

Una Versión de ACO para Problemas con Grafos de muy Gran Extensión

Enrique Alba y Francisco Chicano

Resumen— Los Algoritmos de Optimización basados en Colonias de Hormigas (ACOs) se han aplicado con éxito a problemas de optimización combinatoria que pueden transformarse en una búsqueda dentro de un grafo. Las hormigas artificiales construyen la solución paso a paso añadiendo componentes que se representan mediante nodos del grafo. Este modelo es adecuado cuando el grafo no es muy grande (miles de nodos) pero no es viable cuando el tamaño del grafo supone un desafío para la memoria de un ordenador y no puede generarse ni almacenarse completamente en ella. En este artículo proponemos una variante de ACO que supera los inconvenientes que surgen al trabajar con problemas cuyo grafo subyacente posee un gran tamaño. Además de dar la descripción del modelo, ilustramos su aplicación a un problema con gran interés en la comunidad software: la Verificación de Sistemas Concurrentes mediante Model Checking. De esta forma, no sólo presentamos un nuevo modelo de ACO sino que abrimos una nueva línea de investigación relacionada con la aplicación de ACOs a las técnicas formales en Ingeniería del Software.

Palabras clave— Optimización basada en Colonias de Hormigas, Model Checking

I. INTRODUCCIÓN

Los Algoritmos basados en Colonias de Hormigas (ACOs) son un tipo de metaheurística basada en población cuya filosofía está inspirada en el comportamiento de las hormigas reales cuando buscan comida [1]. La idea principal consiste en usar hormigas artificiales que simulan dicho comportamiento en un escenario también artificial: un grafo. Las hormigas artificiales se colocan en nodos iniciales de dicho grafo y lo recorren saltando de un nodo a otro con la intención de encontrar el camino más corto desde su nodo inicial hasta un nodo final u objetivo. Cada hormiga avanza de forma independiente a las demás, pero la decisión del siguiente nodo a visitar depende de ciertos valores numéricos asociados a los arcos o nodos del grafo. Estos valores modelan los *rastros de feromona* que depositan las hormigas reales cuando caminan. Las hormigas artificiales alteran (al igual que las hormigas reales) los rastros de feromona del camino trazado, de modo que el avance de una puede influir en el camino de otra. De esta forma se incorpora a los ACOs un mecanismo de cooperación indirecta entre las distintas hormigas simuladas, que constituye un factor clave en la búsqueda [2].

Una forma de resolver un problema con esta técnica consiste en transformarlo en una búsqueda de caminos mínimos en grafos. En algunos casos esto se puede hacer de forma natural, como ocurre en

el problema del viajante de comercio (TSP), que fue el primero al que se le aplicó un ACO [3]. En otros problemas, como el entrenamiento de redes neuronales [4], la transformación no es tan directa. En general, los problemas a los que se pueden aplicar los ACOs son aquéllos cuyas soluciones tentativas pueden representarse mediante una secuencia de *componