente, añadiendo al vector posición x_i el vector velocidad v_i para obtener un nuevo vector posición:

$$x_i \leftarrow x_i + v_i \tag{1.2}$$

Una vez calculada la nueva posición de la partícula, se evalúa actualizando $fitness\_x_i$. Además, si el nuevo fitness es el mejor fitness encontrado hasta el momento, se actualizan los valores de mejor posición $pBest_i$ y fitness $fitness\_pBest_i$. El vector velocidad de cada partícula es modificado en cada iteración utilizando la velocidad anterior, un componente *cognitivo* y un componente *social*. El modelo matemático resultante y que representa el *corazón* del algoritmo PSO viene representado por las siguientes ecuaciones:

$$v_i^{k+1} = \omega \cdot v_i^k + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i^k) + \varphi_2 \cdot rand_2 \cdot (g_i - x_i^k) \tag{1.3}$$

$$x_i^{k+1} = x_i^k + v_i^{k+1} \tag{1.4}$$

La Ecuación 1.3 refleja la actualización del vector velocidad de cada partícula i en cada iteración k. El componente *cognitivo* está modelado por el factor $\varphi_1 \cdot rand_1 \cdot (pBest_i - x_i^k)$ y representa la distancia entre la posición actual y la mejor conocida por esa partícula, es decir, la decisión que tomará

la partícula influenciada por su propia experiencia a lo largo de su vida. El componente *social* está modelado por $\varphi_2 \cdot rand_2 \cdot (g_i - x_i^k)$ y representa la distancia entre la posición actual y la mejor posición del vecindario, es decir, la decisión que tomará la partícula según la influencia que el resto del cúmulo ejerce sobre ella. Una descripción más detallada de cada factor se realiza a continuación:

v_i^k : velocidad de la partícula i en la iteración k,

ω : factor *inercia* (descrito en la Sección 1.4),

φ_1, φ_2 : son ratios de aprendizaje (pesos) que controlan los componentes *cognitivo* y *social*,

$rand_1, rand_2$: números aleatorios entre 0 y 1,

x_i^k : posición actual de la partícula i en la iteración k,

$pBest_i$: mejor posición (solución) encontrada por la partícula i hasta el momento,

g_i : representa la posición de la partícula con el mejor $pBest\_fitness$ del entorno de p_i ($lBest$ o *localbest*) o de todo el cúmulo ($gBest$ o *globalbest*).

La Ecuación 1.4 modela el movimiento de cada partícula i en cada iteración k. En la Figura 1.5 se muestra una representación gráfica del movimiento de una partícula en el espacio de soluciones.

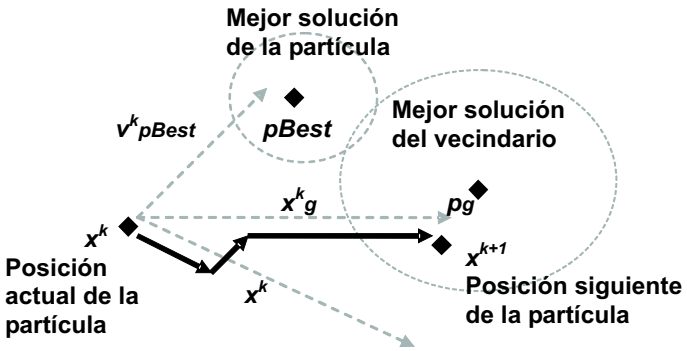

Figura 1.5: Movimiento de una partícula en el espacio de soluciones

En esta gráfica, las flechas de línea discontinua representan la dirección de los vectores de velocidad actual: v_{pBest}^k es la velocidad de la mejor posición tomada por la partícula, v_g^k es la velocidad de la mejor partícula encontrada en el vecindario y v^k es la velocidad actual de la partícula. La flecha de línea continua representa la dirección que toma la partícula para moverse desde la posición x^k hasta la posición x^{k+1}. El cambio de dirección de esta flecha depende de la influencia de las demás direcciones (gradiente) que intervienen en el movimiento.

1.3.1. Tipos de Algoritmos de PSO

Se pueden obtener diferentes tipos de PSO atendiendo a diversos factores de configuración, por ejemplo, según la importancia de los pesos *cognitivo* y *social* y según el tipo de vecindario utilizado.

Por una parte, dependiendo de la influencia de los factores *cognitivo* y *social* (valores φ_1 y φ_2 respectivamente) sobre la dirección de la velocidad que toma una partícula en el movimiento (Ecuación 1.4), Kennedy [30] identifica cuatro tipo de algoritmos:

- ◇ Modelo Completo: $\varphi_1, \varphi_2 > 0$. Tanto el componente *cognitivo* como el *social* intervienen en el movimiento.

- ◇ Modelo sólo *Cognitivo*: $\varphi_1 > 0$ y $\varphi_2 = 0$. Únicamente el componente *cognitivo* interviene en el movimiento.

- ◇ Modelo sólo *Social*: $\varphi_1 = 0$ y $\varphi_2 > 0$. Únicamente el componente *social* interviene en el movimiento.

- ◇ Modelo sólo *Social* exclusivo: $\varphi_1 = 0$, $\varphi_2 > 0$ y $g_i \neq x_i$. La posición de la partícula en sí no puede ser la mejor de su entorno.

Existen estudios realizados en sobre la influencia de los ratios de aprendizaje [33] en los que se recomienda valores de $\varphi_1 = \varphi_2 = 2$.

Por otra parte, desde el punto de vista del vecindario, es decir, la cantidad y posición de las partículas que intervienen en el cálculo de la distancia en la componente *social*, se clasifican dos tipos de algoritmos: PSO Local y PSO Global.

Algoritmo 1 PSO Local

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cumulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    Escoger $lBest_i$, la partícula con mejor fitness del entorno de x_i
    $v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (lBest_i - x_i)$
    $x_i \leftarrow x_i + v_i$
  end for
end while
Salida: la mejor solución encontrada

En el PSO **Local**, se calcula la distancia entre la posición actual de partícula y la posición de la mejor partícula encontrada en el entorno local de la primera. El entorno local consiste en las partículas inmediatamente cercanas en la topología del cúmulo (Subsección 1.3.2). En el Algoritmo 1 se muestra el pseudocódigo de la versión Local de PSO.

Algoritmo 2 PSO Global

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cúmulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
    if $fitness(pBest_i)$ es mejor que $fitness(gBest)$ **then**
      $gBest \leftarrow pBest_i$; $fitness(gBest) \leftarrow fitness(pBest_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    $v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (gBest - x_i)$
    $x_i \leftarrow x_i + v_i$
  end for
end while
Salida: la mejor solución encontrada

Para el PSO **Global**, la distancia en el componente *social* viene dada por la diferencia entre la posición de la partícula actual y la posición de la mejor partícula encontrada en el cúmulo completo $gBest_i$ (ver el pseudocódigo en el Algoritmo 2).

La versión Global converge más rápido pues la visibilidad de cada partícula es mejor y se acercan más a la mejor del cúmulo favoreciendo la intensificación, por esta razón, también cae más fácilmente en óptimos locales. El comportamiento de la versión Local es el contrario, es decir, le cuesta más converger favoreciendo en este caso la diversificación, pero no cae fácilmente en óptimos locales.

1.3.2. Topologías del Cúmulo de Partículas

Un aspecto muy importante a considerar es la manera en la que una partícula interacciona con las demás partículas de su vecindario. El desarrollo de una partícula depende tanto de la topología del cúmulo como de la versión del algoritmo. Las topologías definen el entorno de interacción de una partícula individual con su vecindario. La propia partícula siempre pertenece a su entorno. Los entornos pueden ser de dos tipos:

♢ Geográficos: se calcula la distancia de la partícula actual al resto y se toman las más cercanas para componer su entorno.

♢ Sociales: se define a priori una lista de vecinas para partícula, independientemente de su posición en el espacio.

En la Figura 1.6 se muestran gráficamente ejemplos de entornos geográficos y sociales. Los entornos sociales son los más empleados. Una vez definido un entorno, es necesario definir su tamaño, son habituales valores de 3 y 5 pues suelen tener un buen comportamiento. Obviamente, cuando el tamaño es todo el cúmulo de partículas, el entorno es a la vez geográfico y social, obteniendo así un PSO Global.

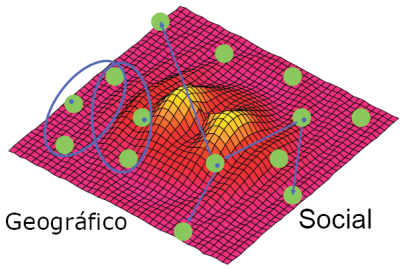

Figura 1.6: Entornos social y geográfico en el espacio de soluciones

Es posible configurar varios tipos de topologías dentro de un entorno social de cúmulo. Una de las más comunes y que inicialmente fue más utilizada es la sociometría *gbest* [31]. En ésta, se establece un vecindario global en el que toda partícula es vecina de la totalidad del cúmulo (PSO Global) favoreciendo la explotación de espacio de soluciones. Otro ejemplo es la sociometría *lbest* (Figura 1.7), que fue propuesta para tratar con problemas de mayor dificultad. En *lbest* (o *anillo*), cada partícula es conectada con sus vecinas inmediatas en el cúmulo, así, por ejemplo, la partícula p_i es vecina de la partícula p_{i-1} y de p_{i+1}. Esta topología ofrece la ventaja de establecer subcúmulos que realizan búsquedas en diversas regiones del espacio del problema y de esta forma se favorece la exploración.

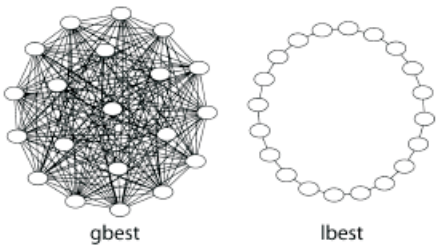

Figura 1.7: Sociometrías de cúmulo *gbest* y *lbest*

Recientes estudios proponen una configuración intermedia llamada topología de *Von Neumann* o *Cuadrada* [34]. En este caso se dispone el cúmulo como una matriz rectangular, por ejemplo de 5×4 para un cúmulo de tamaño 20, donde cada partícula es conectada con las partículas superior, inferior, izquierda y derecha solapando los bordes de manera toroidal (Figura 1.8). Kennedy y Mendes encontraron grandes variaciones entre un pequeño número de configuraciones con diferentes tamaños de la topología de *Von Neumann*, pero no quedó claro cual es la influencia del tipo de topología con respecto a las características de la función. Para algunos problemas distintas topologías eran claramente superiores mientras que con otras se obtenían resultados bastante deficientes.

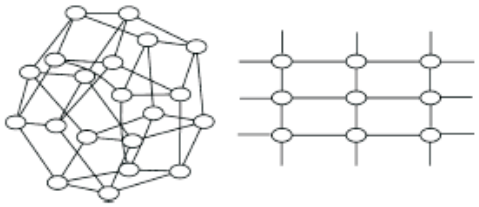

Figura 1.8: Topología de *Von Neumann* o cuadrada

1.4. Aspectos Avanzados de PSO

Un problema habitual de los algoritmos de PSO es que la magnitud de la velocidad suele llegar a ser muy grande durante la ejecución, con lo que las partículas se mueven demasiado rápido por el espacio. El rendimiento puede disminuir si no se fija adecuadamente el valor de v_{max}, la velocidad máxima de cada componente del vector velocidad. En [17] se comparan dos métodos para controlar el excesivo crecimiento de las velocidades: Un factor de *inercia*, ajustado dinámicamente y un *coeficiente de constricción*.

El factor *inercia* ω, es multiplicado por la velocidad actual en la ecuación de actualización de la velocidad (Ecuación 1.3). ω se va reduciendo gradualmente a lo largo del tiempo (medido en iteraciones del algoritmo). Podemos realizar el cálculo de la *inercia* utilizando la Ecuación 1.5 en cada iteración del algoritmo.

$$\omega = \omega_{max} - \frac{\omega_{max} - \omega_{min}}{iter_{max}} \cdot iter \tag{1.5}$$

Donde ω_{max} es el peso inicial y ω_{min} el peso final, $iter_{max}$ es el número máximo de iteraciones y $iter$ es la iteración actual. ω debe mantenerse entre 0.9 y 1.2. Valores altos provocan una búsqueda exhaustiva (más diversificación) y valores bajos una búsqueda más localizada (más intensificación).

Por otra parte, el *coeficiente de constricción* introduce una nueva ecuación para la actualización de la velocidad (Ecuaciones 1.6 y 1.7). Este coeficiente asegura la convergencia [17].

$$v_i^{k+1} = K \cdot [v_i^k + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i^k) + \varphi_2 \cdot rand_2 \cdot (g_i - x_i^k)] \quad (1.6)$$

$$K = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}, \; donde \; \varphi = \varphi_1 + \varphi_2, \varphi > 4 \quad (1.7)$$

Otro aspecto a tener en consideración es el tamaño del cúmulo de partículas, pues determina el equilibro entre la calidad de las soluciones obtenidas y el coste computacional (número de evaluaciones necesarias).

Recientemente, se han propuesto algunas variantes que adaptan heurísticamente el tamaño del cúmulo, de manera que, si la calidad del entorno de la partícula ha mejorado pero la partícula es la peor de su entorno, se elimina la partícula. Por otra parte, si la partícula es la mejor de su entorno pero no hay mejora en el mismo, se crea una nueva partícula a partir de ella. Las decisiones se toman de forma probabilística en función del tamaño actual del cúmulo.

Finalmente, existen trabajos que proponen valores adaptativos para los coeficientes de aprendizaje φ_1 y φ_2. Los pesos que definen la importancia de los componentes *cognitivo* y *social* pueden definirse dinámicamente según la calidad de la propia partícula y del entorno.

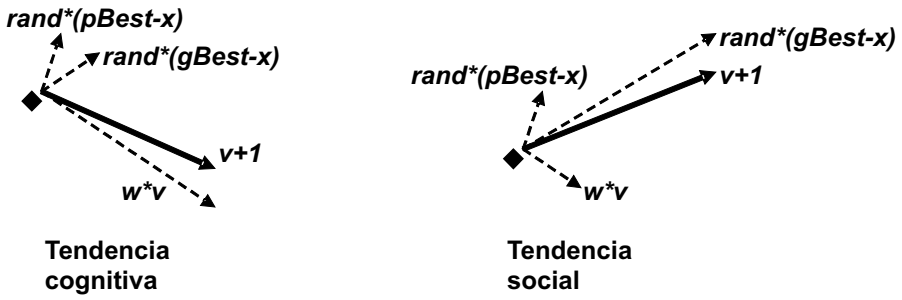

Figura 1.9: Adaptación de coeficientes de aprendizaje

Según la Figura 1.9, cuanto mejor es una partícula, más se tiene en cuenta a sí misma (tendencia *cognitiva*) y no tiene en cuenta factores pasados ni externos. Por el contrario, cuento mejor es el vecino, más tiende a ir hacia él (tendencia *social*).

Capítulo 2

Esqueleto de Código Implementado

En este capítulo presentaremos la plataforma software que hemos utilizado para implementar nuestros algoritmos describiendo brevemente los elementos básicos necesarios para su entendimiento. En primer lugar presentamos una breve noción de lo que se entiende por *esqueleto de código* en el contexto en el que nosotros lo hemos utilizado. En segundo lugar se realiza una descripción de la biblioteca MALLBA para a continuación dar detalles de implementación de los esqueletos. Por último, se distinguen los diferentes tipos de usuario que trabajan sobre un esqueleto de código.

2.1. Introducción

Como definición de *esqueleto de código* podemos decir que se trata de una pauta algorítmica que implementa un método genérico de resolución, que ha de ser debidamente instanciado para resolver un problema concreto. De esta definición se debe resaltar que el esqueleto en sí mismo carece de contenido semántico, es decir, que no puede utilizarse para resolver ningún problema concreto sin antes ser adaptado al problema que se desea resolver.

En nuestro caso, para utilizar cualquiera de los esqueletos del proyecto el usuario sólo debe dar cierta información sobre el problema concreto que desea resolver. Por todo ello los *esqueletos de código* son una herramienta muy importante, ya que permite acercar la tecnología desarrollada tanto para usuarios finales como investigadores que desean hacer uso de ella, sin necesidad de entrar en los detalles propios de dicha tecnología. En este proyecto se ha utilizado la programación orientada a objetos utilizando el lenguaje C++ para la implementación de los esqueletos. Debido a esto, a las características de los *esqueletos de código* se le añaden las propias de la metodología de orientación a objetos, consiguiendo que como resultado tengamos las siguientes ventajas:

- **Reusabilidad:** ya que una misma implementación de un algoritmo nos permite resolver una amplia gama de problemas muy diferentes entre sí.
- **Facilidad de ampliación y de modificación:** como consecuencia de haber utilizado la orientación a objetos, se dispone de varios mecanismos de ampliación como son la herencia, la redefinición de métodos, polimorfismo, ..., que permite ampliar el esqueleto con relativa facilidad. El encapsulamiento en clases de los elementos que forman el esqueleto nos permite modificar uno de ellos sin afectar al resto.
- **Facilidad de aprendizaje y Transparencia de uso:** el esqueleto oculta la implementación del algoritmo, con lo que el usuario final o el experto del problema no necesitan conocerla y sólo necesitan rellenar las clases concretas al problema para utilizarlo.

Los esqueletos incorporados en este proyecto se han diseñado e implementado ajustándonos a las convenciones técnicas impuestas en el proyecto **MALLBA**, en el que colaboran las **Universidades de Málaga** (MA), **La Laguna** (LL) y **la Universidad Politécnica de Cataluña** (BA). Para más información sobre el proyecto **MALLBA** se puede consultar la URL `http://neo.lcc.uma.es/mallba/easy-mallba/index.html`.

2.2. La Biblioteca MALLBA

Debido a que los esqueletos se han desarrollado según las convenciones especificadas en el proyecto **MALLBA**, conviene describir, aunque sea brevemente, esta biblioteca.

Como también se comentó antes todo el código MALLBA está desarrollado en C++, por lo que cada esqueleto de optimización es proporcionado como un conjunto de clases que interactúan entre sí para solucionar el problema de optimización combinatoria instanciado por el usuario. Para diferenciar la parte de esqueleto proporcionada y la que debe ser adaptada al problema concreto todas las clases están etiquetadas con un identificador que lo indica, clases *provided* y clases *required*:

- **Clases Proporcionadas** o clases *provided* son aquellas que se responsabilizan de implementar toda la funcionalidad básica del algoritmo correspondiente. Esta parte es proporcionada por la implementación del algoritmo y no puede ser modificada por el usuario final del mismo.
- **Clases Requeridas** o clases *required* son las responsables de proporcionar al esqueleto todos los aspectos dependientes del problema concreto a resolver. Pese a depender del problema, la interfaz de las mismas es única y prefijada, para permitir a la parte proporcionada utilizarlas sin problemas. Esta parte del esqueleto debe ser rellenada por el usuario o experto del problema de acuerdo con el objetivo fijado.

Aunque los esqueletos de código pueden variar dependiendo del algoritmo que implementen, todos ellos coinciden en un núcleo básico de clases que se pueden ver en el diagrama de clases *UML* [8] de la Figura 2.1.

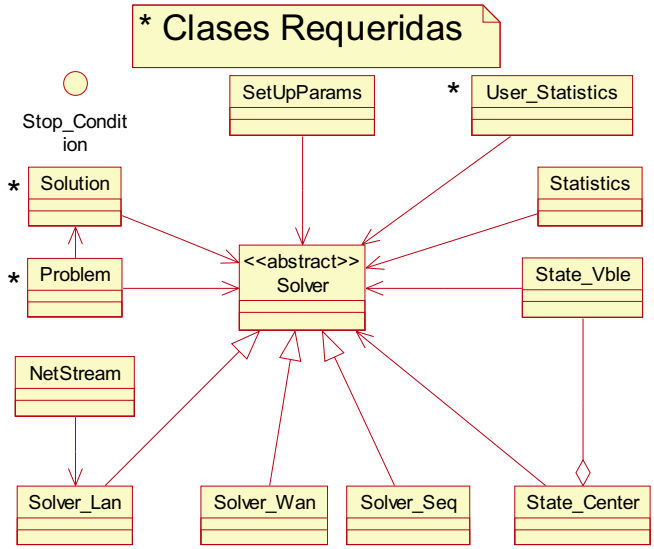

Figura 2.1: Diagrama UML del núcleo básico de los esqueletos MALLBA

Las clases requeridas son `Solution`, `Problem` y `User_Statistics`, siendo el resto clases proporcionadas por el esqueleto. Aparte de esas clases existe también un método común a todos los esqueletos, `terminateQ` (*Terminate Question*), que debe ser proporcionada por el usuario e indica cuándo debe acabar la ejecución del algoritmo.

Clase	Descripción
`SetUpParams`	Configuración del esqueleto.
`Statistics`	Estadísticas sobre el algoritmo.
`State_Center`	Estado del algoritmo.
`State_Vble`	Variable de estado del algoritmo.
`NetStream`	Biblioteca de comunicaciones.
`Stop_Condition`	Condición de Terminación del esqueleto (abstracta).
`Solver`	Ejecución del algoritmo (abstracta).
`Solver_Seq`	Ejecución secuencial del algoritmo.
`Solver_Lan`	Ejecución paralela del algoritmo en una LAN.
`Solver_Wan`	Ejecución paralela del algoritmo en una WAN.
`Problem`	Representa un problema a resolver.
`Solution`	Representa una solución al problema a resolver.
`User_Statistics`	Estadísticas del usuario sobre la resolución del problema.

Tabla 2.1: Descripción de las clases comunes a todos los esqueletos

De cara al usuario final las clases más importantes son las de la jerarquía `Solver`, ya que estas son las clases que encapsulan el motor de optimización del algoritmo (todas las clases se describirán con más detalle en los siguientes apartados o capítulos, aunque puede verse una breve descripción de las clases en la Tabla 2.1).

La clase `Solver_Seq`, se encarga de la ejecución secuencial del patrón de resolución. Además, estos esqueletos ofrecen la posibilidad de ejecución paralela mediante las clases `Solver_Lan` y `Solver_Wan`. El mecanismo de comunicación denominado genéricamente *middleware* se basa en la biblioteca de comunicaciones *NetStream* [2]. Esta biblioteca que ha su vez se basa en MPI, permite a los esqueletos intercambiar sus estructuras de datos de manera eficiente y manteniendo alto nivel de abstracción. En la Figura 2.2 se puede ver representado este mecanismo de comunicación.

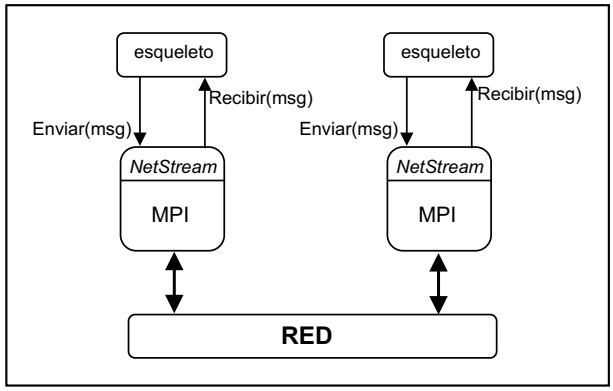

Figura 2.2: Mecanismo de Comunicación

2.3. Implementación de los Esqueletos

La implementación de los esqueletos está divida generalmente en tres ficheros: `<esqueleto>.hh`, `<esqueleto>.req.cc` y `<esqueleto>.pro.cc`:

- **<esqueleto>.hh:** en este fichero se declara la interfaz de todas las clases participantes en el esqueleto. La distinción entre las clases proporcionadas y las requeridas se indica mediante una etiqueta situada antes de la definición de la clase. Dicha etiqueta puede ser **provided**, si la clase es proporcionada por el esqueleto o **required**, si la clase debe ser instanciada por el usuario.

- **<esqueleto>.req.cc:** en este fichero se encuentra la implementación de todas las clases **required** que necesite el esqueleto, así como todas la macros, clases o funciones auxiliares que se necesiten para describir el problema concreto. En esta clase el usuario debe codificar al menos la

implementación de las clases comunes a todos los esqueletos: `Solution`, `Problem` y `User_Statistics`.

- **<esqueleto>.pro.cc:** En este fichero vienen implementadas las clases que proporciona el esqueleto y dependen del algoritmo. Por lo que, al menos, este fichero debe contener la implementación de las clases **provided**: `SetUpParams`, `Statistics`, `Solver`, `Solver_Seq`, ...

Esta división es artificial y realmente no es necesaria seguir esa nomenclatura y reparto, pero usando la nomenclatura propuesta se facilita la utilización del esqueleto, ya que evita la mezcla de implementación de código proporcionado y requerido que puede confundir bastante, ayudando adicionalmente cambiar la parte proporcionada por el esqueleto con mucha facilidad.

Hay clases que aunque son proporcionadas por el esqueleto no vienen en el fichero `<esqueleto>.pro.cc`, como las clases `NetStream`, `State_Vble` y `State_Center`, debido a que estas clases son independientes del algoritmo que se implemente, por lo que se encuentran como ficheros aparte y se encuentran agrupadas dentro de la biblioteca `libmallba.a`.

Otro fichero, que deben poseer todos los esqueletos es el denominado `<esqueleto>.cfg` que incluye la configuración de los parámetros utilizados por los algoritmos (la descripción de todos los parámetros y los valores que pueden tomar se mostrarán conforme se vayan describiendo los esqueletos y problemas concretos).

2.3.1. Clases y Métodos Comunes

En este apartado se describirán las clases y los correspondientes métodos comunes a todos los esqueletos. Comenzaremos con la descripción de las tres las clases requeridas al usuario que son comunes a todos los algoritmos para seguir con las que todos los esqueletos proporcionan, independientemente del algoritmo que implementen.

2.3.2. Clase Requerida `Solution`

Esta clase representa a una posible solución del problema que se está intentando resolver. Los métodos comunes que debe tener esta clase se pueden observar en la Tabla 2.2

De entre todos esos métodos quizás el más importante es `fitness`, debido a que la mayoría de los algoritmos lo utilizan para dirigir la búsqueda.

2.3.3. Clase Requerida `Problem`

Esta clase representa el problema de optimización a resolver. Los métodos que debe incluir cualquier clase `Problem` se muestran en la Tabla 2.3.

Método	Descripción
ostream operator<<(...) *istream operator>>(...)*	Serialización de los objetos `Solution`
NetStream operator<<(...) *NetStream operator>>(...)*	Envío y recepción de una solución a través de la red
Solution& operator=(const Solution& sol)	Asignación
bool operator==(const Solution & sol) const *bool operator!=(const Solution & sol) const*	Operadores de comparación
*char *To_String() const*	Transforma la solución a cadena de caracteres
*void To_Solution(char *_string)*	Transforma la cadena de caracteres en una solución
unsigned int Size() const	Longitud en bytes de la solución
void Initialize()	Genera una solución inicial
double fitness()	Valor de adecuación asociado a la solución

Tabla 2.2: Métodos comunes de la clase `Solution`

Método	Descripción
ostream operator<<(...) *istream operator>>(...)*	Serialización de los objetos `Problem`
Problem& operator=(const Problem & pbm)	Asignación
bool operator==(const Problem & pbm) const *bool operator!=(const Problem & pbm) const*	Operadores de comparación
Direction direction() const	Dirección de optimización del problema (valores `minimize` o `maximize`)

Tabla 2.3: Métodos comunes de la clase `Problem`

De entre todos esos métodos cabe destacar el `operator>>` que nos permite leer los parámetros del problema y el método `direction` que establece si el problema es de maximización o minimización.

2.3.4. Clase Requerida `User_Statistics`

Esta clase almacena las estadísticas que el usuario considere oportunas sobre el funcionamiento del esqueleto. Los métodos comunes de esta clase se muestran en la Tabla 2.4.

El único método que se puede destacar sobre los demás es `Update`, que actualiza las estadísticas a partir del estado contenido en el objeto `Solver`.

Método	Descripción
ostream operator<<(...)	Serialización de los objetos `User_Statistics`
User_Stat & operator=(const User_Stat & us)	Asignación
void Update(const Solver & solver)	Actualiza las estadísticas de usuario a partir del estado del esqueleto
void Clear()	Elimina todas las estadísticas del usuario tomadas hasta el momento

Tabla 2.4: Métodos comunes de la clase `User_Statistics`

2.3.5. Clase Proporcionada `SetUpParams`

Esta clase agrupa todos los parámetros que configuran al patrón de resolución implementado en el esqueleto y además proporciona los métodos para acceder a estos. Los atributos correspondientes a los parámetros comunes a todos lo esqueletos se muestran en la Tabla 2.5.

Atributo	Descripción
_independent_runs	Número de ejecuciones independientes que realizar
_display_state	Indica si se debe mostrar por pantalla el estado durante la resolución del problema
_refresh_global_state	Intervalo de generaciones tras el cual actualizar el estado global del algoritmo
_synchronized	Sincronización entre los procesos en las ejecuciones paralelas. Si su valor es 0 la comunicación es asíncrona, si es 1 es síncrona

Tabla 2.5: Atributos comunes de la clase `SetUpProblem`

2.3.6. Clase Proporcionada `Statistics`

Esta clase recoge información interesante (dependiente del algoritmo implementado en el esqueleto) acerca de la evolución del algoritmo durante la resolución del problema. Los métodos comunes de esta clase se muestran en la Tabla 2.6.

El único método que se puede destacar sobre los demás es el `Update`, que actualiza las estadísticas a partir del estado contenido en el objeto `Solver`.

2.3.7. Clase Proporcionada `Stop_Condition`

Esta clase tiene como objetivo decidir cuándo el algoritmo ha llegado a su terminación. A esta clase sólo se le obliga a tener un método `bool EvaluateCondition(...)`, que indica si el patrón de resolución ha terminado de acuerdo con el estado actual del esqueleto.

Método	Descripción
ostream operator<<(...)	Serialización de los objetos `Statistics`
Stat & operator=(const Stat & stat)	Asignación
void Update(const Solver & solver)	Actualiza las estadísticas a partir del estado del esqueleto
void Clear()	Elimina todas las estadísticas tomadas hasta el momento

Tabla 2.6: Métodos comunes de la clase `Statistics`

Esta clase es abstracta, por lo que para usarla, el experto del sistema debe crear las subclases que crea oportunas con la función de terminación adecuada para el problema que intenta resolver.

2.3.8. Clases Proporcionadas `State_Vble` y `State_Center`

Estas clases son independientes del algoritmo implementado y por ese motivo, como ya se comentó anteriormente, no están implementadas en fichero `<esqueleto>.pro.cc` sino en unos aparte, `State.hh` y `State.cc`.

La clase `State_Vble`, se encarga de almacenar una variable de estado de forma genérica; para ello cuenta con los atributos mostrados en la Tabla 2.7 y los correspondientes métodos de consulta y modificación.

Atributo	Descripción
name	Nombre de la variable de estado
nitems	Número de elementos que constituye esa variable de estado
length	Longitud de cada elemento que constituye la variable de estado
content	Valor almacenado para esta variable de estado (como *char**)

Tabla 2.7: Atributos de la clase `State_Vble`

La clase `State_Center` contienen un conjunto de variables de estado, para ello dispone de los métodos añadir, quitar, consultar, modificar, buscar, ... El objetivo de esta clase es contener el estado global de un esqueleto, con lo que la migración, modificación y consulta del mismo es independiente del esqueleto concreto y fácil.

2.3.9. Clases Proporcionadas Jerarquía `Solver`

Esta jerarquía está formada por la clase abstracta `Solver` y las herederas de ella `Solver_Seq`, `Solver_Lan` y `Solver_Wan`. Estas clases son las más importantes de todo el esqueleto ya que se encargan de implementar en cada uno de los esqueletos el patrón de resolución correspondiente y además mantienen información actualizada del estado en la ejecución del algoritmo (para

ello se relaciona con la clase `State_Center` descrita en la sección anterior). Los métodos comunes a todas esas clases de la jerarquía `Solver` se muestran en la Tabla 2.8.

Método	**Descripción**
void Run()	Ejecución del algoritmo
void Run(unsigned int nb_ evaluation)	Ejecución parcial del algoritmo
void StartUp()	Inicializar el algoritmo antes de lanzar su ejecución
void DoStep()	Acciones a realizar en cada paso de la ejecución del algoritmo

Tabla 2.8: Métodos de la clase `Solver`

Cabe destacar los dos métodos `Run` que permiten al usuario poner en funcionamiento en algoritmo para resolver el problema instanciado. También ofrecen una interfaz única para cualquier tipo de ejecución, ya sea secuencial (con la clase `Solver_Seq`) o paralela (con la clase `Solver_Lan` o la clase `Solver_Wan`). La diferencia entre ambos métodos es que `Run()`, para decidir si ha acabado, sólo consulta la función requerida `terminateQ`. Mientras el método `Void Run(unsigned int nb_evaluation)` aparte de consultar esa función comprueba que no se excedan el número de evaluaciones indicadas en el argumento que se le pasa.

2.4. Usuarios de los Esqueletos de Código

En relación de los esqueletos de código podemos distinguir tres perfiles de usuarios aunque nada impide que los tres perfiles se den en una misma persona o en varias. Los perfiles son:

- **Programador del Esqueleto,** que es el experto en el dominio del algoritmo desarrollado y se encarga de diseñar e implementar el esqueleto a partir del conocimiento del algoritmo. Este perfil es el encargado de 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,** que es el encargado de instanciar el esqueleto con los datos adecuados al 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 debe ser capaz de identificar la necesidad y finalidad de las clases y métodos que debe rellenar para el problema que está resolviendo.

- **Usuario Final,** es el encargado de utilizar el esqueleto de código una vez que ha sido instanciado por el usuario correspondiente al perfil anterior. El usuario correspondiente al perfil que se está comentando 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 a ejecución del algoritmo, permitiéndole hacer un seguimiento, obtener estadísticas, conocer el tiempo consumido hasta la mejor solución, ...

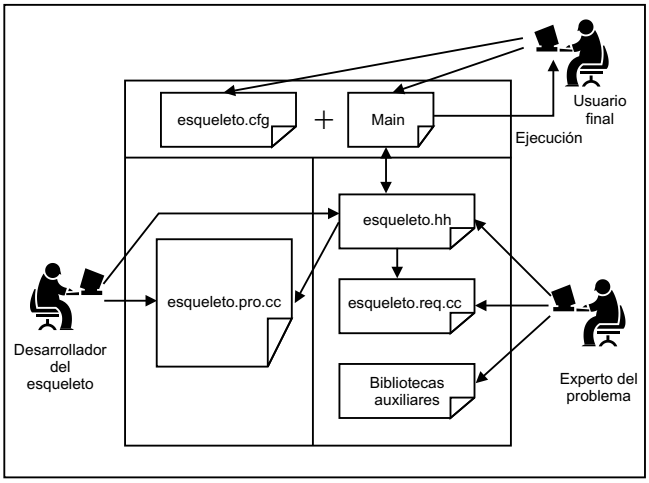

Figura 2.3: Representación de los perfiles que intervienen en un esqueleto

En la Figura 2.3, se muestra gráficamente el papel de cada uno de estos perfiles.

Una vez se han completado las tareas de los tres perfiles, el usuario final dispone de un programa que implementa el motor del algoritmo y resuelve el problema con el que se ha instanciado el esqueleto, usando la configuración que él/ella ha creído oportuna.

En este proyecto, se da el caso, que el autor del mismo adopta los tres perfiles anteriormente indicados, ya que ha desarrollado esqueletos, los ha instanciado con problemas y, por último, ha sintonizado los parámetros del esqueleto, ha ejecutado y tomado estadísticas, resultados, ...

Capítulo 3

Esqueleto para el Algoritmo PSO

En este capítulo se presenta el esqueleto de código para el algoritmo PSO desarrollado en este proyecto. En primer lugar se realiza una introducción al algoritmo PSO y su funcionamiento básico. En segundo lugar, se describen las diferentes versiones implementadas así como los operadores con los que trabajan. Tras esto, se presenta el diseño del esqueleto PSO y su implementación. En las últimas secciones se exponen los diferentes ficheros que componen el esqueleto y la estructura básica del fichero utilizado para su configuración.

3.1. Introducción

Como se ha visto en el Capítulo 1, dentro de las técnicas metaheurísticas y en concreto de los algoritmos bioinspirados se encuentran los algoritmos basados en cúmulos de partículas o Particle Swarm Optimization (PSO). Estos algoritmos poseen las siguientes características [31]:

- En primer lugar, PSO asume un intercambio de información (interacciones sociales) entre los agentes de búsqueda. Es decir, las partículas modifican su dirección en función de las direcciones de las partículas de su vecindario.
- Además, almacena información histórica de la experiencia propia de cada agente (influencia individual). Cada partícula decide su nueva dirección en función de la mejor posición por la que pasó anteriormente.
- Suele tener una convergencia rápida a buenas soluciones.

Una posible definición de PSO podría ser como sigue: Particle Swarm Optimization es una técnica estocástica de búsqueda a ciegas de soluciones cuasi-óptimas. Mantiene un cúmulo (swarm) de partículas cuyas posiciones representan un conjunto de posibles soluciones. Este cúmulo es sometido a ciertas transformaciones con las que se trata de modificar las posiciones de las partículas en un espacio de búsqueda, "moviéndolas" hacia las regiones más prometedoras.

PSO comparte características con otras técnicas bioinspiradas poblacionales como los Algoritmos Evolutivos. Por ejemplo:

- La población es inicializada aleatoriamente y evoluciona iterativamente buscando una solución lo más óptima posible.
- Además, ambas técnicas realizan una búsqueda a ciegas, es decir, no disponen de ningún conocimiento específico del problema, de manera que la búsqueda se basa exclusivamente en los valores de la función objetivo.
- Trabajan con información codificada, no directamente sobre el dominio del problema, sino sobre representación de sus elementos.
- Son técnicas estocásticas, sobre todo referida en fases como las de inicialización y transformación. Ello proporciona control sobre el factor de penetración de la búsqueda.

Sin embargo, difieren en que PSO no tiene operadores de evolución como la mutación o el cruce, sino que tiene operadores de movimiento. Además, no se crean nuevos individuos como en los Algoritmos Evolutivos sino que los mismos individuos (partículas) iniciales son modificados a lo largo del proceso de búsqueda.

Debido a la naturaleza del propio algoritmo y sus operadores, PSO fue inicialmente pensado para la resolución de problemas de optimización que requieren codificación continua (ya que se fundamenta en operaciones sobre vectores gradiente). No obstante, existen variantes del PSO adaptadas para trabajar con codificación binaria y para permutaciones de enteros [11, 33].

3.2. Funcionamiento del PSO Básico

El algoritmo PSO secuencial (Algoritmo 3) procede de manera iterativa modificando un cúmulo de partículas mediante la aplicación de "movimiento" a cada una de ellas.

Este pseudocódigo define el algoritmo PSO básico. Dependiendo de la implementación de las partículas, la representación de las soluciones, y las fases de "actualización de velocidad" y "movimiento", se obtendrá una de las tres variantes principales del PSO (codificación continua, binaria y permutaciones de enteros).

3.3. PSO para Codificación Continua

Esta es la versión más popular del algoritmo PSO y es la que se viene exponiendo a lo largo de esta memoria. Concretamente en el Capítulo 2 se realiza una descripción genérica referida a esta variante.

Algoritmo 3 Pseudocódigo del algoritmo PSO básico

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cúmulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
    if $fitness(pBest_i)$ es mejor que $fitness(g_i)$ **then**
      $g_i \leftarrow pBest_i$; $fitness(g_i) \leftarrow fitness(pBest_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    elegir p_j, la partícula con mejor fitness del vecindario p_i
    actualizar la velocidad de la partícula p_i, de acuerdo con los valores de p_i y p_j
    mover la partícula p_i de acuerdo con su nueva velocidad
  end for
end while

En esta sección, se añaden algunos comentarios más específicos de la codificación continua, sobre todo en lo que respecta a la representación de las partículas y operadores. En Algoritmo 4 se muestra una versión más específica del pseudocódigo del PSO para codificación continua.

Algoritmo 4 Pseudocódigo del algoritmo PSO para codificación continua

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cúmulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
    if $fitness(pBest_i)$ es mejor que $fitness(g_i)$ **then**
      $g_i \leftarrow pBest_i$; $fitness(g_i) \leftarrow fitness(pBest_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    $v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (g_i - x_i)$
    $x_i \leftarrow x_i + v_i$
  end for
end while
Salida: la mejor solución encontrada

Para no confundir el ámbito de actuación de los operadores, es decir, PSO Global o PSO Local, representamos la mejor partícula del vecindario como g_i, pudiendo ser ésta bien $gBest_i$ o bien $lBest_i$ dependiendo de la versión elegida.

3.3.1. Representación de las Partículas

Cada partícula en cualquier versión del PSO está formada como mínimo por tres componentes:

$$p := \{x, pBest, v\} \tag{3.1}$$

En el PSO para codificación continua, x es un vector real que representa un punto (posición) en el espacio de búsqueda, en la mayoría de los casos, una solución. El vector $pBest$ tiene la misma codificación que x pues representa la mejor posición (solución) encontrada por la partícula p hasta el momento.

El vector real v representa la velocidad de la partícula, es decir, la dirección de la partícula o el vector gradiente en el espacio de búsqueda. En cada iteración, la velocidad es actualizada mediante la suma con los factores *cognitivo* y *social*, por lo que la magnitud de la velocidad puede crecer de forma importante durante la ejecución y las partículas se moverían muy rápido por el espacio. Por este motivo, se suele acotar el valor de la velocidad en un rango $[v_{max}, -v_{max}]$, siendo v_{max} la velocidad máxima inicial de cada componente del vector velocidad.

3.3.2. Operador Actualización de Velocidad

La actualización de la velocidad de cada partícula en el PSO para codificación continua viene dada por la siguiente ecuación:

$$v \leftarrow \omega \cdot v + \varphi_1 \cdot rand_1 \cdot (pBest - x) + \varphi_2 \cdot rand_2 \cdot (g - x) \tag{3.2}$$

Puesto que los vectores x, $pBest$, g y v son reales, las operaciones de suma, resta y multiplicación se realizan sin ningún tipo de transformación. No obstante, hay que tener en cuenta que los valores aleatorios obtenidos en $rand_1$ y $rand_2$ deben estar comprendidos en [0,1].

3.3.3. Operador Movimiento

Del mismo modo, el operador movimiento de la partícula se expresa mediante la suma del vector posición x y el vector velocidad v, generando la nueva posición en x donde se situará la partícula en la siguiente iteración. Esta nueva posición representa una nueva solución.

$$x \leftarrow x + v \tag{3.3}$$

3.4. PSO para Codificación Binaria

En la actualidad, se están estudiando una gran cantidad de problemas de optimización que requieren de una representación binaria de sus soluciones. Por este motivo, es necesario proveer una versión del algoritmo PSO que trabaje con este tipo de codificación. En este caso, las posiciones en el espacio de búsqueda se representan mediante cadenas de bits, por lo que el algoritmo y los operadores pueden variar sustancialmente respecto a la versión continua.

Kennedy y Eberhart propusieron en [32] una adaptación inicial del PSO binario. En esta versión, la posición de la partícula es un vector cadena de bits mientras que la velocidad se representa mediante un vector real.

El operador movimiento, al igual que en el caso continuo, depende de la velocidad, aunque en esta variante el significado del concepto de velocidad varía considerablemente: si la velocidad es alta respecto a un valor umbral, el nuevo valor será 1, y si es baja se decidirá 0 (véase la Ecuación 3.6). El valor umbral (representado por ρ) está comprendido en el intervalo $[0, 1]$. Debido a que el umbral está en ese rango, también debemos normalizar el valor de la velocidad al mismo rango, para ello se suele utilizar la función sigmoidal (Ecuación 3.4), la cual es habitualmente utilizada en redes neuronales.

$$sig(v_i^k) = \frac{1}{1 + exp(-v_i^k)} \tag{3.4}$$

De esta forma, en cada iteración k se obtiene 0 ó 1 hasta completar una cadena de bits correspondiente a cada posición de la partícula. El algoritmo PSO binario queda descrito por las ecuaciones 3.5 y 3.6 de actualización de velocidad y movimiento que se presentan a continuación:

$$v_i^{k+1} \leftarrow \omega \cdot v_i^k + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i^k) + \varphi_2 \cdot rand_2 \cdot (g_i - x_i^k) \tag{3.5}$$

$$\rho_i^{k+1} < sig(v_i^{k+1})\ entonces\ x_i^{k+1} = 1;\ en\ otro\ caso\ x_i^{k+1} = 0 \tag{3.6}$$

donde, ρ_i^{k+1} es el vector de valores umbrales comprendidos en [0,0 1,0]. Esta adaptación, transforma mediante la función sigmoide toda la información sobre la dirección que lleva la partícula en dos únicos niveles de decisión limitados por el valor umbral. Esta pérdida de información hace la versión inicial del PSO binario ineficiente en muchos casos.

Actualmente, se están realizando no pocos estudios con el objetivo de desarrollar una versión binaria del PSO más eficiente que la anteriormente vista. Clerc propuso en [11] una serie de versiones que proporcionaban mejores resultados. En concreto, para este proyecto se ha realizado una implementación basada en una versión del PSO binario a la que Clerc llamó *Derivation 0*. En esta versión (Algoritmo 5), se aplica la trasformación mediante operadores de

aritmética modular, con lo que se consigue una menor perdida de información. A continuación, pasamos a describir la representación de las partículas y los operadores según se especifican en la versión *Derivation 0*.

3.4.1. Representación de las Partículas

La posición de una partícula está representada por una cadena de ceros y unos. En este caso, para obtener otro valor de posición se añade 0, $1\ mod_2$, ó $-1\ mod_2$. La idea es usar velocidades que tengan justo estos tres posibles valores $\{-1, 0, 1\}$, y combinar esas tres tendencias para que el resultado sea siempre 0 ó 1 utilizando la función módulo.

Algoritmo 5 PSO para codificación binaria *Derivation 0*

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cúmulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
    if $fitness(pBest_i)$ es mejor que $fitness(g_i)$ **then**
      $g_i \leftarrow pBest_i$; $fitness(g_i) \leftarrow fitness(pBest_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    $v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (g_i - x_i)$
    $x_i \leftarrow x_i + v_i$
    $x_i \leftarrow (4 + x_i) mod_2$
    $v_i \leftarrow (3 + v_i) mod_3 - 1$
  end for
end while
Salida: la mejor solución encontrada

3.4.2. Operador Actualización de Velocidad

Para la actualización del vector velocidad se añade además la Ecuación 3.8, que está encargada de realizar la transformación modular de los valores de velocidad. Mediante la operación mod_3 transforma los nuevos valores de velocidad v_i a valores del conjunto $\{-1, 0, 1\}$. Esta operación se realiza tras el movimiento de la partícula, ya que la velocidad sin transformación interviene en éste.

$$v_i \leftarrow \omega \cdot v_i + \varphi_1 \cdot rand_1 \cdot (pBest_i - x_i) + \varphi_2 \cdot rand_2 \cdot (g_i - x_i) \tag{3.7}$$

$$v_i \leftarrow (3 + v_i) mod_3 - 1 \tag{3.8}$$

Los coeficientes φ_1 y φ_2 son obtenidos según una distribución uniforme en el rango $[-1, 1]$.

3.4.3. Operador Movimiento

De manera similar a la actualización de la velocidad, tras realizar la operación de movimiento se lleva a cabo una trasformación modular de los valores de la nueva posición de la partícula.

$$x_i \leftarrow x_i + v_i \tag{3.9}$$

$$x_i \leftarrow (4 + x_i) mod_2 \tag{3.10}$$

Mediante el mod_2 de los valores de la nueva posición x_i de la partícula, se realiza la trasformación modular a ceros y unos. La suma de cuatro a la posición x_i se debe a que esta puede tener un valor negativo tras la suma de la velocidad, así, con esta operación se garantizan únicamente valores (cero o uno) positivos.

3.5. PSO para Permutaciones de Enteros

Los problemas basados en permutaciones de enteros son otro gran subconjunto dentro de los problemas de optimización. Sin ir más lejos, para el problema del viajante de comercio (TSP), bien conocido en la literatura, la forma más usual de codificar las soluciones es mediante este tipo de representación. En este proyecto, con la intención de abordar este tipo de problemas, se ha desarrollado una versión del PSO para permutaciones de enteros basada en la especificación realizada por Clerc en [10].

Algoritmo 6 PSO para permutaciones de enteros

$S \leftarrow InicializarCumulo()$
while no se alcance la condición de parada **do**
  for $i = 1$ to $size(S)$ **do**
    evaluar cada partícula x_i del cúmulo S
    if $fitness(x_i)$ es mejor que $fitness(pBest_i)$ **then**
      $pBest_i \leftarrow x_i$; $fitness(pBest_i) \leftarrow fitness(x_i)$
    end if
    if $fitness(pBest_i)$ es mejor que $fitness(g_i)$ **then**
      $g_i \leftarrow pBest_i$; $fitness(g_i) \leftarrow fitness(pBest_i)$
    end if
  end for
  for $i = 1$ to $size(S)$ **do**
    $v_i \leftarrow v_i \circ \varphi_1 \otimes (pBest_i \ominus x_i) \circ \varphi_2 \otimes (g_i \ominus x_i)$
    $x_i \leftarrow x_i \oplus v_i$
  end for
end while
Salida: la mejor solución encontrada

Como se puede ver en el pseudocódigo de Algoritmo 6, las ecuaciones de actualización de velocidad y de movimiento utilizan una serie de nuevos operadores para la suma ($\oplus$, $\circ$), la diferencia ($\ominus$) y la multiplicación ($\otimes$) de permutaciones. Antes de describirlos, mostraremos cómo se representan las partículas.

3.5.1. Representación de las Partículas

La posición x de un partícula está formada por una lista de n posibles valores enteros sin que existan repeticiones ni omisiones. Como en los casos anteriores, la posición representa una solución al problema, siendo n la longitud de dicha solución. Un sencillo ejemplo de longitud 6 puede ser:

$$x = (2, 4, 3, 6, 1, 5)$$

La velocidad v está representada por una lista de pares de enteros $(i \rightarrow j)$. Cada uno de estos pares representa un intercambio a realizar sobre los elementos de la posición x. Por ejemplo, si aplicamos el intercambio del par $(4 \rightarrow 1)$ a la posición anterior obtendremos la nueva posición $x = (2, 1, 3, 6, 4, 5)$. Así, el movimiento de una partícula viene dado por la sucesiva aplicación de pares de la lista velocidad a la lista posición. Un ejemplo de lista velocidad es el siguiente:

$$v = [(3 \rightarrow 1), (6 \rightarrow 6), (1 \rightarrow 5)]$$

Aplicando esta lista v a la posición anterior se obtiene: en primer lugar $x' = (2, 4, 1, 6, 3, 5)$ al aplicar $(3 \rightarrow 1)$, y en segundo lugar $x'' = (2, 4, 5, 6, 3, 1)$ al aplicar sobre x' el par $(1 \rightarrow 5)$. El par $(6 \rightarrow 6)$ no realiza ninguna permutación sobre la posición. Por este motivo, se toma como el tamaño de la velocidad $|v|$ el número de pares que realizan intercambios, excluyendo los que no hacen nada, es decir, los pares $(i \rightarrow i)$. Para el ejemplo anterior $|v| = 2$.

3.5.2. Operador Actualización de Velocidad

La Ecuación 3.11 empleada para realizar la actualización de la velocidad tiene el mismo formato que para las versiones continua y binaria del PSO, aunque es necesario describir los operadores de suma, diferencia y multiplicación para permutaciones de enteros.

$$v_i \leftarrow v_i \circ \varphi_1 \otimes (pBest_i \ominus x_i) \circ \varphi_2 \otimes (g_i \ominus x_i) \tag{3.11}$$

Operador Resta de Posiciones ($\ominus$)

Al restar dos posiciones el resultado es una lista velocidad, es decir, una lista de pares. Un par i está formado por el i-ésimo valor del segundo operando

y el i-ésimo valor de primer operando de la resta. Por ejemplo, si restamos las siguientes posiciones:

$$(2,4,3,6,1,5) \ominus (5,1,3,2,4,6),$$

el resultado será la velocidad:

$$[(5 \to 2),(1 \to 4),(3 \to 3),(2 \to 6),(4 \to 1),(6 \to 5)]$$

Operador Suma de Velocidades ($\circ$)

La suma de dos velocidades consiste ir "solapando" los pares de las listas de dichas velocidades, de manera que se obtiene una nueva lista de pares. Para que se produzca un nuevo par, deben coincidir los valores final e inicial de los pares involucrados, es decir, los pares $(i \to j)$ y $(j \to k)$ se solapan formando el par $(i \to k)$. Si dos pares correspondientes no se solapan, el resultado será el primer par. A continuación se muestra un ejemplo completo de suma de velocidades:

$$[(1 \to 1),(2 \to 5),(5 \to 4),(1 \to 3)]$$

$$\circ$$

$$[(1 \to 1),(5 \to 1),(4 \to 3),(2 \to 4)]$$

dando como resultado

$$[(1 \to 1),(2 \to 1),(5 \to 3),(1 \to 3)]$$

Operador Producto Coeficiente Velocidad ($\otimes$)

Mediante el producto coeficiente velocidad se realiza la multiplicación de un coeficiente real φ y una lista velocidad. Se dan varios casos:

- si $\varphi \epsilon [0,1]$, se obtiene $\varphi' = rand(0,1)$ para $\begin{cases} \varphi' \leq \varphi \Rightarrow (i \to j) \to (i \to i) \\ \varphi' > \varphi \Rightarrow (i \to j) \to (i \to j) \end{cases}$
- si $\varphi > 1$, tal que $\varphi = k + \varphi'$, con k entero y $\varphi' < 1$, se realiza *velocidad más velocidad* k veces y *coeficiente* φ' *por velocidad* una vez.

Por ejemplo, si multiplicamos:

$$0{,}5 \otimes [(1 \to 3),(2 \to 5),(4 \to 4),(3 \to 2)],$$

para la secuencia de valores aleatorios (φ') 0.2, 0.8, 0.5 y 0.3 se obtiene como resultado:

$$[(1 \to 1),(2 \to 5),(4 \to 4),(3 \to 3)]$$

3.5.3. Operador Movimiento

Al igual que en las versiones anteriores del PSO, el movimiento se modela mediante la suma de la velocidad a la posición de la partícula (Ecuación 3.12).

$$x_i \leftarrow x_i \oplus v_i \tag{3.12}$$

Operador Suma de Posición con Velocidad ($\oplus$)

Mediante este operador, se realiza el proceso de permutar los valores de la posición de una partícula para generar una nueva posición. Para cada par $(i \rightarrow j)$ de la lista de velocidad, se lleva a cabo en el vector posición un intercambio de los elementos i y j. El resultado es la nueva posición a la que se mueve la partícula tras la realización de sucesivos intercambios. Por ejemplo, sean v una lista velocidad y x la posición de la partícula, al sumarlas:

$$x = (3, 5, 2, 6, 4, 1)$$

$$\oplus$$

$$v = [(2 \rightarrow 1), (4 \rightarrow 2), (3 \rightarrow 3)]$$

se obtiene como resultado

$$x = (3, 5, 1, 6, 2, 4)$$

3.6. Diseño del Esqueleto de Código

En el esqueleto de código PSO, se implementa un algoritmo basado en cúmulo de partículas lo suficientemente genérico para permitir todas las opciones de configuración mostradas en las secciones anteriores. Mediante su correspondiente fichero de configuración se puede escoger los parámetros concretos con los que ejecutar el algoritmo.

En la Figura 3.1 se muestra el diagrama de clases UML que representa la estructura del esqueleto PSO. Como se puede observar, conserva la clases comunes de todos los esqueletos MALLBA mostradas en la Figura 2.1. Cada clase implementa nuevos métodos específicos para representar a un PSO.

Dos nuevas clases aparecen en este diagrama: por un lado la clase `Swarm`, que representa el cúmulo del algoritmo, y que está formada por objetos de la clase `Solution`, que representa a una partícula. Por otro lado tenemos la clase `Migration`, que proporciona métodos para el intercambio de soluciones entre diferentes procesos `Solver_LAN/WAN`.

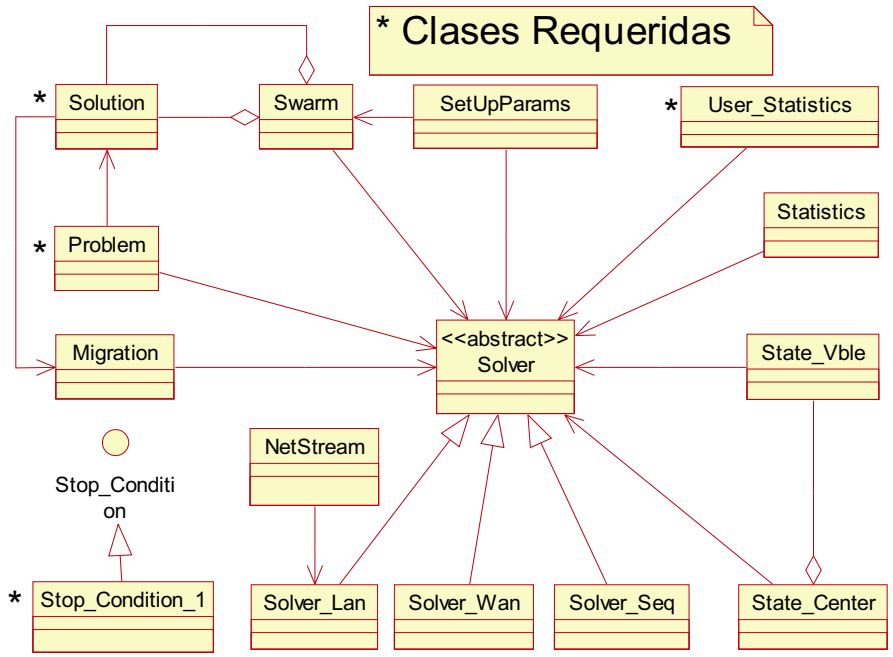

Figura 3.1: Diagrama de clases UML del esqueleto PSO

El mismo diagrama de clases representa las diferentes versiones de PSO (continua, binaria y permutaciones de enteros), aunque la implementación es diferente en cada una. En el apartado de implementación del esqueleto se verá más en detalle el conjunto de métodos que ofrece cada clase.

3.7. Implementación del Esqueleto de Código

En las siguientes secciones vamos a describir cómo se ha llevado a cabo la implementación de este esqueleto, tanto la parte requerida (que es la que debe rellenar el usuario o el experto del problema) como la proporcionada por el esqueleto.

3.7.1. Parte Requerida del Esqueleto

La parte requerida de este esqueleto está formada por las clases `Problem`, `Solution` y `User_Statistics`.

La clase `Problem` sirve para representar al problema de optimización que queremos resolver con este patrón de resolución y debe contener al menos los parámetros básicos que definen al problema, así como la capacidad de leer éstos de un fichero donde se indique la instancia del problema (para mayor detalle

de los métodos que contiene esta clase se remite al lector al Apartado 2.3.3).

La clase `Solution` representa a una partícula del cúmulo. No sólo representa una posible solución al problema tratado, sino que además incorpora toda la información específica de una partícula. Esta información comprende: la posición actual (solución), la mejor posición encontrada durante la vida de la partícula y la velocidad.

Quizás uno de los aspectos más importantes de esta clase (y casi de todo el esqueleto) para la óptima resolución del problema, sea el método `fitness`, que nos indica la aptitud de la solución y es uno de los valores principales por el que se guía el algoritmo en la exploración del espacio de búsqueda. Otro aspecto importante que también influye en el guiado del algoritmo es el método `best_fitness`, que proporciona el mejor fitness encontrado por la partícula en toda su historia. En la Tabla 3.1 se muestran los métodos específicos de la clase `Solution` para el algoritmo PSO. Una descripción de los métodos comunes está en la Tabla 2.2 del Apartado 2.3.2.

Método	**Descripción**
double best_fitness()	Devuelve el mejor fitness histórico
void best_fitness(const double)	Actualiza el mejor fitness histórico
<value>current(const int index)	Devuelve el valor i-ésimo de la posición actual
void current(const int index, <value>)	Actualiza el valor i-ésimo de la posición actual
<value>best(const int index)	Devuelve el i-ésimo mejor valor histórico
void best(const int index, <value>)	Actualiza el i-ésimo mejor valor histórico
<value>velocity(const int index)	Devuelve el valor i-ésimo de la velocidad
void velocity(const int index, <value>)	Actualiza el valor i-ésimo de la velocidad

Tabla 3.1: Métodos específicos de la clase `Solution` para PSO

En la clase `User_Statistics` se recogen las estadísticas de interés para el usuario y de la cual se ha dado en el esqueleto una implementación por defecto. En esta implementación se utiliza una lista para recoger los datos, en la que cada nodo de la misma está constituido por una estructura denominada `user_stat` que tiene los campos descritos en la Tabla 3.2. Se inserta en la lista con los datos recogidos al final de cada ejecución independiente del algoritmo.

Campo	**Descripción**
trial	Número de ejecuciones
nb_generation_best_found_trial	Generación donde se encontró la mejor solución
best_cost_trial	Coste de la mejor solución encontrada
worst_cost_trial	Coste de la mejor solución encontrada
time_best_found_trial	Tiempo en encontrar mejor solución
time_spent_trial	Tiempo total transcurrido en la ejecución

Tabla 3.2: Campos de la estructura `user_stat`

3.7.2. Parte Proporcionada del Esqueleto

La parte proporcionada de este esqueleto está compuesta por las siguientes clases: `SetUpParams`, `Statistics`, `Swarm`, `Migration` y la jerarquía de clases `Solver`.

La clase `SetUpParams` contiene los parámetros de configuración del esqueleto como ya se comentó en el Apartado 2.3.5. Adicionalmente a los parámetros que se explicaban en ese apartado, este esqueleto añade algunos específicos para la configuración del PSO. Dichos atributos se pueden ver en la Tabla 3.3.

Atributo	Descripción
_independent_runs	Número de ejecuciones independientes
_nb_evolution_steps	Número de generaciones máximo por ejecución
_swarm_size	Tamaño del cúmulo
_neighborhood_size	Tamaño del vecindario
_delta_min	Cota inferior de velocidad
_delta_max	Cota superior de velocidad
_individuality_weight	Proporción cognitiva
_ind_min_weight *_ind_max_weight*	Intervalo de factor aleatorio cognitivo
_sociality_weight	Proporción social
_soc_min_weight *_soc_max_weight*	Intervalo de factor aleatorio social
_weight_max	Proporción máxima de inercia
_weight_min	Proporción mínima de inercia

Tabla 3.3: Atributos de la clase `SetUpParmas`

La clase `Statistics`, recoge información interesante acerca de la evolución del algoritmo a lo largo del proceso de resolución. En este caso los datos que se recogen se pueden ver en la Tabla 3.4.

Atributo	Descripción
trial	Número de ejecucion
nb_generation	Número de generación en la ejecución
best_cost	Coste de la mejor solución del cúmulo en la generación.
global_best_cost	Coste de la mejor solución encontrada hasta el momento.
average_cost	Coste medio de la soluciones en el cúmulo.
standard_deviation	Desviación típica del coste de las soluciones del cúmulo.

Tabla 3.4: Datos estadísticos recogidos en la clase `Statistics`

En todo PSO se distingue un cúmulo (swarm) que evoluciona con el paso de las generaciones, en el PSO implementado para el esqueleto el cúmulo de partículas está representado por la clase `Swarm`. Esta clase está formada por un conjunto de partículas cuyas posiciones son las soluciones al problema. Se implementa mediante un atributo privado `_swarm` que es un vector de

apuntadores a partículas. La clase `Swarm` implementa los métodos que permiten transformar el cúmulo actual en uno nuevo. Como método principal en este sentido destaca el método público `evolution`, que se encarga de realizar un paso de evolución del cúmulo. El siguiente pseudocódigo (Figura 3.2) representa el método `evolution`.

```
1:  void evolution()
2:  {
3:      evaluate(C) //evaluar todas las partículas del cúmulo C
4:      updateAll(P(i).best_position) //mejor posición individual
5:      update(C.best_position) //mejor posición global
6:      for all particles P(i) of C
7:      {
8:        P'(i)=choose(P(i)) //mejor partícula del entorno de P(i)
9:        P(i).velocity=updateVelocity(P(i),P'(i)) //actualiza velocidad
10:       P(i+1)=move(P(i)) //mueve la partícula
11:     }
12: }
```

Figura 3.2: Pseudocódigo del método `evolution` de la clase `Swarm`

Otros métodos importantes de esta clase con los que quizás se entienda mejor el anterior pseudocódigo pueden observarse en la Tabla 3.5.

Método	**Descripción**
initialize	Genera un cúmulo inicial de partículas
solution	Devuelve la partícula actual
neighbor_with_best_fitness	Devuelve la mejor partícula del vecindario de una dada
exchange	Aplica los operadores para el intercambio de soluciones con otros cúmulos

Tabla 3.5: Otros métodos destacados de la clase `Swarm`

Entre el resto de los miembros de la clase cabe destacar el atributo privado `_fitness_values`, que se trata de un arreglo que ha sido incorporado a la clase para mejorar la eficiencia, ya que permite disponer en todo momento del valor de aptitud (fitness) actual para cada solución en el cúmulo, así como su posición en la misma. Cuando sea necesario realizar operaciones de reordenación o cálculos en alguno de los métodos implementados se trabajará sobre esta estructura, evitando tener que manejar las soluciones contenidas en el cúmulo, ya que ello podría suponer una fuente de ineficiencia.

Por otra parte, cuando se considere un PSO paralelo (LAN/WAN), con una serie de islas evolucionando en diferentes máquinas, será necesario incluir

una serie de operadores que permitan a estas islas interactuar y colaborar en el proceso de exploración del espacio de búsqueda, mediante el intercambio de soluciones que forman las mismas [3]. En el esqueleto de código desarrollado, sólo se ha incorporado un operador de este tipo representado por la clase `Migration`, aunque se puede ampliar el esqueleto con relativa facilidad para incluir nuevos operadores. La clase `Migration` incorpora el método `execute` que implementa las acciones propias para las comunicaciones (síncrona o asíncrona) y cooperación entre las diferentes islas, siguiendo una topología de anillo. Para configurar este operador hay que indicarle varios parámetros, como cada cuántas generaciones se realiza la migración, cuántas y cuáles soluciones migramos, las soluciones del cúmulo que se reemplazarán con las nuevas soluciones que lleguen desde otras islas y si el intercambio debe hacerse de manera síncrona o asíncrona.

Por último, dentro de la parte proporcionada por el esqueleto se encuentra la jerarquía de clases encabezada por la clase abstracta `Solver`. Esa clase, contiene un gran número de variables que conforman el estado del algoritmo, que son implementadas mediante la utilización de la clase `State_Center`. El estado del algoritmo está formado por un gran número de variables que abarcan desde el número de la ejecución independiente actual y el de la generación actual hasta los parámetros de los diversos operadores pasando por las mejores y peores soluciones encontradas. Esta clase también proporciona una serie de métodos necesarios en la evolución del PSO, que serán heredados por las clases que implementan la ejecución secuencial (`Solver_Seq`) y paralela (`Solver_Lan` y `Solver_Wan`) del algoritmo. Estos métodos son `Run` que es el método ejecuta el algoritmo, el método `StartUp` que se encarga de inicializar el algoritmo antes de ser ejecutado y `DoStep` que agrupa las acciones a realizar en cada generación del algoritmo. La clase `Solver_Seq` se encarga de la ejecución secuencial de este algoritmo y el pseudocódigo de su método `Run` puede verse en la Figura 3.3.

```
1:  void Run()
2:  {
3:      trial = 0;
4:      do {
5:        trial++;
6:        StartUp();
7:        iter = 0;
8:        do {
9:          DoStep();
10:       } while (iter <= MAX_ITERS && !TerminateQ())
11:     }while (iter <= MAX_TRIALS && !TerminateQ())
12: }
```

Figura 3.3: Método `Run` de la clase `Solver`

En este pseudocódigo, la variable `trial` almacena el número de ejecución independiente, `iter` almacena el número de generación en la ejecución y las constantes `MAX_TRIALS` y `MAX_ITERS` indican los máximos valores que pueden tomar dichas variables. Esas constantes pueden ser alteradas en el fichero de configuración del algoritmo.

Para finalizar el apartado, las clases `Solver_Lan` y `Solver_Wan` proporcionan una versión paralela del patrón PSO. Aparte de los métodos heredados de `Solver`, disponen de métodos específicos para la ejecución paralela como son `Pid`, que retorna el identificador de cada proceso, `Send_Local_State_to`, que implementa el envío de información acerca del estado local de cada proceso, `Receive_Local_State`, que implementa la recepción de información del estado de otro proceso, `Check_For_Refresh_Global_State`, que implementa las acciones a realizar por el proceso que se encarga de mantener el estado global del algoritmo y `Reset`, que limpia las colas de entrada de los procesos al finalizar cada ejecución independiente.

No todos los procesos que forman el algoritmo, participan en el proceso de exploración, ya que uno de ellos, el que tienen identificador 0 (proceso Master) se centra en realizar únicamente tareas de recolección de información acerca del estado local del resto de los procesos y a partir de ella mantener un estado global del algoritmo paralelizado. Este proceso no consume CPU, debido a que no realiza espera activa y está bloqueado la mayor parte del tiempo, desbloqueándose solamente cuando se detecta la entrada de un mensaje. La comunicación entre los procesos se puede ver en la Figura 3.4.

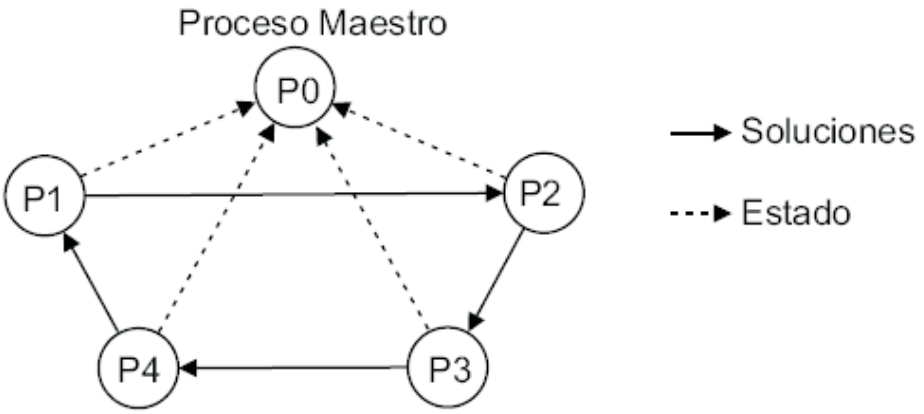

Figura 3.4: Comunicación entre procesos del PSO paralelo

El intercambio de soluciones entre poblaciones se puede realizar de manera síncrona o asíncrona dependiendo de un parámetro configurable por el usuario por medio del fichero `PSO.cfg`. En el caso de comunicaciones síncronas,

cuando un proceso envía soluciones hacia el siguiente en el anillo, este proceso se bloquea en espera de recibir soluciones del anterior proceso en el anillo. En el caso de comunicaciones asíncronas, el proceso tras enviar las soluciones al siguiente proceso del anillo, no espera (no se bloquea) a recibir soluciones del proceso anterior del anillo, sino que continua con la ejecución normal del algoritmo y cada cierto número de generaciones (otro parámetro configurable por el usuario) comprueba si tiene soluciones pendientes de recibir y en caso afirmativo las recoge de su cola de entrada.

3.8. Ficheros que Componen el Esqueleto

El esqueleto desarrollado en este proyecto está formado por una serie de ficheros donde se definen e implementan las clases que constituyen tanto la parte oculta como la visible. Los ficheros que componen este esqueleto son:

- ◇ `PSO.hh`: Definición en C++ de todas las clases del esqueleto.
- ◇ `PSO.req.cc`: Cabecera de los métodos de la clases requeridas del esqueleto a rellenar.
- ◇ `PSO.pro.cc`: Implementación de las clases proporcionadas por el esqueleto, que implementan el patrón de resolución del PSO.
- ◇ `PSO.cfg`: Valores para los parámetros del esqueleto cuando se inicie la ejecución del mismo.

3.9. Configuración del Esqueleto

Un aspecto importante a la hora de ejecutar el algoritmo para resolver un problema, es la sintonización del algoritmo mediante la elección de varios parámetros que intervienen en su funcionamiento. Todos estos parámetros se pueden asignar en el fichero `PSO.cfg` cuyo formato lo podemos ver en la Figura 3.5.

Las líneas no numeradas son palabras claves, que deben aparecer con ese formato y orden en el fichero de configuración, el resto son parámetros que deben ser rellenados por el usuario. En la línea 1 debe indicarse el número de ejecuciones independientes que se deben realizar y en la línea 2 cuántas generaciones se realizan en cada prueba. Las líneas 3, 4 y 5 introducen información sobre el tamaño del cúmulo, de las partículas y del vecindario. La linea 6, establece un parámetro para mostrar, o no, en pantalla información de proceso durante la ejecución. Mediante los parámetros 7 y 8 se establecen las cotas superior e inferior de velocidad del PSO. Desde el parámetro número 9 hasta

el 16 se introducen valores de proporciones cognitivas (individuality) y social (sociality) para el algoritmo. La línea 17 establece la frecuencia de migración en el caso del PSO paralelo. Por último, en las líneas 18, 19 y 20 se introducen los parámetros referentes a la ejecución LAN como: refresco de información por pantalla, modo de ejecución síncrono o asíncrono y la frecuencia de chequeo de soluciones foráneas.

```
General
1:  50 // number of independent runs
2:  100 // number of generations
3:  100 // Swarm size
4:  64 // Particle size
5:  8 // Neighborhood size
6:  1 // display state ?
PSO-params
7:  -10.0 // deltaMin
8:  10.0 // deltaMax
Weight-factors
9:  1.5 // individuality weight
10: 0.0 // ind min
11: 1.0 // ind max
12: 2.5 // sociality weight
13: 0.0 // soc min
14: 1.0 // soc max
15: 1.4 // w max
16: 0.4 // w min
Migration-params
17: 8 // Migration freq
LAN-configuration
18: 10 // refresh global state
19: 0 // 0: running in asynchronized mode / 1: running in synchronized mode
20: 1 // interval of generations to check solutions from other swarms
```

Figura 3.5: Ejemplo de fichero de configuración `PSO.cfg`

Capítulo 4

Problemas Abordados

En este capítulo se presentan los problemas estudiados a lo largo del proyecto. En cada sección se describe cada problema, primero se describirá de manera informal, para luego pasar a una descripción formal mediante la explicación del modelo empleado para su representación y la función de evaluación utilizada. Por último, se detalla la implementación de estos problemas para el esqueleto de código PSO.

4.1. Caso de Estudio: Location Area Management (LA)

Uno de los aspectos más estudiados en el campo de las telecomunicaciones y concretamente en la computación móvil (*mobile computing*), es el seguimiento de la posición de los usuarios o *gestión del área de localización*. En este proyecto, para ser fieles con la literatura dedicada a este problema, lo llamaremos en adelante *Location Area Management* (LA) [25, 50, 53].

Con la finalidad de encaminar las llamadas recibidas a los terminales móviles apropiados, una red inalámbrica debe recopilar periódicamente y mantener información sobre la localización de cada terminal móvil. Esto supone un consumo importante de los limitados recursos de dicha red. Además del ancho de banda utilizado para el registro (*location update*) y localización (*paging*) entre terminales móviles y estaciones base, también se consume energía de los dispositivos portátiles. Más aún, las frecuencias de las señales se pueden degradar perjudicando a la calidad de servicio (*Quality of Service–QoS*) debido a la aparición de interferencias. Por otra parte, un error en la localización de un terminal obligaría a realizar una búsqueda adicional cuando se recibiese una nueva llamada lo cual incrementaría el consumo aún más. Por lo tanto, el objetivo de este problema es equilibrar las operaciones de registro y búsqueda para de esta forma minimizar el coste en el seguimiento de la localización de los terminales móviles.

Existen dos estrategias simples para la gestión de la localización: la estrategia de actualizar siempre o *Always Update* (AU) y la estrategia de nunca actualizar o *Never Update* (NU). En AU, cada terminal móvil realiza una actualización de la posición siempre que entra en una nueva celda. Por lo tanto, el consumo de recursos utilizados para la actualización de la localización será alto. En la estrategia NU, no se realiza actualización de la localización. En este caso, cuando se recibe una llamada, se realiza una operación de búsqueda para determinar el usuario implicado. Claramente, en esta situación, la sobrecarga debida a la operación de búsqueda será alta, pero no se utilizarán recursos para la actualización de la localización. Estas dos estrategias representan los dos extremos de las estrategias en la gestión de localización, en las cuales, un coste es minimizado y el otro es maximizado. La mayoría de los sistemas celulares de telecomunicaciones utilizan una combinación de ambas estrategias.

Actualmente, una de las estrategias de gestión de la localización más comúnmente usada en los sistemas existentes es el esquema de *Location Areas* (LA) [25]. En este esquema, una red es particionada en regiones o LAs, estando formada cada región por una o más celdas (Figura 4.1 Izq.). La estrategia NU puede ser entonces usada dentro de cada LA, es decir, no se realizan actualizaciones de la localización cuando los usuarios se mueven entre celdas que están dentro de la misma región. Únicamente se realizarán actualizaciones de la localización cuando un usuario se mueva hacia fuera, pasando a otra LA diferente. Cuando se reciba una llamada, sólo será necesario buscar al usuario en las celdas que estén dentro del LA.

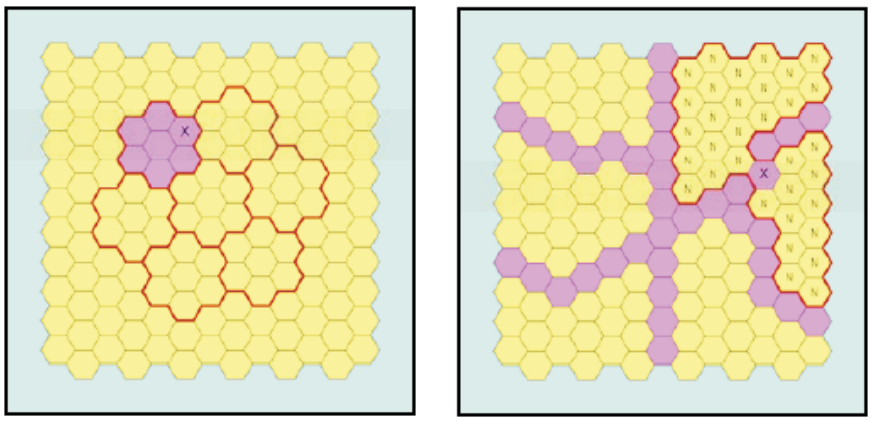

Figura 4.1: Esquemas de configuración: Location Areas (Izq.) y Reporting Cells (Der.)

Por ejemplo, en la Figura 4.1 (Izq.), si el usuario X recibe una llamada, entonces la búsqueda se reduce la las 6 celdas del LA donde se encuentra (celdas oscuras). La partición óptima en LA's para obtener el mínimo coste en la actualización de la localización, se ha reconocido como una problema NP-completo [25].

Otro esquema para la gestión de la localización similar al de LA se propuso en [5]. En esta estrategia, un subconjunto de celdas en una red son designadas como *reporting cells* (RCs–celdas oscuras en la Figura 4.1 Der.). Cada terminal móvil realiza una operación de actualización de la localización sólo cuando entra en una de estas *reporting cells*. Cuando se recibe una llamada, la búsqueda es confinada a la última *reporting cell* que visitó el usuario y su vecindario. Por ejemplo, en la Figura 4.1 (Der.), si el usuario X recibe una llamada, la búsqueda se realiza en la última *reporting cell* que visitó el usuario y en las celdas que pertenecen a su vecindario, marcadas en la figura con N (*neighborhood*). A estas celdas del vecindario se les llama *nonreporting cells* (nRCs). Obviamente, existe una cierta configuración de reporting cells que permite establecer un vecindario de *nonreporting cells* para toda la red no acotado, como muestra la Figura 4.2. Se ha demostrado en [5], que encontrar un conjunto óptimo de *reporting cells* tal que el coste de gestión de la localización sea minimizado, es un problema NP-completo.

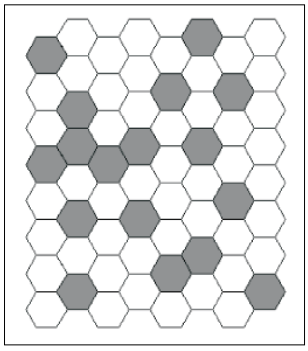

Figura 4.2: Red de RCs y nRCs con un vecindario no acotado

4.1.1. Modelo Empleado

Como se ha mencionado anteriormente, la gestión de la localización requiere elementalmente dos operaciones: actualización de la localización (LU) y la búsqueda de usuarios (*paging*). Evidentemente, una buena estrategia de LU reducirá la sobrecarga en la búsqueda. Al mismo tiempo, no se debería realizar excesivas LUs puesto que consumiría bastantes recursos de la red inalámbrica, siendo estos limitados.

Para determinar el *coste medio* de una estrategia de gestión de la localización, se puede asociar una componente de coste a cada LU realizada, así como a cada búsqueda llevada a cabo por celda (para encontrar al usuario solicitante). La componente de coste generalmente utilizada es el *ancho de banda* inalámbrico consumido (carga de tráfico inalámbrico característico de la red). Es decir, el tráfico inalámbrico desde un terminal móvil a la estación

base (y viceversa) durante la LU y la búsqueda. El tráfico entre dispositivos conectados mediante cableado en la red supone un coste muy bajo, así que no se considera en este problema.

Con todo esto, el coste total de las dos componentes consideradas (actualización de la localización y búsqueda de usuario) durante un período de tiempo T puede ser medido obteniendo el coste medio de la estrategia LA [49]. Así, para calcular el coste total según esta estrategia se puede utilizar la siguiente ecuación:

$$Coste\,Total = C \cdot N_{LU} + N_p, \tag{4.1}$$

donde N_{LU} denota el número de actualizaciones de localización que se ha realizado durante el período T, N_p denota el número de búsquedas de usuario realizadas durante el período T y C es una constante que representa el ratio de coste de LU y búsqueda. En la mayoría de los casos, se asume un valor de $C = 10$ para compensar el hecho de que el coste de LU suele ser varias veces más alto que el de búsqueda.

Una vez definida la estrategia para abordar el problema, se describe a continuación la estructura de red inalámbrica estudiada.

Estructura de Red

La gran mayoría, si no todas, de las redes inalámbricas existentes hoy en día están estructuradas en celdas. Cada celda contiene una estación base conectada a una red cableada. Estas celdas se representan mediante hexágonos, resultando seis posibles vecinos para cada una.

Como se ha mencionado en apartados anteriores, en el esquema de LA mediante reporting cells, algunas celdas de la red son designadas como RCs. Los terminales móviles realizan LUs (actualizan sus posiciones) una vez entran en una de estas RCs.

Cuando se recibe una llamada para un usuario, éste debe ser localizado. Sin embargo, no es necesario que la búsqueda se realize en todas las celdas, sino, únicamente en la última RC visitada y en su vecindario formado por *nonreporting cells*. Para clarificar este factor vecindario (*vicinity*), se asume la configuración RC de la red mostrada en la Figura 4.3, en la cual, las celdas RCs están en color gris oscuro. En la Figura 4.3a y 4.3c, las celdas N pertenecen al vecindario de como mucho tres RCs. El número de vecinos para la celda X es 25, 17 y 22 en la Figura 4.3a, 4.3b y 4.3c respectivamente. El número de vecinos de una RC comprende el número de nRCs del vecindario más 1 (la propia RC). Con todo esto, la celda N pertenece a tres vecindarios de tamaño 25, 17 y 22. Considerando el peor caso, se toma como tamaño de vecindario de una nRC el mayor de los vecindarios al cual pertenece, es decir, 25 para la celda N. Por ejemplo, en la Figura 4.3d, la celda N pertenece a los vecindarios

de todas las RCs X y como factor de vecindad (tamaño de vecindario) se le asigna el mayor de los vecindarios de todas las RCs X.

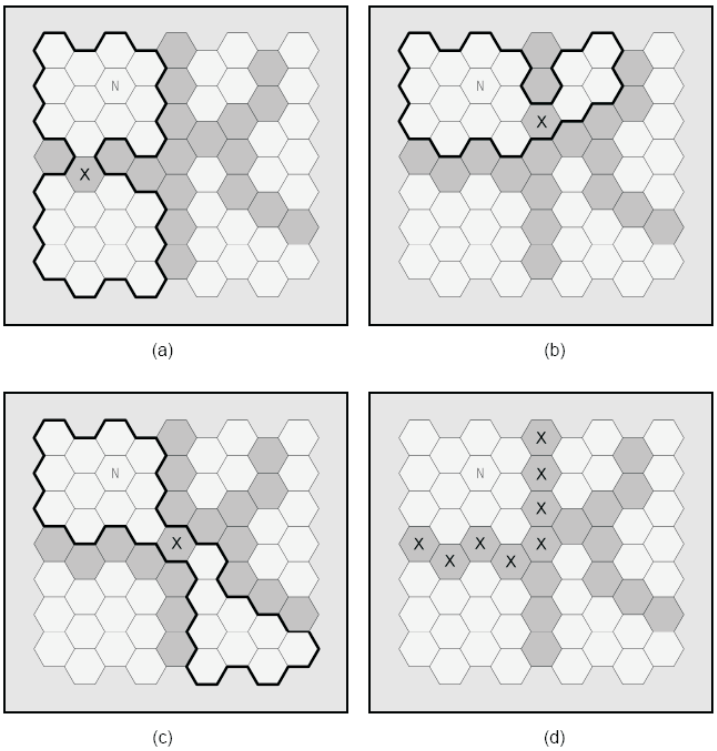

Figura 4.3: Red de RCs y nRCs con diferentes vecindarios: (a), (b) y (c) ilustran el vecindario de una RC X determinada, mientras que (d) ilustra las RCs X a cuyo vecindario pertenece la nRC N

4.1.2. Función de Evaluación

Para evaluar la bondad de una configuración de red RC, se asocia a cada celda i un peso por *movimiento* de un terminal móvil dentro de ésta y un peso por *llamada recibida* a dicho terminal, denotado como w_{mi} y w_{ci} respectivamente. El peso de movimiento representa la frecuencia o el número total de movimientos en una celda, mientras que el peso por llamada recibida representa la frecuencia o el número total de llamadas recibidas en una celda. Obviamente, si una celda i es RC, el número de LUs ocurridas en esa celda dependerá del peso de movimiento de dicha celda. Además, debido a que la recepción de llamadas produce la operación de búsqueda, el número total de búsquedas (*paging*) realizadas estará directamente relacionado con el peso por llamada recibida en las celdas de la red. De este modo, tenemos las siguientes fórmulas para el número total de LUs y *paging* (N_{LU} y N_p respectivamente) realizados durante un periodo T:

$$N_{LU} = \sum_{i \in S} w_{mi}, \tag{4.2}$$

$$N_p = \sum_{j=0}^{N-1} w_{cj} \cdot v(j), \tag{4.3}$$

donde w_{mi} es el peso de movimiento asociado a la celda i, w_{cj} es el peso por llamadas recibidas asociado a la celda j, $v(j)$ denota el valor de vecindario (*vicinity factor*) de la celda j, N es el número total de celdas de la red y S es el número de RCs de la red.

Uniendo las ecuaciones 4.1, 4.2 y 4.3 se obtiene la ecuación para calcular el coste total de LA para configuración RCs de redes inalámbricas:

$$Coste\ Total = C \cdot \sum_{i \in S} w_{mi} + \sum_{j=0}^{N-1} w_{cj} \cdot v(j) \tag{4.4}$$

Para explicar el procedimiento de cálculo del coste total de LA en una red con el esquema RC, se presenta el ejemplo de red basado en la Figura 4.4. En este caso, el número total de LUs para la configuración de la Figura 4.4a es calculado a continuación:

$$N_{LU} = 453 + 395 + 541 + 492 + 432 + 361 + 409 + 123 = 3206$$

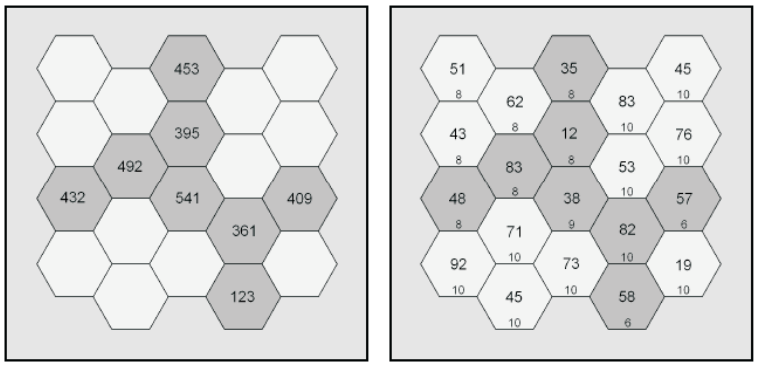

Figura 4.4: Ejemplo del número de LUs (a) y llamadas recibidas (b) en una configuración de RCs

Para calcular es coste de búsqueda (N_p) se debe obtener inicialmente el factor de vecindad de cada celda incluyendo RCs y nRCs. En la Figura 4.4b aparecen dos números en cada celda: en el centro de cada celda se dispone el número de llamadas entrantes y el número en el borde representa el factor de vecindad. En número total de transacciones de búsqueda se calcula como sigue:

$$N_p = 51 \times 8 + 43 \times 8 + 48 \times 8 + 92 \times 10 + 62 \times 8 + 83 \times 8 + 71 \times 10 + 45 \times 10 + 35 \times 8 + 12 \times 8 + 38 \times 9 + 73 \times 10 + 83 \times 10 + 53 \times 10 + 82 \times 10 + 58 \times 6 + 45 \times 10 + 76 \times 10 + 57 \times 6 + 19 \times 10 = 10094$$

Para un valor de la constante $C = 10$, el coste de la red se calcula mediante la ecuación del coste total:

$$Coste\,Total = 3206 \times 10 + 10094 = 42154\,unidades$$

En este proyecto, se ha utilizado la Ecuación 4.4 como función de evaluación en la implementación del PSO para la resolución del problema LA.

4.1.3. Resolución del Problema LA con el Esqueleto PSO

Este problema se ha resuelto con el esqueleto PSO en la versión para codificación binaria (ver Capítulo 3). Siguiendo la filosofía de desarrollo mediante esqueletos de código, la implementación del problema se realizó en las clases requeridas, es decir, la clase `Problem`, la clase `Solution` y la clase `User_statistics`. Para esta última clase se utilizó la implementación por defecto.

Antes de ver las clases requeridas es necesario presentar la estructura de datos `cell`. Esta estructura se utiliza para almacenar de manera ordenada la información necesaria sobre cada celda de una red. La estructura `cell` está compuesta de los siguientes atributos:

- `number`, indica el número de celda en la configuración de red.
- `wc`, especifica el peso por llamada recibida para esta celda.
- `wm`, especifica el peso de movimiento de usuario para esta celda.
- `vicinity`, almacena el valor del factor de vecindad de esta celda.
- `belong`, es un vector de boleanos con un valor para cada celda de la red. Sólo en el caso de que esta celda sea nRC, tendrá valor `true` si pertenece al vecindario de la RC con índice i del vector. El resto estará a `false`.
- `visited`, es un vector de boleanos que especifica las celdas de la red que pertenecen al vecindario de esta celda (siendo ésta RC).

La Clase `Problem`

En primer lugar, se explican los atributos privados específicos de la clase `Problem`:

- `_dimension`, indica la dimensión del problema, es decir, la longitud de la red de celdas.

- `_network`, es un arreglo de celdas, implementado mediante el tipo de la biblioteca de MALLBA `Rarray<cell>` (Apéndice B.2). Contiene toda la información obtenida del fichero de datos del problema y los cálculos de vecindarios en proceso.

Debido a que son atributos privados, se ofrecen métodos públicos asociados correspondientes para poder consultar esos valores. Además de esos métodos tenemos que decidir qué valor devuelve el método `direction`, que en este caso, al tratarse de minimización de una función, devolverá por tanto la constante `minimize`.

Por último, para realizar el proceso de cálculo del vecindario, se implementó para la clase `Problem` el método público `calculateVicinity`, el cual, para una celda RC, calcula su factor *vicinity* y actualiza los valores de las celdas nRCs pertenecientes a su vecindario.

La Clase `Solution`

Como se explicó en la Sección 3.4.1, la clase `Solution` representa una partícula del cúmulo. Una solución al problema LA se codifica mediante la posición de una partícula del PSO binario y consiste en una cadena de ceros y unos. Una celda nRC se representa mediante un 0 y una celda RC se representa mediante un 1. Por ejemplo, la red mostrada en la Figura 4.5 se representa mediante el vector $(0_0, 1_1, 1_2, 0_3, 1_4, 0_5, 1_6, 1_7, 1_8, 1_9, 0_{10}, 1_{11}, 0_{12}, 1_{13}, 1_{14}, 0_{15})$. Los subíndices indican el número de celda.

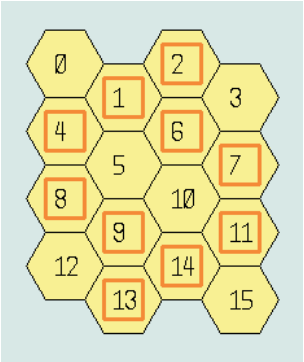

Figura 4.5: Red de RCs representadas con cuadrados oscuros

Además de los métodos comentados en la Sección 3.7.1 para el acceso a la posición actual, mejor posición y velocidad de la partícula, se destacan en esta clase los métodos para la inicialización y cálculo de fitness.

El método `Initialize`, crea una solución inicial, almacenando en el vector posición valores 0 ó 1 generados aleatoriamente. Inicialmente, el vector velocidad también toma valores aleatorios -1, 0 y 1.

Por último, el método `fitness` se encarga del cálculo del coste total de la red y de almacenarlo. Primero, genera celda a celda el valor de los vecindarios y el coste de N_{LU}. En una segunda fase, se obtiene el coste de N_p para finalizar calculando el coste total mediante la función de evaluación explicada en la Sección 4.1.2.

4.2. Caso de Estudio: Gene Ordering in Microarray Data (GOMAD)

En esta sección se explica el segundo problema objeto de estudio en este proyecto. Se trata del problema Gene Ordering in Microarray Data (GOMAD) que pasamos a describir a continuación.

Debido a la gran cantidad de datos generados por el *Proyecto del Genoma Humano*, así como toda la información que viene siendo generado por otras iniciativas relacionadas con la genética, la tarea de interpretar las relaciones funcionales entre los genes que aparecen en estos trabajos supone uno de los grandes desafíos abordados por la comunidad científica [36]. Bajo el punto de vista tradicional en Biología Molecular, el escenario de trabajo está basado en la premisa *"un gen en un experimento"*. Este enfoque normalmente limita el entendimiento respecto al *"genoma completo"*, haciendo la función de interacción entre los genes de complicado seguimiento. Por esta razón, se están desarrollando nuevas tecnologías en los últimos años. En este sentido, la famosa técnica de *Microarrays de ADN*[1] [9] está suscitando un gran interés puesto que permite monitorizar la actividad de un genoma completo en un simple experimento.

Con la intención de analizar la enorme cantidad de datos disponible, se hacen necesarias técnicas de reducción. En este sentido, se supone que un genoma está influenciado por no más de ocho o diez genes de media. Para realizar dicha reducción y permitir a los biólogos moleculares centrar sus trabajos en un significativo subconjunto de genes, se utilizan técnicas de clustering como *K-means* o métodos aglomerativos [18]. Sin embargo, todavía podrían ser mejores en cuanto a las soluciones que obtienen.

Para el análisis de microarrays de DNA se disponen un gran número de moléculas de DNA en un substrato sólido (slide) de manera sistemática. Un experimento de microarray consiste en la exposición de estas muestras de DNA a otras muestras de DNA complementario (cDNA). A este proceso se le llama

[1]En adelante usaremos DNA en vez de ADN para no confundirlo con las referencias y las figuras.

hibridación (hybridization en la Figura 4.6), obtenido a partir de aislamiento (isolation) con *ARN*[2] *mensajero* (mRNA). Estas moléculas mRNA se marcan con tinte fluorescente visible bajo microscopio (normalmente en rojo y verde para objetivo y muestra de referencia respectivamente). Debido a la naturaleza complementaria entre las moléculas de DNA y cDNA, se forman parejas mediante bases. Por supuesto, los moléculas cDNA que no corresponden con ningún gen en el microarray serán eliminadas del experimento. Finalmente, el microarray es escaneado midiendo la fluorescencia de cada muestra. Este valor de fluorescencia indica el nivel de expresión del gen correspondiente con respecto a cada muestra.

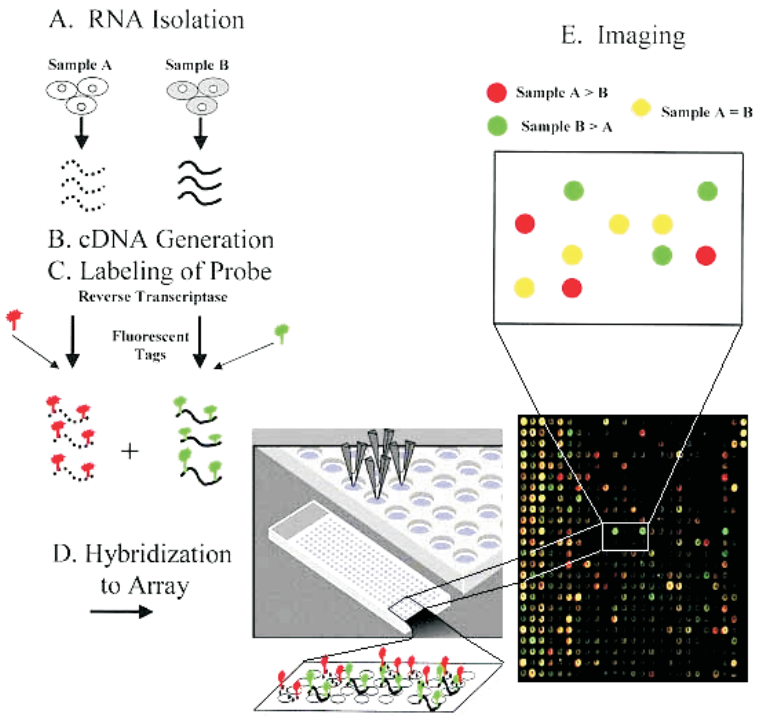

Figura 4.6: Proceso de obtención de un microarray

Para el procesamiento de la imagen escaneada, el microarray resultado se suele expresar como una matriz $G = \{g_{ij}\}_{i=1\ldots n}^{j=1\ldots m}$, donde n es el número de genes y m es el número de experimentos por gen. Utilizando esta representación, el objetivo ahora es encontrar un *orden óptimo* tal que los genes con similar patrón de expresión estén lo más próximos posibles en dicho orden. De este modo, se obtendrán regiones de similar expresión facilitando y mejorando el proceso de clustering a posteriori.

[2]En adelante usaremos RNA en vez de ARN para no confundirlo con las referencias y las figuras.

Al proceso de encontrar un orden óptimo de los genes en un microarray se le conoce como el problema de *Gene Ordering in Microarry Data* (GOMAD) [12, 37]. En este proyecto se ha implementado un algoritmo PSO para la resolución del GOMAD. A continuación se describe el modelo empleado para su representación, la función de evaluación utilizada y los detalles de implementación con el esqueleto PSO.

4.2.1. Modelo Empleado

La definición del problema GOMAD realizada en la sección anterior lleva implícita la noción de "distancia" como la medida de similitud entre genes. En la literatura se han utilizado diferentes medidas de distancia: los coeficientes de correlación de Pearson, τ de Kendall o Spearman son algunos ejemplos. En este trabajo se ha considerado como métrica la distancia Euclídea (Ecuación 4.5).

$$D[\pi_i, \pi_j] = \sqrt{\sum_{k=1}^{m} (\pi_{ik} - \pi_{jk})^2} \tag{4.5}$$

En esta ecuación, π_i y π_j representan dos genes. Cada gen viene codificado como un vector de m valores reales representando los niveles de expresión de cada muestra (véase la Figura 4.7). Para obtener una buena ordenación de los genes, se ha seguido en este estudio un algoritmo polinómico para la medición de distancias globales basado en el modelo del TSP (Traveling Salesman Problem). Bajo este criterio de optimalidad, se debe minimizar la suma de las distancias entre genes adyacentes en la secuencia del microarray. Este enfoque ha sido utilizado en el contexto de Algoritmos Meméticos [12]. Trabajando según el modelo TSP, cada gen se representa por un número entero desde el 1 hasta n (número total de genes del microarray) sin repeticiones ni omisiones (Figura 4.7). De esta forma, se obtiene un vector de enteros $(1, 2, ..., n)$ cuya distancia total es la suma de las distancias entre los enteros adyacentes. El orden óptimo vendrá dado por la permutación del vector de enteros con distancia total mínima.

Este simple criterio puede servir como una primera aproximación a la mejor ordenación, pero no es capaz de alcanzar el aspecto global de la secuencia resultante como se verá en la siguiente subsección.

4.2.2. Función de Evaluación

El gran inconveniente de este mecanismo para el cálculo de la distancia total en un microarray, es que sólo utiliza información de los genes inmediatamente adyacentes. Esta función tiene una visión bastante "miope" de una

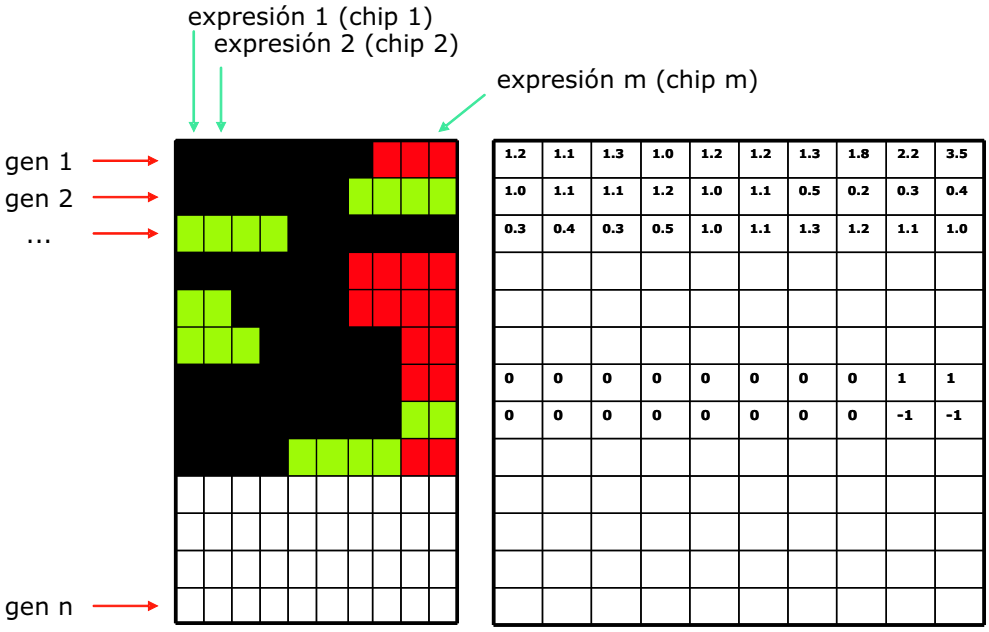

Figura 4.7: Representación de microarrays mediante vectores de enteros

solución, provocando que algunas soluciones con genes agrupados en conjuntos disjuntos reducidos sean evaluadas como buenas. Para realizar un buen agrupamiento de genes es necesario tener una visión "panorámica". En este tipo de situaciones, se suelen utilizar "ventanas flotantes". La función distancia total está formada en este caso por dos términos: el primer término suma las distancias entre el gen situado en el centro de la ventana y los genes situados dentro de la longitud de la ventana. El segundo término suma las distancias parciales según se va moviendo la ventana a lo largo de la secuencia. Siendo $\pi = \langle \pi_1, \pi_2, ...\pi_n \rangle$ el orden de los n genes de un vector solución, se representa la función de distancia total con ventana flotante como sigue:

$$Distancia\ Total(\pi) = \sum_{l=1}^{n} \sum_{i=max(l-s_w,1)}^{min(l+s_w,n)} w(i,l) D[\pi_l, \pi_i], \tag{4.6}$$

$$w(i,l) = s_w - |l-i| + 1 \tag{4.7}$$

donde $2s_w + 1$ es el tamaño de la ventana, y $w(i,l)$ es una función de peso que mide la influencia del gen situado en la posición l sobre el gen situado en la posición i. Considerando los pesos proporcionales a $s_w - |l-i| + 1$ (es decir, lineal en la distancia entre genes) y normalizado para que la suma de todos los pesos implicados en cada aplicación de la Ecuación 4.6 sea 1.

4.2.3. Resolución del Problema GOMAD con el Esqueleto PSO

Este problema se ha resuelto con el esqueleto PSO en la versión para permutaciones de enteros (ver Capítulo 3). Siguiendo la filosofía de desarrollo mediante esqueletos de código, la implementación del problema se realizó en las clases requeridas `Problem`, `Solution` y `User_statistics`. Para esta última clase se utilizó la implementación por defecto.

Antes de pasar a describir estas clases es necesario explicar la estructura de datos `m_array`. Mediante esta estructura se almacenan los datos del problema que son leídos del fichero de instancia del microarray. La estructura `m_array` comprende los siguientes campos:

- `number`, contiene el número de gen en la secuencia del microarray.
- `id`, especifica el nombre o número de catalogación biológica. No es necesario pero se mantiene por tener una referencia en los resultados.
- `gene`, arreglo de `double` que almacena los niveles de expresión de un determinado gen.

La Clase `Problem`

En la clase `Problem` se disponen de los siguientes atributos privados:

- `_dimension`, indica la dimensión del problema, es decir, el número de genes del microarray tratado.
- `_microarray`, es un arreglo de `m_array` implementado mediante el tipo de la biblioteca MALLBA `Rarray<m_array>`. Contiene toda la información sobre los genes y sus expresiones obtenida del fichero instancia del problema.

Debido a que son atributos privados, se necesitarán de los métodos asociados correspondientes para poder consultar esos valores. Además de esos métodos tenemos que decidir qué valor devuelve el método `direction` que en este caso, como se trata de minimización de una función, devolverá por tanto la constante `minimize`.

Por último, `distance` es un método específico al problema mediante el que se calcula la distancia (o similitud) entre dos genes del microarray comparando sus niveles de expresión muestra a muestra.

La Clase `Solution`

Referenciando de nuevo la Sección 3.5.1, la clase `Solution` representa una partícula del cúmulo. Una solución al problema GOMAD se codifica mediante la posición de una partícula del PSO para permutaciones de enteros, consistiendo en un vector de enteros sin repetición y sin omisiones. Cada gen del microarray se representa mediante un entero en el vector posición, por ejemplo (3,4,6,2,5,1). El orden de los enteros en este vector refleja la disposición de los genes en el microarray y la distancia total será la suma de las distancias entre los genes adyacentes. La distancia mínima vendrá dada por una permutación de este vector.

Para representar la velocidad de la partícula se utiliza una lista de pares tal y como se especifica en la Sección 3.5.1.

Además de los métodos comentados en la Sección 3.7.1 para el acceso a la posición actual, mejor posición y velocidad de la partícula, se destacan en esta clase métodos para: sumar velocidad a posición, inicialización y cálculo de fitness.

El método `vel_plus_pos`, implementa el operador de suma de la velocidad de la partícula a su posición. Para esto, según los genes indicados en cada par de la lista velocidad, se hacen los intercambios de dichos genes en el vector posición. Como resultado, se obtiene una permutación de la posición de la partícula.

El método `Initialize`, crea una solución inicial, almacenando en el vector posición valores enteros (sin repetición y sin omisión) generados aleatoriamente. Inicialmente, el vector velocidad está vacío.

Por último, el método `fitness` se encarga del cálculo de la distancia total entre la secuencia de genes del microarray. Para esto se codifica la función distancia total con ventana flotante especificada en la Sección 4.2.2.

Capítulo 5

Experimentos y Resultados

En este capítulo se muestran los experimentos realizados sobre los problemas LA y GOMAD, también se discutirán los resultados de esas pruebas. Cada apartado esta dedicado a un problema distinto y su estructura es la siguiente: primero se pasa a comentar las instancias empleadas en las pruebas, en segundo lugar se describen los parámetros de cada esqueleto con los que se han realizado las pruebas, tras esto se presentan los resultados y se finaliza con las conclusiones que se deducen de ellos. En cada problema, por cada esqueleto con el que hayan realizado las pruebas tenemos una prueba secuencial y dos en paralelo (LAN en modo síncrono y en modo asíncrono) en la red local que forma el entorno del que disponemos (ver el Apéndice A).

5.1. Problema LA

En esta sección pasamos a exponer los experimentos realizados sobre el primer caso de estudio abordado que es el problema LA.

5.1.1. Instancias

Las instancias utilizadas para el problema LA consisten en ficheros de datos en los que se incluyen los parámetros característicos de una red inalámbrica de celdas. Estos ficheros fueron generados mediante un simulador mediante el que se obtienen esquemas de redes que reflejan con gran aproximación configuraciones de la vida real. Todas las instancias, así como los resultados obtenidos tras su evaluación con otras técnicas de optimización, fueron proporcionados por investigadores de la *Escuela de Tecnologías de la Información* de la *Universidad de Sydney*.

Los ficheros de instancias contienen datos sobre la configuración de cada celda de una red LA con un tamaño determinado. Las redes son de topología cuadrada y se disponen de diferentes tamaños: 4×4, 6×6, 8×8 y 10×10 (Fi-

gura 5.1). Para cada tamaño se ha trabajado con tres instancias diferentes.

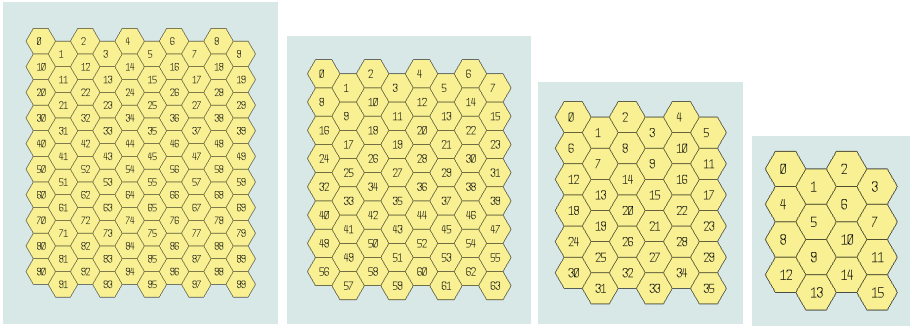

Figura 5.1: Topología y tamaños de la redes de celdas

En la Figura 5.2 aparece un ejemplo de fichero de instancia para una red de 10×10 celdas. Cada una de estas celdas se identifica mediante una etiqueta entre corchetes (por ejemplo, [Cell1]), y contiene información sobre el número de actualizaciones de localización (Number of Location Update: 304) y sobre el número de llamadas recibidas (Number of Arrived Calls: 98).

```
[Network]
Number of Cells=100

[Cell 0]
Number of Location Updates:144
Number of Arrived Calls:83

[Cell 1]
Number of Location Updates:304
Number of Arrived Calls:98

[Cell 2]
Number of Location Updates:201
Number of Arrived Calls:66
.
.
.
[Cell 99]
Number of Location Updates:162
Number of Arrived Calls:82
```

Figura 5.2: Fichero de instancia de una red de 10×10 celdas

5.1.2. Descripción de los Experimentos

Para el problema LA se han realizado experimentos mediante el esqueleto del algoritmo PSO para codificación binaria (ver Apartado 4.1.3), al cual se le ha añadido un mecanismo de *búsqueda local* de un nivel sobre cada partícula en cada iteración. Como se ha comentado anteriormente, se dispone de tres instancias por cada tamaño de red.

Para cada instancia se han realizado 50 ejecuciones independientes en cada una de las siguientes versiones del PSO: secuencial (SEQ), paralelo en

modo síncrono (LAN_SYNC) y paralelo en modo asíncrono (LAN_ASYNC). Las pruebas de las versiones paralelas se realizaron en el entorno LAN especificado en el Apéndice A.

La configuración del esqueleto viene determinada en el fichero `PSO.cfg` como se muestra en la Figura 5.3. Los valores marcados con * varían dependiendo de la instancia y el tipo de experimento.

```
General
50          // number of independent runs
100         // number of generations
*           // Swarm size
*           // Particle size
8           // Neighborhood size
0           // display state ?
Binary-pso-params
-10.0       // deltaMin
10.0        // deltaMax
Weight-factors
1.5         // iWeight
0.0         // iMin
1.0         // iMax
2.5         // sWeight
0.0         // sMin
1.0         // sMax
1.4         // Wmax
0.4         // Wmin
Migration-params
8           // Migration freq
LAN-configuration
100         // refresh global state
*           // 0: running in asynchronized mode / 1: running in synchronized mode
1           // interval of generations to check solutions from other populations
```

Figura 5.3: Fichero de configuración del esqueleto PSO binario

El tamaño del cúmulo, representado en *swarm_size*, toma el valor de 100 en los experimentos con el PSO secuencial y 20 en los experimentos con el PSO en paralelo. Esto se debe a que las pruebas en paralelo utilizan 5 máquinas (ver Apéndice A), por lo que para compensar el número de partículas (5 cúmulos de 20 partículas $\simeq$ 1 cúmulo de 100 partículas) se reduce el cúmulo a este valor.

El tamaño de las partículas representado en el valor *particle_size* será 16, 36, 64 ó 100 dependiendo del tamaño de red de la instancia (4×4, 6×6, 8×8 ó 10×10 respectivamente).

Por último, en la configuración LAN (`LAN-configuration`), el segundo valor será 0 si se ejecuta en modo asíncrono y 1 si se ejecuta en como síncrono.

5.1.3. Resultados

En este apartado se muestran los resultados de las pruebas explicadas en el apartado anterior. En las siguientes tablas se mostrarán los resultados obtenidos tanto en ejecuciones secuenciales, LAN en modo síncrono y LAN en modo asíncrono.

Todas las tablas tienen el mismo formato. En la primera columna se disponen los tipos de red LA con los números de instancia entre paréntesis. En la segunda columna aparecen los resultados obtenidos con un algoritmo que combina una red neuronal de Hopfield con la técnica de *Ball Dropping* (HNN+BD) [52], estos resultados son óptimos. El resto de columnas corresponden a resultados y estadísticas obtenidas de la ejecución del algoritmo PSO. Las abreviaturas utilizadas significan lo siguiente:

- **M** es el fitness (según la Ecuación 4.4) de la mejor solución encontrada,
- **MV** es la media de las mejores soluciones encontradas en cada ejecución independiente,
- **PE** es el porcentaje de error calculado como (mejor conocido - mejor encontrado) / mejor conocido,
- **E** evaluación en la que se encontró la mejor solución,
- **T** tiempo (nº de segundos s) en que se encontró la mejor solución y
- **H** es el número de veces (*Hit*) que se encontró la mejor solución.

	HNN+BD	**PSO Binario SEQ**					
Red (N°)	**M**	**M**	**MV**	**PE**	**E**	**T**	**H**
4×4 (1)	98535	**98535**	98535	0.00000 %	32	0.17 s	50
4×4 (2)	97156	**97156**	97156	0.00000 %	16	0.17 s	50
4×4 (3)	95038	**95038**	95038	0.00000 %	16	0.17 s	50
6×6 (1)	173701	173753	182255	0.00029 %	2880	45.42 s	0
6×6 (2)	182331	**182331**	188506	0.00000 %	360	7.19 s	1
6×6 (3)	174519	**174519**	176153	0.00000 %	360	21.55 s	2
8×8 (1)	308929	308972	315053	0.00013 %	3328	966.54 s	0
8×8 (2)	287149	**287149**	301383	0.00000 %	1472	78.55 s	1
8×8 (3)	264204	265594	270998	0.00526 %	3968	684.44 s	0
10×10 (1)	379057	382705	390143	0.00962 %	5000	834.78 s	0
10×10 (2)	358167	363580	372271	0.01510 %	8400	1955.43 s	0
10×10 (3)	370868	378220	387101	0.01981 %	8000	2987.80 s	0

Tabla 5.1: Valores de fitness y estadísticas obtenidas de las pruebas secuenciales para el problema LA

Algunas configuraciones de redes LA obtenidas se muestran en las figuras 5.4 y 5.5. Concretamente se disponen los resultados de las redes 8×8 **(1)** y 10×10 **(2)** tanto obtenidos por el algoritmo HNN+BD como los obtenidos por

el algoritmo PSO binario en las ejecuciones secuenciales. A pesar de obtener resultados de coste total (fitness) muy parecidos, se puede apreciar la diferencia en la disposición de las *Reporting Cells* (representadas con cuadrados oscuros). Por ejemplo, en la Figura 5.4 el fitness obtenido por HNN+BD es de 308929, mientras que la configuración obtenida con PSO binario en esta misma figura corresponde a un fitness de 308972 (el porcentaje de error es de 0.00013 %). En el caso de la Figura 5.5, el fitness correspondiente a la configuración de red obtenida por el algoritmo HNN+BD es de 358106, mientras que el algoritmo PSO binario obtiene una configuración con fitness 363580 (con un porcentaje de error del 0.0151 %). Estos datos están incorporados en la Tabla 5.3.

	HNN+BD	**PSO Binario LAN SYNC**					
Red (N°)	**M**	**M**	**MV**	**PE**	**E**	**T**	**H**
4×4 (1)	98535	**98535**	98535	0.00000 %	16	0.17 *s*	50
4×4 (2)	97156	**97156**	97156	0.00000 %	16	0.17 *s*	50
4×4 (3)	95038	**95038**	95038	0.00000 %	16	0.17 *s*	50
6×6 (1)	173701	**173701**	176460	0.00000 %	1836	11.01 *s*	2
6×6 (2)	182331	**182331**	184711	0.00000 %	203	0.99 *s*	4
6×6 (3)	174519	175182	177116	0.00037 %	396	13.91 *s*	0
8×8 (1)	308929	310928	316444	0.00647 %	3904	61.61 *s*	0
8×8 (2)	287149	289041	297831	0.00658 %	3272	11.42 *s*	0
8×8 (3)	264204	267107	270131	0.01098 %	3840	83.08 *s*	0
10×10 (1)	379057	380398	390734	0.00353 %	4200	107.59 *s*	0
10×10 (2)	358167	364785	371376	0.01847 %	6500	72.41 *s*	0
10×10 (3)	370868	380427	384807	0.02577 %	5600	151.87 *s*	0

Tabla 5.2: Valores de fitness y estadísticas obtenidas de las pruebas LAN en modo síncrono con cinco máquinas

	HNN+BD	**PSO Binario LAN ASYNC**					
Red (N°)	**M**	**M**	**MV**	**PE**	**E**	**T**	**H**
4×4 (1)	98535	**98535**	98535	0.00000 %	16	0.17 *s*	50
4×4 (2)	97156	**97156**	97156	0.00000 %	16	0.17 *s*	50
4×4 (3)	95038	**95038**	95038	0.00000 %	16	0.17 *s*	50
6×6 (1)	173701	**173701**	177080	0.00000 %	1872	1.52 *s*	1
6×6 (2)	182331	**182331**	184712	0.00000 %	2520	2.01 *s*	18
6×6 (3)	174519	**174519**	176616	0.00000 %	2160	1.44 *s*	2
8×8 (1)	308929	310823	315481	0.00613 %	4352	64.05 *s*	0
8×8 (2)	287149	290942	298579	0.01320 %	1152	108.26 *s*	0
8×8 (3)	264204	267296	277181	0.01170 %	4224	61.99 *s*	0
10×10 (1)	379057	384126	392731	0.01337 %	5600	266.90 *s*	0
10×10 (2)	358167	370813	380597	0.03530 %	3500	212.91 *s*	0
10×10 (3)	370868	380275	383881	0.02536 %	4700	186.97 *s*	0

Tabla 5.3: Valores de fitness y estadísticas obtenidas de las pruebas LAN en modo asíncrono con cinco máquinas

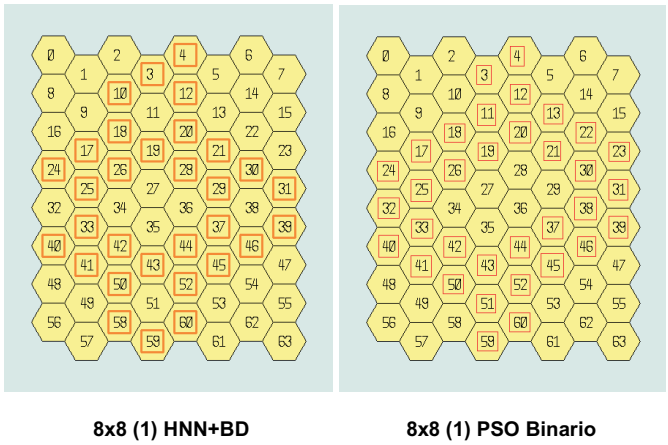

Figura 5.4: Mejores configuraciones de redes LA obtenidas obtenidas por HNN+BD (Izq.) y PSO binario (Der.) a partir de la instancia 8×8 (1)

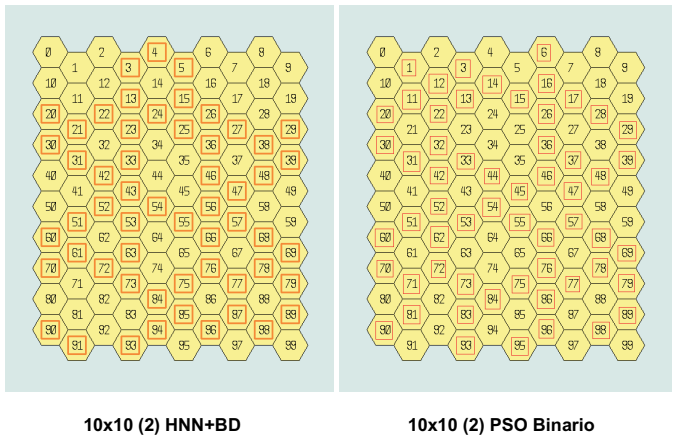

Figura 5.5: Mejores configuraciones de redes LA obtenidas obtenidas por HNN+BD (Izq.) y PSO binario (Der.) a partir de la instancia 10×10 (2)

Otro aspecto a tener en cuenta sobre el funcionamiento del esqueleto PSO binario en la resolución del problema LA es la progresión del mejor fitness que éste va obteniendo en cada iteración. En la Figura 5.6, se muestra una gráfica sobre la evolución del mejor fitness obtenido (cada diez iteraciones) para el problema LA (instancia 10×10 (3)) con el PSO secuencial, el PSO LAN en modo síncrono y el PSO LAN en modo asíncrono. El perfil de esta gráfica es similar para todas las instancias de LA por lo que se ha elegido como un ejemplo significativo.

Del mismo modo, en las gráficas de la Figura 5.7 se muestran las medias aritméticas y las desviaciones típicas (AVG y STD respectivamente) de los valores de fitness de todas las partículas del cúmulo muestreados cada diez iteraciones sobre la misma instancia del problema LA. Las barras en estas

gráficas corresponden a las desviaciones típicas con el punto medio en la media aritmética de cada iteración muestreada.

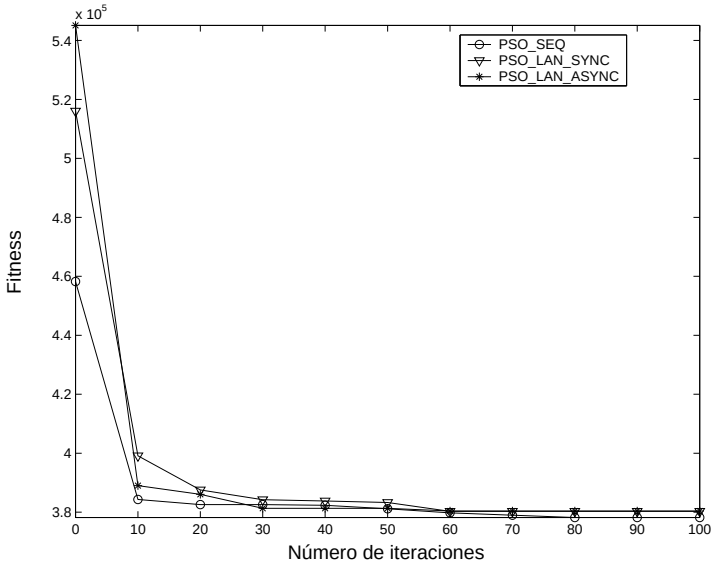

Figura 5.6: Mejor fitness obtenido (muestreado cada diez iteraciones) por el PSO secuencial (PSO_SEQ), el PSO LAN en modo síncrono (PSO_LAN_SYNC) y el PSO LAN en modo asíncrono (PSO_LAN_ASYNC) para la instancia LA 10×10 (3)

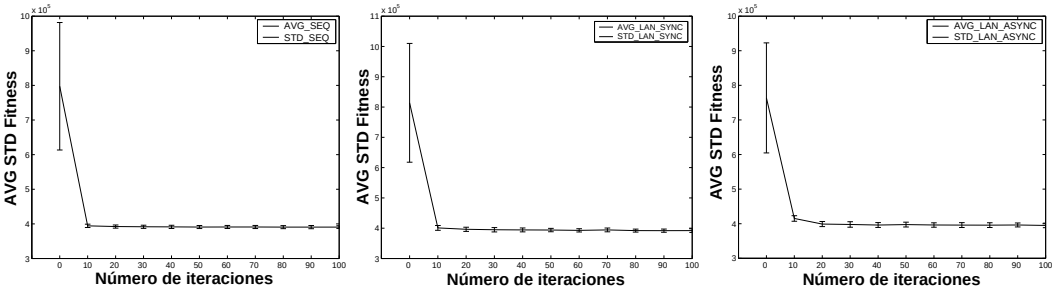

Figura 5.7: Media aritmética (AVG) y desviación típica (STD) muestreadas cada diez iteraciones del algoritmo PSO secuencial (PSO_SEQ), el algoritmo PSO LAN en modo síncrono (PSO_LAN_SYNC) y el algoritmo PSO LAN en modo asíncrono (PSO_LAN_ASYNC) para la instancia LA 10×10 (3)

Por último, hemos realizado pruebas estableciendo como final de la ejecución un cierto fitness para medir el *speedup* (SU) que claramente falta en las ejecuciones del algoritmo PSO en las versiones paralelas con respecto a las ejecuciones del mismo algoritmo en la versión secuencial.

Para el cálculo del *speedup* hemos utilizado la fórmula estándar (Ecuación 5.1) en la que p es el número de procesadores, T_1 es el tiempo de ejecución del algoritmo secuencial y T_p es el tiempo de ejecución del algoritmo paralelo con p procesadores.

$$SU = \frac{T_1}{T_p} \tag{5.1}$$

De esta forma, se obtiene un *speeup ideal* cuando $SU = p$ lo cual se considera un buen factor de escalabilidad. En nuestro caso, el *speeup ideal* sería 5 pues es el número de procesadores que utilizamos en las versiones paralelas.

Para obtener una estimación de cómo de buena es la paralelización realizada, hemos utilizado la Ecuación 5.2 mediante la que se calcula el tanto por ciento de la eficiencia (E) lograda. Así, podemos hacernos una idea del esfuerzo computacional empleado en tareas como la sincronización y la comunicación entre procesos.

$$E = \frac{SU}{p} \times 100 \tag{5.2}$$

En la Tabla 5.4, se muestran los valores de *Speedup* y *Eficiencia* calculados respecto a las ejecuciones secuenciales y LAN en modo síncrono y respecto a las ejecuciones secuenciales y LAN en modo asíncrono para todas las instancias del problema LA. En la columna **Final**, se disponen los fitness de parada establecidos para cada instancia.

Red (N°)	$Speedup_{s/ls}$	$Speedup_{s/las}$	$Eficiencia_{s/ls}$	$Eficiencia_{s/las}$	**Final**
4×4 (1)	0.849	0.833	16 %	16 %	98535
4×4 (2)	0.841	0.827	16 %	16 %	97156
4×4 (3)	0.843	0.857	16 %	16 %	95038
6×6 (1)	2.892	3.044	57 %	60 %	180000
6×6 (2)	2.083	2.911	41 %	58 %	184000
6×6 (3)	1.695	2.146	33 %	42 %	177000
8×8 (1)	1.275	3.397	25 %	67 %	316000
8×8 (2)	1.768	2.962	35 %	59 %	300000
8×8 (3)	3.078	3.825	61 %	76 %	275000
10×10 (1)	1.620	2.451	32 %	49 %	395000
10×10 (2)	4.620	4.724	92 %	94 %	380000
10×10 (3)	1.981	4.656	39 %	93 %	387000

Tabla 5.4: *Speedup* y *Eficiencia* calculados respecto a las ejecuciones secuencial y LAN en modo síncrono (*s/ls*) y respecto a las ejecuciones secuencial y LAN en modo asíncrono (*s/las*) para el problema LA

5.1.4. Conclusiones

Pese a que las instancias del problema probadas no han presentado gran dificultad al esqueleto y en gran parte de las ejecuciones se ha conseguido alcanzar el valor óptimo del problema (y las que no lo han conseguido se han quedado muy cerca) presentado por los resultados obtenidos por el algoritmo HNN+BD, se pueden sacar bastante conclusiones acerca del esqueleto de código empleado.

La primera observación es que el PSO binario resuelve adecuadamente las instancias más pequeñas y en un tiempo razonable, mientras que para las instancias mayores se acerca bastante al óptimo pero no consigue obtenerlo en la gran mayoría de los casos. Debido a que el algoritmo PSO se ha diseñado en este proyecto como un esqueleto de propósito general, es complicado obtener resultados óptimos tal y como lo hace HNN+BD, pues éste trabaja con operadores más específicos a la naturaleza del problema.

Respecto a los resultados obtenidos por las versiones del PSO (secuencial, LAN síncrono y LAN asíncrono), hay que destacar que en general las pruebas con el esqueleto secuencial obtienen mejores resultados que las pruebas del esqueleto paralelo (tal y como reflejan los porcentajes de error en las tablas). Esto se debe a que en la paralelización las partículas del cúmulo completo se reparten entre las distintas islas, lo que disminuye la diversidad dentro de cada una proporcionando una convergencia más rápida a soluciones subóptimas. En este caso, la migración de soluciones no compensa la menor diversidad y de ahí la obtención de peores soluciones. Sin embargo, el algoritmo secuencial trabaja con más diversidad ya que utiliza el cúmulo completo durante toda la ejecución. Ajustando los parámetros de migración (aumentando la frecuencia de intercambio) lograremos acercar el comportamiento del PSO paralelo al secuencial.

Otro aspecto es el tiempo empleado en encontrar la mejor solución. Las ejecuciones secuenciales en general requieren un mayor tiempo que las ejecuciones paralelas, puesto que trabajando con más procesadores (cinco más en este caso) paralelamente se agiliza de manera considerable la obtención de soluciones óptimas, a pesar de los retardos causados por la comunicación entre procesos y la sincronización.

Además, la diferencia entre las configuraciones del esqueleto paralelo, es decir, LAN síncrono y asíncrono, se manifiesta principalmente en el tiempo empleado en encontrar las mejores soluciones. Generalmente (salvo casos excepcionales), las ejecuciones síncronas requieren más tiempo puesto que no se realizan migraciones hasta que todos los procesos lleguen a un mismo punto de ejecución (barrera), por lo que todos los procesos deben esperar hasta que el más lento llegue a dicho punto dependiendo de la frecuencia de migración.

Bajo el punto de vista del *speedup* y de la *eficiencia* obtenida en la paralelización, la versión LAN asíncrona obtiene generalmente una mejor *eficiencia*

que la versión síncrona por los motivos comentados anteriormente. Por otra parte, respecto al tamaño de la instancia manejada, se puede observar que para instancias pequeñas del problema, la paralelización del algoritmo no supone un gran ahorro de tiempo pues el *speedup* con respecto al algoritmo secuencial es bastante bajo. En este caso los tiempos requeridos para alcanzar las soluciones óptimas es muy bajo en todos los casos y no se aprecia diferencia entre las ejecuciones secuenciales y paralelas. Sin embargo, para instancias mayores se obtienen mejores niveles de *speedup*, ya que el tiempo requerido en las ejecuciones secuenciales crece bastante respecto a las ejecuciones paralelas, por lo tanto, en estos casos la paralelización contribuye de manera significativa en la mejora de la *eficiencia* del algoritmo.

Por último, se puede decir que el esqueleto PSO binario resuelve el problema LA proporcionando buenos resultados en un tiempo razonable. Sin embargo, para las instancias de mayor tamaño, se obtienen resultados más alejados del óptimo y se requiere un esfuerzo computacional mayor.

5.2. Problema GOMAD

En esta sección pasamos a describir los experimentos realizados sobre el problema GOMAD segundo caso de estudio abordado en este proyecto.

5.2.1. Instancias

Las instancias utilizadas para el problema GOMAD consisten en grandes ficheros de datos que contienen los valores de expresión de los genes muestreados en experimentos de microarrays. Estos ficheros corresponden a experimentos reales, y fueron obtenidos de las bases de datos públicas *GEPAS* (con URL `http://www.es.embnet.org/Services/MolBio/gepas/index.html`) y *Wohl Virion Centre DB* (con URL `http://www.biochem.ucl.ac.uk/bsm/virus_database/`). Las instancias utilizadas para este proyecto son las siguientes:

- HERPES: estos datos fueron obtenidos de [29]. Se trata de muestras obtenidas en el proceso de inducción del sarcoma asociado al virus *herpes de Kaposi*. Comprenden 106 genes (21 muestra por gen).
- FIBROBLAST: estos datos fueron obtenidos de [19]. Corresponden a la respuesta del *fibroblasto* humano a la aplicación de suero en la cicatrización. Comprenden 517 genes (19 muestras por gen).
- DIAUXIC: estos datos fueron obtenidos de [14]. Corresponden a un experimento hecho por un grupo de Stanford sobre el *salto diauxico*, es decir, el paso de condiciones aerobias a condiciones anaerobias en *Saccharomyces cerevisiae* (levadura de la cerveza). Comprenden 210 genes (7 muestras por gen).

En estos ficheros, se disponen los genes en filas en las que el primer valor es el identificador de referencia biológica del gen y a continuación están sus correspondientes valores de muestras.

5.2.2. Descripción de los Experimentos

Para el problema GOMAD se han realizado pruebas mediante el esqueleto del algoritmo PSO para permutaciones de enteros (ver Apartado 4.2.1), al que se le ha añadido un mecanismo de *búsqueda local* de un nivel sobre la mejor partícula del cúmulo en cada iteración. Como se ha comentado anteriormente, se dispone de tres instancias correspondientes a tres experimentos de microarray.

Para cada instancia se han realizado 30 ejecuciones independientes, ya que 30 es un número significativo de ejecuciones y al requerir un gran esfuerzo computacional no se han realizado 50 ejecuciones independientes como en el problema LA. Típicamente, en LA la instancia más compleja tarda 2987

segundos mientras que en GOMAD tarda 26972 segundos (26972 >> 2987). Estas pruebas se han realizado sobre cada una de las siguientes versiones del PSO: secuencial (SEQ), paralelo en modo síncrono (LAN_SYNC) y paralelo en como asíncrono (LAN_ASYNC). El entorno LAN en el que se ejecutaron las versiones paralelas se ha especificado en el Apéndice A.

La configuración del esqueleto viene determinada en el fichero `PSO.cfg` como se muestra en la Figura 5.8. Los valores marcados con * varían dependiendo de la instancia y el tipo de prueba.

```
General
30          // number of independent runs
500         // number of generations
*           // Swarm size
*           // Particle size
8           // Neighborhood size
0           // display state ?
Binary-pso-params
-10.0       // deltaMin
10.0        // deltaMax
Weight-factors
2.0         // iWeight
0.0         // iMin
1.0         // iMax
2.0         // sWeight
0.0         // sMin
1.0         // sMax
1.4         // Wmax
0.4         // Wmin
Migration-params
8           // Migration freq
LAN-configuration
100         // refresh global state
*           // 0: running in asynchronized mode / 1: running in synchronized mode
1           // interval of generations to check solutions from other populations
```

Figura 5.8: Fichero de configuración del esqueleto PSO para permutaciones de enteros

El parámetro *swarm_size* así como los parámetros de la etiqueta `LAN -configuration` toman los mismos valores que en la configuración del problema LA.

El tamaño de partícula, representado en el parámetro *particle_size* se inicializa con los valores 106, 517 ó 210, que son el número de genes de cada instancia utilizada (HERPES, FIBROBLAST y DIAUXIC respectivamente).

5.2.3. Resultados

En este apartado se muestran los resultados de las pruebas realizadas sobre el problema GOMAD con el algoritmo PSO para permutaciones de enteros. En las tablas 5.5, 5.6 y 5.7 se muestran los resultados obtenidos tanto en ejecuciones secuenciales, LAN en modo síncrono y LAN en modo asíncrono.

La nomenclatura seguida en la disposición de los resultados en las tablas es la siguiente.

- **M** es el fitness (según la ecuación 4.6) de la mejor solución encontrada,
- **MV** es la media de las mejores soluciones encontradas en cada ejecución independiente,
- **PE** es el porcentaje de error calculado como (mejor conocido - mejor encontrado) / mejor conocido,
- **E** evaluación en la que se encontró la mejor solución,
- **T** tiempo (nº de segundos s) en que se encontró la mejor solución y
- **H** es el número de veces (*Hit*) que se encontró la mejor solución.

	Memético	PSO para Permutaciones de Enteros SEQ					
Instancia	**MV**	**M**	**MV**	**PE**	**E**	**T**	**H**
HERPES	598.798	500.478	**554.202**	0.00000 %	39800	707.54 s	1
FIBROBLAST	1376.804	1359.680	1391.060	0.01035 %	50000	21436.40 s	0
DIAUXIC	-	355.047	**388.323**	0.00000 %	49700	797.55 s	1

Tabla 5.5: Valores de fitness y estadísticas obtenidas de las pruebas secuenciales para el problema GOMAD

	Memético	PSO para Permutaciones LAN SYNC					
Instancia	**MV**	**M**	**MV**	**PE**	**E**	**T**	**H**
HERPES	598.798	497.459	**559.080**	0.00000 %	50000	1586.71 s	1
FIBROBLAST	1376.804	1359.680	**1362.670**	0.00000 %	47000	16652.60 s	1
DIAUXIC	-	349.232	**381.630**	0.00000 %	49900	909.14 s	1

Tabla 5.6: Valores de fitness y estadísticas obtenidas de las pruebas LAN en modo síncrono utilizando cinco máquinas

	Memético	PSO para Permutaciones LAN ASYNC					
Instancia	**MV**	**M**	**MV**	**PE**	**E**	**T**	**H**
HERPES	598.798	496.263	**552.345**	0.00000 %	41900	375.03 s	1
FIBROBLAST	1376.804	1362.980	1386.700	0.00000 %	50000	26972.40 s	0
DIAUXIC	-	345.796	**374.734**	0.00000 %	50000	2206.16 s	1

Tabla 5.7: Valores de fitness y estadísticas obtenidas de las pruebas LAN en modo asíncrono utilizando cinco máquinas

En la segunda columna de cada una de estas tablas se pueden ver otros resultados (disponibles en la literatura [12]) obtenidos con *Algoritmos Meméticos*. Para este estudio se utilizaron las mismas instancias de HERPES y FIBROBLAST (no disponemos de resultados para DIAUXIC) además de la misma función de evaluación que en este proyecto. Estos valores corresponden a la media de los fitness obtenidos en diez ejecuciones independientes del Algoritmo Memético con configuraciones diferentes de búsqueda local y paralelismo.

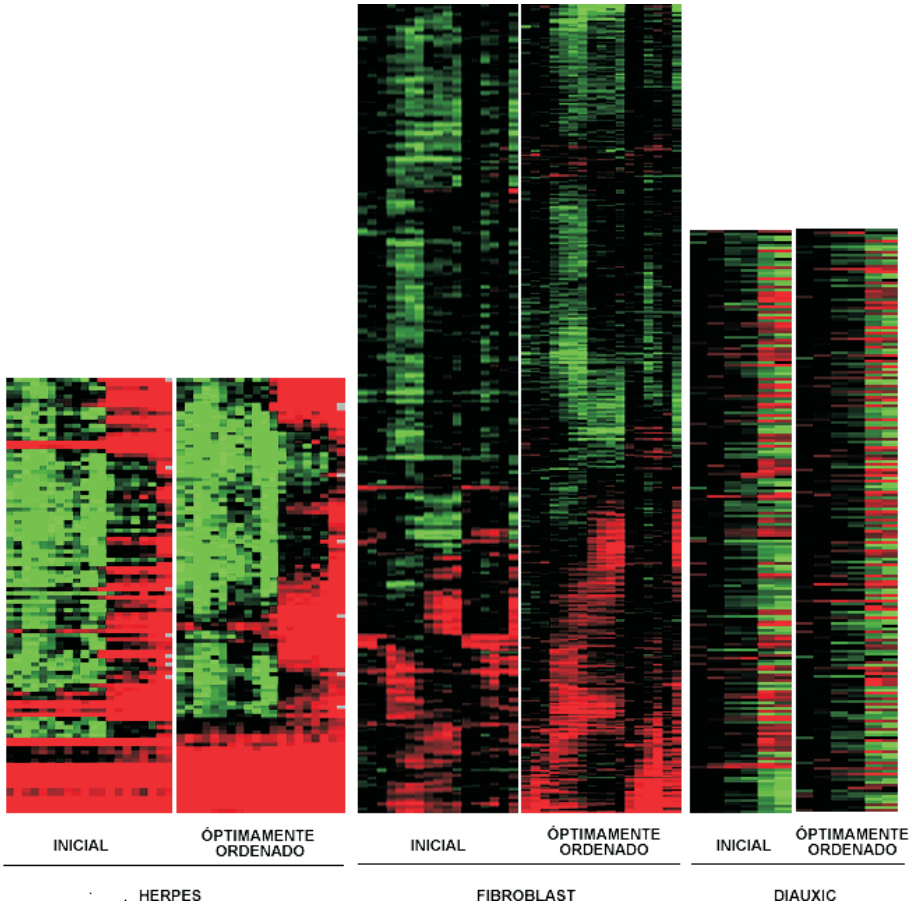

Figura 5.9: Microarrays inicial y óptimamente ordenado de las tres instancias objeto de estudio

Una de las mejores formas de comprobar el efecto de la ordenación de genes en un microarray es comparando la disposición de éstos antes y después de la ordenación. En la Figura 5.9, se muestran las imágenes de los microarrays de las instancias procesadas con el microarray inicial a la izquierda y el microarray óptimamente ordenado a la derecha. Como se puede observar, los microarrays ordenados presentan mayores áreas homogéneas en la distribución de los niveles de color, facilitando de este modo la posterior clusterización.

Por motivos de espacio en este documento, el tamaño los microarrays presentados en la Figura 5.9 no es exactamente proporcional entre los tres ejemplos, aunque se ha escalado perfectamente el tamaño de cada microarray inicial con respecto al ordenado.

En la Figura 5.10, se muestra una gráfica de la evolución que experimenta el fitness en ejecuciones independientes del PSO para permutaciones de enteros en la resolución del problema GOMAD. Los valores presentados corresponden a los fitness muestreados cada cincuenta iteraciones (de un total

de quinientas) en las ejecuciones del PSO secuencial, el PSO LAN en modo síncrono y el PSO LAN en modo asíncrono. Para el problema se ha utilizado la instancia HERPES, aunque el perfil de la gráfica resultado es similar en las demás instancias.

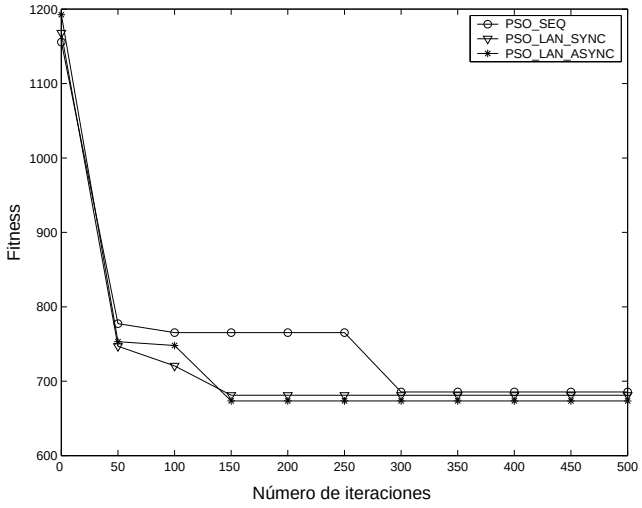

Figura 5.10: Mejor fitness obtenido (muestreado cada cincuenta iteraciones) por el PSO secuencial (PSO_SEQ), el PSO LAN en modo síncrono (PSO_LAN_SYNC) y el PSO LAN en modo asíncrono (PSO_LAN_ASYNC) para la instancia de GOMAD HERPES

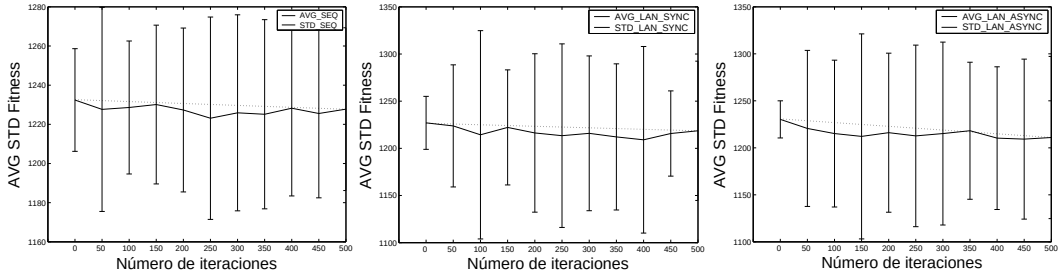

Figura 5.11: Medias aritméticas y desviaciones típicas muestreadas cada cincuenta iteraciones para el PSO secuencial (PSO_SEQ), el PSO LAN en modo síncrono (PSO_LAN_SYNC) y el PSO LAN en modo asíncrono (PSO_LAN_ASYNC) para la instancia de GOMAD HERPES

Correspondiendo con la gráfica anterior, en la Figura 5.11 se presentan las gráficas de barras de las medias aritméticas y las desviaciones típicas de los valores de fitness de todas las partículas del cúmulo.

De la misma manera que en el problema anterior (Apartado 5.1.3), terminamos este apartado presentando los valores de *speedup* y *eficiencia* (Ta-

bla 5.8) calculados a partir de las ejecuciones del algoritmo PSO para permutaciones de enteros en sus versiones secuencial y paralelas.

Instancia	$Speedup_{s/ls}$	$Speedup_{s/las}$	$Eficiencia_{s/ls}$	$Eficiencia_{s/las}$	**Final**
HERPES	3.269	3.569	65 %	71 %	600
FIBROBLAST	2.786	2.834	55 %	56 %	1390
DIAUXIC	2.529	2.116	50 %	42 %	390

Tabla 5.8: *Speedup* y *Eficiencia* calculados respecto a las ejecuciones secuencial y LAN en modo síncrono (*s/ls*) y respecto a las ejecuciones secuencial y LAN en modo asíncrono (*s/las*) para el problema GOMAD

5.2.4. Conclusiones

Para este problema se han utilizado instancias de diferente magnitud tanto en el número de genes como en la cantidad de muestras tomadas de cada uno de ellos. Comparando estos valores con los resultados obtenidos por el Algoritmo Memético, podemos pensar que el algoritmo PSO para permutaciones de enteros trabaja de manera adecuada, ya que en la mayoría de los casos se obtienen resultados similares e incluso mejores que este algoritmo.

Posiblemente, parte del buen funcionamiento del esqueleto PSO para permutaciones de enteros reside en la naturaleza de los operadores. Casualmente, para este problema trabajan de manera adecuada, aunque es necesario estudiar si este comportamiento se mantiene para otros problemas que utilizan representaciones con permutaciones de enteros.

De esta forma, por ejemplo, el operador resta de posiciones ($\ominus$ en la Sección 3.5.2) introduce un gran número de pares de intercambios en el vector velocidad. Con esto se consigue una gran variabilidad en las permutaciones obtenidas como solución tras el movimiento de las partículas, y de este modo, diversificar el conjunto de soluciones en proceso. Unido a la búsqueda local de un nivel realizada sobre la mejor partícula en cada iteración se consigue cierto grado de intensificación, lo que ayuda a la obtención de óptimos que desvían la dirección general del cúmulo. El efecto de esta diversificación se manifiesta en la gráfica de las medias y desviaciones típicas de la sección anterior, pues, mientras que la media de los valores de fitness mejoran, la desviación típica se mantiene estable.

En cuando a las configuraciones del PSO, es interesante destacar que según reflejan las tablas del apartado anterior, la ejecución paralela LAN asíncrona es la que proporciona mejores resultados (medias) empleando el menor tiempo para ello. Al ser GOMAD un problema que trabaja con grandes instancias, la evaluación de una solución comprende gran parte de la carga de proceso. Por este motivo, la paralelización del algoritmo en islas que trabajan con subcúmulos más pequeños agiliza de manera considerable la ejecución de

una iteración del algoritmo. Añadido a que la comunicación se realiza de manera asíncrona, las islas disponen rápidamente de nuevas partículas migradas, contribuyendo así a la diversidad dentro de cada subcúmulo.

Continuando con este razonamiento, el *speedup* obtenido en la paralelización proporciona niveles de *eficiencia* aceptables para todas las instancias, por lo que podemos afirmar que las versiones LAN del PSO para permutaciones de enteros trabajan adecuadamente.

Capítulo 6

Conclusiones y Trabajo Futuro

En este capítulo se hace un resumen de los resultados conseguidos a lo largo de este proyecto. Además de este resumen, se describen las dificultades en la realización del proyecto, terminando con un apartado donde se comentan las posibles extensiones futuras que se podrán realizar para obtener mejores resultados tanto en la resolución de los problemas aquí descritos como en una amplia gama de problemas de la vida real.

6.1. Resumen de Resultados

Como resultado de la realización de este proyecto fin de carrera, se puede indicar que se ha implementado un esqueleto del algoritmo PSO con diferentes versiones para codificación continua, binaria y para permutaciones de enteros. Este esqueleto supone una extensión de la biblioteca MALLBA y podrá ser aplicado a una gran variedad de problemas de optimización. Las tres versiones incorporan características comunes, que facilitarán más aún su aprendizaje, e implementan mecanismos de comunicaciones para las versiones distribuidas de los algoritmos.

Para evaluar la eficiencia de las versiones del PSO implementadas, se han resuelto dos problemas de carácter práctico como son LA y GOMAD. Con las pruebas realizadas se ha visto el comportamiento de el algoritmo desarrollado. En general, para problemas relativamente complejos los algoritmos responden positivamente encontrando buenas soluciones con relativa facilidad. No obstante, en ambos casos ha sido necesario la incorporación de un mecanismo de búsqueda local para intensificar la exploración durante la ejecución de los algoritmos.

Para la resolución de problemas complejos, sin la necesidad de mecanismos de búsqueda local, se necesitaría realizar un estudio en profundidad de cómo aportar conocimiento a los algoritmos para acotar el espacio de búsqueda.

Sin embargo, cabe destacar el rendimiento de la versión PSO para permutaciones de enteros, pues para las instancias utilizadas en los experimentos, consigue mejores resultados que los obtenidos por otras técnicas metaheurísticas evaluadas en trabajos similares.

6.2. Dificultades Encontradas

En la realización de este proyecto me he encontrado con bastantes dificultades y aunque algunas tenían un resolución relativamente fácil, otras (sobre todo las relacionadas con la la versión WAN del esqueleto) no han podido ser resueltas.

Las primeras dificultades encontradas fue familiarizarme con una técnica heurística que nunca antes había utilizado, como son los Algoritmos Basados en Cúmulos de Partículas. Además, he necesitado invertir bastante tiempo en el estudio y la búsqueda de nuevas variantes para codificación binaria y para permutaciones de enteros, ya que, debido a la relativa novedad de este tipo de algoritmos, todavía no se han obtenido buenos modelos de estos algoritmos.

La siguiente dificultad fue el estudio y comprensión de los problemas, además de su implementación en el esqueleto de código. En el caso de LA, en primer lugar implementé el problema basándome en la configuración encontrada en varios artículos citados anteriormente y utilizando instancias contenidas en estos documentos. Posteriormente, recibimos nuevas instancias con nuevos resultados obtenidos a partir de una nueva configuración del problema. Esto supuso la realización de pequeñas modificaciones en la implementación del problema y nuevos experimentos, con la intención de poder comparar los resultados obtenidos. Por otra parte, respecto al problema GOMAD, lo más complicado fue el estudio de la naturaleza del problema, ya que trata de experimentos de genética y técnicas de biomedicina, disciplinas de las cuales no tenía conocimientos. Incluso la obtención e interpretación de las instancias fue bastante complicada, pues requerían el conocimiento de una serie de términos y conceptos desconocidos para mí.

La tercera dificultad fue que como base del proyecto debía utilizar código desarrollado por otras personas, lo que implica cierta dificultad ya que no sólo hay que familiarizarse con un código no desarrollado por ti mismo (que ya tiene su dificultad) sino que hay que conocerlo hasta el mínimo detalle para realizar modificaciones y ampliaciones sobre él.

Además de esos problemas, surgieron otros problemas pero ya mucho más sencillos, como la necesidad de instalar y configurar nuevo software, incompatibilidades entre software instalado, . . .

6.3. Extensiones Futuras

Entre las posibles mejoras a este proyecto, una de las más importantes es la de realizar más pruebas para poder afinar más la configuración del esqueleto y obtener mejores resultados. Esto resultaría especialmente importante en el caso del problema LA que es admite mejoras claramente. En este sentido, una buena extensión sería desarrollar una nueva versión del PSO binario más afinada que la actual, es decir, en la que la pérdida de información en la operación de movimiento sea mínima.

Respecto al problema GOMAD, podría evaluarse el esqueleto con muchas más instancias reales de microarrays y realizar un estudio más extenso de los resultados. Además, la incorporación de una nueva clase (o método) para la realización de clustering jerárquico a partir de la ordenación de genes resultado proporcionaría una funcionalidad añadida a la implementación de este problema. No obstante, esta posible extensión sería muy específica del problema y supondría una extensión aparte del modelo del esqueleto PSO.

Por otra parte, aunque se ha realizado una versión del esqueleto PSO para la resolución de problemas de codificación continua, no se han descrito en esta memoria problemas que utilicen este tipo de codificación, pues la gran mayoría de los esfuerzos se han centrado en la resolución de los problemas LA y GOMAD que utilizan otro tipo de codificación. Una probable extensión será la de codificar problemas de naturaleza continua y evaluar el esqueleto en esta versión.

Para finalizar, es interesante la evaluación del esqueleto en una red WAN. Puesto que las clases para la ejecución paralela en WAN se han implementado, y se han probado en una red LAN, sería interesante probarlo en redes de área extensa como se ha realizado en proyectos similares relacionados con la biblioteca MALLBA. Aunque para esto se necesitaría establecer la infraestructura de comunicaciones y solventar problemas de acceso y seguridad, aspectos no triviales que justifican un estudio aparte.

Apéndices

En los siguiente apéndices, se describen y se comentarán algunos aspectos no definidos a lo largo de la memoria y que pese a eso es conveniente que sean comentados aunque sea brevemente para comprender mejor el documento o para ampliar el conocimiento de detalles que sólo fueron mencionados sin profundizar en ellos.

El primero de los apéndices describe el entorno del que se dispuso para la realización del proyecto, incluyendo tanto los elementos *Hardware* como las herramientas *Software* utilizadas. Se describirán así mismo ciertas pruebas realizadas sobre los elementos *Hardware* para comprobar su eficiencia.

En el segundo apéndice se comentará en detalle la descripción de la biblioteca `NetStream`, que ha sido bastante importante a la hora del desarrollo del proyecto debido a que es la biblioteca que se ocupa de las comunicaciones. También se describirán aunque en menor detalle otras bibliotecas que se han utilizado en los esqueletos de código.

Terminamos la memoria con un breve manual, que ayudará al usuario que desee utilizar esos esqueletos a instalar este software, configurarlo y comentará cómo ejecutar los problemas desarrollados y como desarrollar otros problemas utilizando estos esqueletos.

Apéndice A

Entorno

En este apéndice se comentará el estado de los elementos *hardware* y *software*, de los que se disponen para el desarrollo del proyecto. También se discutirán los problemas que se han tenido para la configuración y utilización de dichas herramientas. Por último se describirá las pruebas realizadas para comprobar la eficiencia del sistema y los resultados obtenidos.

A.1. Material Disponible

Durante el desarrollo de este proyecto y para la realización de las pruebas se ha dispuesto de una serie de máquinas repartidas en el cluster **IX-cluster** del Departamento de Lenguajes y Ciencias de la computación de la Universidad de Málaga.

Este cluster de PCs está compuesto por 16 ordenadores equipados con procesadores Pentium IV y con sistema operativo Linux (distribución Suse 9.0 con kernel 2.4.19). Las características completas se pueden ver en la Tabla A.1.

Procesador	Pentium IV, 1.6 Ghz
Memoria Principal	512 MB
Disco Duro	60 GB
Adaptador de Red	FastEthernet 10/100 Mbps

Tabla A.1: Características de las máquinas del cluster IX-cluster

El acceso al cluster se realiza a través de la máquina ix0 que es el servidor de NFS y NIS, del cual se importan las cuentas de usuarios y algunos programas. Esta máquina tiene la dirección IP 150.214.214.25 mediante la que se accede desde el exterior. Internamente, ix0 tiene la dirección IP 192.168.3.254 y las demás máquinas comparten el mismo subrango de direcciones de la red LAN (véase la Tabla A.2).

Dirección IP	Nombre de la Máquina
192.168.3.1	ix1.lcc.uma.es
192.168.3.2	ix2.lcc.uma.es
192.168.3.3	ix3.lcc.uma.es
192.168.3.4	ix4.lcc.uma.es
192.168.3.5	ix5.lcc.uma.es
192.168.3.6	ix6.lcc.uma.es
192.168.3.7	ix7.lcc.uma.es
192.168.3.8	ix8.lcc.uma.es
192.168.3.9	ix9.lcc.uma.es
192.168.3.10	ix10.lcc.uma.es
192.168.3.11	ix11.lcc.uma.es
192.168.3.12	ix12.lcc.uma.es
192.168.3.13	ix13.lcc.uma.es
192.168.3.14	ix14.lcc.uma.es
192.168.3.15	ix15.lcc.uma.es
192.168.3.16	ix16.lcc.uma.es

Tabla A.2: Direcciones de las máquinas de IX-cluster

En general, las máquinas cuentan con todo el software necesario para el desarrollo del proyecto, destacando las ultimas vesiones[1] del compilador gcc de C++ y de la biblioteca MPI (implementación MPICH versión 1.2.2.3).

A.2. Comunicaciones

Las comunicaciones que podemos establecer entre los diferentes máquinas comentadas en la sección anterior se realizan mediante *ssh*. A continuación se describe la biblioteca de comunicaciones utilizada.

A.3. Biblioteca de Comunicaciones

Para las comunicaciones se está utilizando la biblioteca *NetStream* (para mayor información consultar el Apéndice B) en su última versión (actualmente la 1.6). Esta biblioteca es una interfaz orientada a objetos sobre la biblioteca estándar de paso de mensajes *MPI* que facilita su uso de manera considerable.

Como implementación de la MPI se ha optado por *MPICH*, en su última versión disponible (versión 1.2.2.3) y configurada para que la inicialización de los procesos la haga mediante *ssh*.

Para lanzar una aplicación MPI, se necesita una máquina que pueda acceder mediante *ssh* a todas las demás sobre las que quiere ejecutar el programa. En la conexión de las máquinas de **IX-cluster**, la identificación *ssh* se realiza mediante encriptación con clave pública.

[1] cuando se escribió este documento

Apéndice B

Otras Bibliotecas

Para el desarrollo de los esqueletos de código, así como para la instanciación de éstos con los problemas comentados en el Capítulo 4, se ha utilizado una serie de bibliotecas para el manejo de listas, vectores, matrices, ... con el fin de facilitar la compresión y modularidad del código. Estas bibliotecas no son muy extensas, porque sólo se ha pretendido satisfacer las funciones requeridas en este proyecto, aunque están abiertas a futuras ampliaciones en el caso de que se deseen utilizar en otras aplicaciones.

B.1. La Biblioteca NetStream

Esta biblioteca proporciona una implementación en C++ de una clase denominada `NetStream` [2], que proporciona servicios para el intercambio de información a través de una red, independientemente de que sea *LAN* o *WAN*. Algunos de estos servicios han sido incorporados en las implementaciones de las versiones paralelas de los esqueletos desarrollados en el proyecto. Debido a su gran influencia sobre las versiones paralelas de los esqueletos esta clase merece una atención especial.

Los servicios incorporados en la biblioteca han sido desarrollados para ser utilizados tanto en una red de área local (*LAN*) como en un de área extensa (*WAN*), proporcionando un mecanismo eficaz de paso de mensajes útil para ser incorporados en una gran variedad de programas paralelos que se ejecuten sobre estos dos tipos de redes. El paso de mensajes es un conocido mecanismo de comunicación muy útil en una gran cantidad de aplicaciones. Las dos bibliotecas más populares que se clasifican en el paradigma de paso de mensajes en C++ son *MPI* y *PVM*. Sin embargo, estas bibliotecas trabajan a un nivel bajo, por lo que requiere que el usuario que las utilice sea experto en programación paralela. Precisamente para solventar este problema ha sido desarrollada esta biblioteca, que apoyándose en la biblioteca *MPI* [1], proporciona una serie de servicios de fácil manejo para el paso de mensajes.

Estos servicios pueden clasificarse en dos tipos, servicios **básicos** y **avanzados**, lo que hace que la interfaz proporcionada por la biblioteca sea apropiada tanto para usuarios no especializados como para expertos en el desarrollo de programas paralelos.

Las principales ventajas que han llevado a utilizar esta biblioteca en el proyecto son:

- Proporciona una interfaz sencilla para aquellos usuarios que no son expertos en programación paralela. Los servicios se implementan en esta biblioteca mediante métodos de la clase `NetStream`, estos métodos se caracterizan por proporcionar una clara interfaz para que el tiempo necesario para aprender a utilizarlos sea el mínimo.

- Permite el acceso tanto de redes de área local como de área extensa, presentando una interfaz uniforme para ambos entornos.

- La interfaz que se proporciona es flexible y está orientada a objetos, con lo que se consigue separar la implementación de los servicios conceptuales.

- El envío de objetos a través de la red es eficiente, lo que es muy importante, ya que la biblioteca será utilizada para el envío de mensajes de forma intensiva a través de la red. En [27], se muestra un breve estudio de la pérdida de eficiencia de la biblioteca respecto a *MPI*, y muestra que la pérdida es mínima.

La forma de utilizar esta biblioteca para la comunicación a través de una red consistirá en la definición de un objeto `NetStream` y posteriormente ir invocando los métodos apropiados de la clase para llevar a cabo la comunicación mediante los servicios prestados por estos. A la hora de utilizar la biblioteca cabe destacar la importancia de los operadores de inserción (<<) y extracción (>>) de la clase, que sirven para expresar la recepción y envío de información sobre la red representada por el objeto `NetStream`. Al utilizar estos operadores se consigue una uniformidad de la biblioteca a la hora del envío y recepción de datos, además permite introducir y extraer una secuencia de más de un objeto o dato en la red en un sola sentencia, reduciéndose así la complejidad del código resultante. Por ejemplo:

```
NetStream ns; ... ns << 9 << 'a' << "hello world"; ...
```

Los servicios básicos ofertados por esta clase se pueden observar en la Tabla B.1. Con esos métodos el usuario no especializado dispone de una manera simple de enviar y recibir información a través de la red.

Los métodos que empiezan con el carácter subrayado "_", han sido nombrados así debido a que con el mismo nombre existe un manipulador que

Método	Descripción
init	Inicializa el sistema de comunicación.
finalize	Finaliza el sistema de comunicación.
operator<<	Permite el envío de mensajes
operator>>	Permite la recepción de mensajes
_set_target	Indica a que proceso se dirigirán los mensajes.
_get_target	Obtiene el proceso al que se dirigen los mensajes.
_set_source	Indica de que proceso se reciben los mensajes.
_get_source	Obtiene de que proceso se están recibiendo los mensajes.
_pnumber	Obtiene el número de procesos que se están ejecutando en paralelo.
_my_pid	Obtiene el identificador del proceso.
wait	Permite esperar hasta la llegada de un mensaje

Tabla B.1: Servicios básicos ofrecidos por la clase `NetStream`

permite llamar estos servicios dentro de los operadores de inserción (`<<`) y extracción (`>>`).

Además de esos servicios básicos se disponen de servicios avanzados, destinados a usuarios más especializados en la creación de programas paralelos. Entre estos servicios se tienen métodos de sincronización global (`barrier`), métodos para la difusión de mensajes a todos los procesos (`broadcast`) y métodos de chequeo no bloqueantes de la cola de entrada (`probe`). También incorpora un servicio de envío de datos empaquetados, que permite un aprovechamiento más eficiente de los recursos de la red. El envío-recepción de datos empaquetados se hace mediante los manipuladores `pack_begin` y `pack_end`. Un ejemplo de envío y recepción de datos empaquetados se muestra a continuación:

```
NetStream ns; ... ns << pack_begin; ns << 1 << 6.3 << "mensaje";
ns << pack_end; ... ns << pack_begin; ns >> entero >> real >>
cadena; ns << pack_end; ...
```

B.2. Otras Bibliotecas

Aparte de biblioteca de comunicaciones definida anteriormente, se han utilizado y desarrollado otras bibliotecas:

- `Rlist`: Esta biblioteca se utiliza para el manejo de listas dinámicas. Esta biblioteca dispone de dos clases: `Rlist`, que representa la lista en sí mismo y dispone de todos los métodos necesarios para crearla, añadirle elementos, consultarla y borrarla y la clase `Rlist_item`, que es una clase auxiliar que ayuda en el manejo de cada uno de los elementos que conforman la lista.

- `Rarray`: Esta biblioteca se utiliza para el manejo de arreglos dinámicos, ofreciendo para ello la clase paramétrica `Rarray`, que dispone de todos los métodos necesario para utilizarlo, destacando el `operator[]`, que permite acceder y modificar cada elemento del arreglo directamente.

- `Matrix`: Esta biblioteca facilita el uso de matrices. Para ello facilita la clase paramétrica `Matrix`, que permite realizar las operaciones típicas sobre matrices como suma, resta, multiplicación, realizar la traspuesta, ...También destaca la inclusión del `operator[]`, que permite acceder y modificar cada elemento del matriz.

- `Time`: Esta biblioteca ofrece funciones que permiten calcular el tiempo que se tarda en ejecutar cierto código. Las funciones son `_used_time()` y `_used_time(float time)`. La segunda de ellas devuelve el tiempo transcurrido desde el tiempo indicado en el parámetro que debe ser el resultado obtenido de llamar a la primera función.

- `Random`: Esta biblioteca ofrece funciones para la generación de números aleatorios, además de otras de utilidad en los esqueletos. Esas funciones se muestran en la Tabla B.2.

Función	**Descripción**
undefined()	Devuelve un valor indefinido.
infinity()	Devuelve el valor infinito.
rand01()	Devuelve un número aleatorio en el rango [0,1].
rand_int(min,max)	Devuelve un número aleatorio en el rango [min,max].
random_seed(seed)	Selecciona la semilla para generación de números aleatorios.

Tabla B.2: Funciones de la Biblioteca `Random`

Apéndice C

Manual de Usuario

En este apéndice se proporciona una guía útil sobre la instalación, configuración y ejecución de los esqueletos de código desarrollados, así como las instanciaciones hechas de los mismos con los problemas comentados en el Capítulo 4.

C.1. Instalación y Configuración

Los esqueletos de código han sido desarrollados en un entorno que utiliza como sistema operativo Unix/Linux, por lo que para sus funcionamiento debe ser instalados en máquinas que dispongan de ese tipo de sistema operativo. En el CD que se adjunta en esta documentación se encuentra una versión comprimida de los esqueletos. Además, en el sitio Web de MALLBA con URL `http://neo.lcc.uma.es/mallba/easy-mallba/index.html` se dispone de una versión actualizada. Concretamente los esqueletos y problemas desarrollados a lo largo del proyecto se encuentran en el fichero `mallba.tar.gz`. Para instalarlo, lo primero que se debe hacer es copiar dicho fichero al directorio donde deseamos instalar las bibliotecas. Una vez copiado ya se puede descomprimir. Por ejemplo si el usuario *user*, deseara instalar estas bibliotecas al directorio `Programs` dentro de su directorio local debería realizar las siguientes acciones:

```
$ cp /mnt/cdrom/mallba.tar.gz ~/Programs $ cd ~/Programs $ gzip -d
mallba.tar.gz $ tar -xvf mallba.tar
```

Tras descomprimir el archivo dentro del directorio elegido se habrá creado un directorio `Mallba` dentro del directorio donde instalamos la Biblioteca, cuyo contenido debe ser:

`Makefile`	*Fichero*	382 B	Fichero para crear la Biblioteca
`environment`	*Fichero*	347 B	Variables de entorno
`inc`	*Directorio*	116 B	Enlaces establecidos
`lib`	*Directorio*	83 KB	Biblioteca Mallba (`libmalllba.a`)
`rep`	*Directorio*	43.5 MB	Repositorio de los esqueletos de código
`src`	*Directorio*	127 KB	Fuentes de las Bibliotecas auxiliares.
`ProblemInstances`	*Directorio*	1.3 MB	Instancias para los problemas.

Una vez descomprimida la Biblioteca, pasamos a su configuración, para ello será necesario tener instalada la biblioteca de paso de mensajes *MPI*, en concreto su implementación *MPICH*, que se puede encontrar para descargarse gratuitamente en `http://www-unix.mcs.anl.gov/mpi/mpich/`. En el resto del manual se supondrá que dicha biblioteca está configurada utilizando para las comunicaciones el método `ch_p4` (que es más común de todos los que posee). Si estuviese configurado con otro método quizás hubiera que cambiar la forma en que se lanza la ejecución de los problemas.

La configuración de biblioteca consiste en modificar el fichero `environment` que contiene las variables de entorno necesarias para permitir la utilización de la Biblioteca. Las variables de entorno definidas en dicho fichero son:

`MALLBA_DIR`= ruta donde se encuentra el directorio `Mallba`
`MALLBA_INC`= ruta donde se encuentra el directorio `inc` en `Mallba`
`MALLBA_LIB`= ruta donde se encuentra el directorio `lib` en `Mallba`
`MALLBA_SRC`= ruta donde se encuentra el directorio `src` en `Mallba`
`MALLBA_REP`= ruta donde se encuentra el directorio `rep` en `Mallba`
`MPI_BIN`= ruta donde se encuentran los ejecutables de *MPICH*

Aparte de esas variables, en el fichero se definen otras que dependen de las anteriores:

```
CXX=${MPI_BIN}/mpiCC
RUN=${MPI_BIN}/mpirun
CPPFLAGS=-I/${MALLBA_INC}
LDFLAGS=-L/${MALLBA_LIB}
LOADLIBES=-lmallba
```

Por último, para crear los ejecutables y las bibliotecas correspondientes lo único que faltaría será ejecutar la siguiente orden:

```
$ make all
```

Esa instrucción se encargará de construir todos los ficheros necesarios para poder utilizar todos los esqueletos y los problemas.

En caso que se desee cambiar la configuración, se eliminarían los ficheros creados con la configuración anterior, para ello te utilizaría la orden:

```
$ make clean
```

Luego se modifica la configuración y finalmente se volvería realizar la orden `$ make all`.

C.2. Ejecución de los Esqueletos de Código

El esqueleto desarrollado en el proyecto se encuentran contenido en el directorio `/rep`, de forma que los ficheros que componen cada uno de ellos se agrupan en directorios con el nombre del esqueleto. Así, dentro del directorio `/rep` nos encontramos con los siguientes subdirectorios:

- `/PSO`: ficheros del esqueleto de código *PSO* (que implementa el algoritmo Particle Swarm Optimization).
- `/SA`: ficheros del esqueleto de código *SA* (que implementa el algoritmo Recocido Simulado).
- `/newGA`: ficheros del esqueleto de código *newGA* (que implementa el algoritmo Algoritmo Genético).
- `/CHC`: ficheros del esqueleto de código *CHC* (que implementa el algoritmo CHC).
- `/ES`: ficheros del esqueleto de código *ES* (que implementa el algoritmo Estrategias Evolutivas).
- `/hybrids/CHC+ES`: ficheros del esqueleto de código *CHCES* (que implementa el algoritmo híbrido entre el algoritmo CHC y las Estrategias Evolutivas).
- `/hybrids/newGA+SA`: ficheros del esqueleto de código *newGASA* (que implementa el algoritmo híbrido entre el Algoritmo Genético y el Recocido Simulado).

Cada directorio de los anteriores contiene los ficheros necesarios para instanciarlo con un problema y una serie de directorios que representan los problemas que han sido resueltos utilizando el correspondiente esqueleto.

Una vez dentro del directorio del problema que queramos ejecutar, ya podemos lanzar las pruebas que deseemos.

Para lanzar las pruebas secuenciales se utilizaría la siguiente orden:

```
$ MainSeq <esqueleto>.cfg <fichero_problema> <fichero_resultado>
```

donde `<esqueleto>.cfg` es el fichero de la configuración correspondiente al esqueleto con el que se está intentando resolver el problema. `<fichero_problema>` representa el archivo donde se encuentra la instancia del problema que deseamos resolver y `<fichero_resultado>` indica el fichero donde deseamos que se escriban los resultados de la ejecución. En el fichero `Makefile` existe una etiqueta `SEQ` donde hay una línea similar a la vista anteriormente, si se desea, se puede modificar para que utilice los ficheros que queremos con lo que la ejecución secuencial se limitaría a ejecutar la orden:

```
$ make SEQ
```

Para lanzar las pruebas paralelas (sin importar si se trata de LAN o WAN) se utilizaría la siguiente orden:

```
$ mpirun -p4pg pgfile MainLan
```

donde `mpirun` es una herramienta de *MPICH* que permite lanzar ejecuciones paralelas, el `-p4pg pgfile` es una opción de dicha herramienta que permite cargar un fichero donde viene la configuración de las máquinas donde se desea ejecutar. El formato de este fichero son tantas líneas como máquinas diferentes participen en la ejecución, donde cada línea debe tener el formato:

```
<maquina> <numero_procesos> <ejecutable> [<usuario>]
```

Por ejemplo:

```
mallba6 0 /home/muser2/ejecutable1 mallba1 4
/home/muser2/ejecutable2 beowulf 1 /home/usuario6/ejecutable1
usuario6
```

Con ese fichero, se lanzaría una ejecución compuesta por un proceso (en concreto el que tiene identificador 0) en `mallba6` que ejecuta el fichero `ejecutable1`, cuatro procesos en `mallba1` que ejecuta el fichero `ejecutable2` y un proceso en la máquina `beowulf` que vuelve a ejecutar el fichero `ejecutable1`, pero esta vez se entra en la cuenta correspondiente al usuario *usuario6*.

El fichero `MainLan` supone que en el directorio donde se lanza existe el fichero `Config.cfg` que debe tener el siguiente formato:

```
<esqueleto>.cfg <fichero_problema> <fichero_resultado>
```

Otra forma de ejecutarlo puede ser realizando `$ make LAN`

Además, es necesario mencionar que como los algoritmos híbridos disponen de tres versiones secuenciales y otras tres versiones paralelas, se proporcionan al usuarios varios ficheros `MainSeq` y `MainLan`, más concretamente se tienen los ficheros:

`MainSeq1 MainLan1`	Ejecución de la hibridación fuerte
`MainSeq2 MainLan2`	Ejecución de la hibridación débil 1
`MainSeq3 MainLan3`	Ejecución de la hibridación débil 2

A parte de la variación de los nombres de los ejecutables, la ejecución de los algoritmos híbridos es igual al de los algoritmos normales anteriores.

C.3. Nuevas Instanciaciones de los Esqueletos

Para llevar a cabo nuevas instanciaciones para resolver un problema mediante uno de los esqueletos desarrollados en el proyecto se deben seguir los siguientes pasos:

1. Nos situamos en el directorio correspondiente al esqueleto con el que queremos resolver el problema. Por ejemplo:

   ```
   $ cd ~/Mallba/rep/PSO
   ```

2. Creamos un directorio donde se situará la instanciación correspondiente al problema y nos situamos dentro de él. Por ejemplo:

   ```
   $ mkdir Problem $ cd Problem
   ```

3. Copiamos los fichero del esqueleto con la orden:

   ```
   $ cp ../*.hh ../*.cc ../*.cfg ../Makefile .
   ```

4. Rellenamos con la implementación concreta del problema el fichero con extensión `.req.cc` (quizás se tenga que modificar el fichero `.hh` también con los nuevos métodos que se añadan).

5. Compilamos para crear el ejecutable con:

   ```
   $ make all
   ```

6. Rellenamos la configuración correspondiente al esqueleto y ya se puede ejecutar como se mostró en el Apartado anterior.

Si se cambia la implementación de un problema ya compilado, antes de volver a compilar se debe borrar los ficheros de la compilación anterior con:

```
$ make clean
```

En caso de error en la compilación o ejecución, se debe corregir y realizar las ordenes `make` anteriores. Finalmente, con el código libre de errores tanto en la compilación como en la ejecución ya es posible ejecutar los esqueletos tal y como se comentó en la sección anterior.

Índice de tablas

Índice de figuras

Bibliografía

[1] Al-Tawil, Khalid, and Moritz. Perfomance Modeling and Evaluation of MPI. 1999.

[2] E. Alba. Netstream: A Flexible and Simple Message Passing Service for LAN/WAN Utilization. *E.T.S.I.I. Málaga. Departamento de Lenguajes y Ciencias de la Computación*, 2001.

[3] E. Alba. *Parallel Metaheuristics: A New Class of Algorithms.* John Wiley & Sons, October 2005.

[4] E. Alba and MALLBA group. Mallba: A Library of Skeletons for Combinatorial Optimisation. In *Proceedings of the Euro-Par*, number LNCS 2400, pages 927–932, Paderborn (GE), 2002. B. Monien and R. Feldmann.

[5] N. Bar and I. Kessler. Tracking Mobile Users in Wireless Communications Networks. *IEEE Trans. Information Theory*, 39:1877–1886, 1993.

[6] T. Bäck, D. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation.* IOP Publishing and Oxford University Press, New York and Bristol (UK), Feb 1997.

[7] C. Blum and A. Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *El Lenguaje Unificado de Modelado.* Addison Wesley, 1999.

[9] P. Brown and D. Botstein. Exploring the New World of the Genome with DNA Microarrays. *Nature Genetics*, 21:33–37, 1999.

[10] M. Clerc. *Discrete Particle Swarm Optimization. Illustrated by the Traveling Salesman Problem.* In URL `http://clerc.maurice.free.fr/pso/pso_tsp/Discrete_PSO_TSP.htm`, Apr 2003.

[11] M. Clerc. Binary Particle Swarm Optimisers: Toolbox, Derivations, and Mathematical Insights. In URL `http://clerc.maurice.free.fr/pso/`, Feb 2005.

[12] C. Cotta, A. Mendes, V. Garcia, P. França, and P. Moscato. Applying Memetic Algorithms to the Analysis of Microarray Data. In *Proceedings of the Applications of Evolutionary Computing*, volume 2611 of *Lecture Notes in Computer Science*, pages 22–32, Berlin, 2003. Springer-Verlag.

[13] T. Crainic and M. Toulouse. *Handbook of Metaheuristics*, chapter Parallel Strategies for Metaheuristics, pages 475–513. Kluwer Academic Publishers, 2003.

[14] J. DeRisi, V. Iyer, and P. Brown. Exploring the Metabolic and Genetic Control of Gene Expression on a Genomic Scale. *Science*, 278(5338):680–686, Oct 1994.

[15] M. Dorigo. *Optimization, Learning and Natural Algorithms*. PhD thesis, Dipartamento di Elettronica, Politecnico di Milano, 1992.

[16] M. Dorigo. The Ant Colony Optimization Metaheuristic: Algorithms, Applications and Advances. Technical Report IRIDIA-2000-32, Universit'e Libre de Bruxelles, IRIDIA, 2000.

[17] R. Eberhart and Y. Shi. Comparing Inertia Weights and Constriction Factors in Particle Swarm Optimization. In *Proceedings of the International Congress on Evolutionary Computation*, volume 1, pages 84–88, July 2000.

[18] M. Eisen, P. Spellman P. Brown, and D. Botstein. Cluster Analysis and Display of Genome-Wide Expression Patterns. In *Proceedings of the National Academy of Sciences of the USA*, volume 95, pages 14863–14868, 1998.

[19] Iyer et al. The Transcriptional Program in the Response of Human Fibroblasts to Serum. *Science*, 283:83–87, 1999.

[20] T. Feo and M. Resende. Greedy Randomized Adaptive Search Procedures. *Journal of Global Optmization*, 6:109–133, 1999.

[21] L. Fogel, J. Owens, and M. Walsh. *Artificial Intelligence Through Simulated Evolution*. 1966.

[22] F. Glover. Heuristics for Integer Programming Using Surrogate Constraints. *Decision Sciences*, 8:156–166, 1977.

[23] F. Glover. Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*, 13:533–549, 1986.

[24] F. Glover and G. Kochenberger. *Handbook of Metaheuristics*. Kluwer Academic Publishers, Norwell, MA, 2002.

[25] P. Gondim. Genetic Algorithms and the Location Area Partitioning Problem in Cellular Networks. In *Proceedings of IEEE the 46th Vehicular Technology Conference*, pages 1835–1841, 1996.

[26] G. Gudise and G. Venayagamoorthy. Comparison of Particle Swarm Optimization and Backpropagation as Training Algorithms for Neural Networks. In *Proceedings of the IEEE Swarm Intelligence Symposium 2003*, pages 110–117, Indianapolis, Indiana, USA, 2003.

[27] F. Guerrero. *Algoritmos para el Problema de la Asignación de Frecuencias a Enlaces de Teléfonos Móviles*. Universidad de Málaga, 2001.

[28] J. Holland. *Adaptation in Natural and Artificial Systems*. The MIT Press, Cambridge, Massachusetts, first edition, 1975.

[29] R. Jenner, M. Alba, C. Boshoff, and P. Kellam. Kaposi's Sarcoma-Associated Herpesvirus Latent and Lytic Gene Expression as Revealed by DNA Arrays. *Journal of Virology*, 2:891–902, 2001.

[30] J. Kennedy. The Particle Swarm: Social Adaptation of Knowledge. *IEEE International Conference on Evolutionary Computation*, pages 303–308, Apr 1997.

[31] J. Kennedy and R. Eberhart. Particle Swarm Optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, Perth, Australia, Nov 1995.

[32] J. Kennedy and R. Eberhart. A Discrete Binary Version of the Particle Swarm Algorithm. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, volume 5, pages 4104–4109, 1997.

[33] J. Kennedy, R. Eberhart, and Y. Shi. *Swarm Intelligence*. San Francisco: Morgan Kaufmann Publishers, 2001.

[34] J. Kennedy and R. Mendes. Neighborhood Topologies in Fully Informed and Best-of-Neighborhood Particle Swarms. *Man and Cybernetics, Part C, IEEE Transactions on Systems*, 36:515– 519, July 2006.

[35] S. Kirkpatrick, C. Gelatt, and M. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

[36] E. Koonin. The Emerging Paradigm and Open Problems in Comparative Genomics. *Bioinformatics*, 15:265–266, 1999.

[37] A. Mendes, C. Cotta, and V. Garcia. Gene Ordering in Microarray Data Using Parallel Memetic Algorithms. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 604–611, Washington, DC, USA, 2005.

[38] N. Mladenovic and P. Hansen. Variable Neighborhood Search. *Computers Oper. Res*, 24:1097–1100, 1997.

[39] M. Omran, A. Salman, and A. Engelbrecht. Image Classification Using Particle Swarm Optimization. In *Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning 2002*, pages 370–374, Singapore, 2002.

[40] S. Oualline. *Practical C++ Programming.* O'Reilly, 2nd edition, Dec 2002.

[41] C. Ourique, E. Biscaia, and J. Pinto. The Use of Particle Swarm Optimization for Dynamical Analysis in Chemical Processes. In *Proceedings of the Computers and Chemical Engineering*, volume 26, pages 1783–1793, 2002.

[42] K. Parsopoulos, E. Papageorgiou, P. Groumpos, and M. Vrahatis. A First Study of Fuzzy Cognitive Maps Learning Using Particle Swarm Optimization. In *Proceedings of the IEEE Congress on Evolutionary Computation 2003*, pages 1440–1447, Canbella, Australia, 2003.

[43] G. Polya. *How to Solve It. The New Aspect of Mathematical Method.* Princeton University Press, 1971.

[44] I. Rechenberg. *Evolutionsstrategie: Optimierung Technischer Systeme Nach Prinzipien der Biologischen Evolution.* Fromman-Holzboog Verlag, Stuttgart, 1973.

[45] C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* Blackwell Scientific Publishing, Oxford, UK, 1993.

[46] B. Secrest and G. Lamont. Communication in Particle Swarm Optimization Illustrated by the Traveling Salesman Problem. In *Proceedings of the Workshop on Particle Swarm Optimization 2001*, Indianapolis, 2001.

[47] O. Stephan and A. Zomaya. *Handbook Of Bioinspired Algorithms And Applications.* CHAPMAN and HALL/CRC, 2005.

[48] R. Subatra and A. Zomaya. A Comparison of Three Artificial Life Techniques for Reporting Cell Planning in Mobile Computing. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):142–153, Feb 2003.

[49] R. Subrata and A. Zomaya. Location Management in Mobile Computing. *aiccsa*, 00:0287, 2001.

[50] Riky Subrata and Albert Y. Zomaya. Evolving Cellular Automata for Location Management in Mobile Computing Networks. *IEEE Trans. Parallel Distrib. Syst.*, 14(1):13–26, 2003.

[51] T. Stützle. Local Search Algorithms for Combinatorial Problems Analysis, Algorithms and New Applications. Technical report, DISKI Dissertationen zur Künstliken Intelligenz. Sankt Augustin, Germany, 1999.

[52] J. Taheri and A. Zomaya. The Use of a Hopfield Neural Network in Solving the Mobility Management Problem. *icps*, 00:141–150, 2004.

[53] Javid Taheri and Albert Y. Zomaya. A Simulated Annealing Approach for Mobile Location Management. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium - Workshop 6*, page 194, Washington, DC, USA, 2005. IEEE Computer Society.

[54] X. Xie, W. Zhang Z., and Yang. Solving Numerical Optimization Problems by Simulating Particulates in Potential Field with Cooperative Agents. In *International Conference on Artificial Intelligence*, Las Vegas, NV, USA, 2002.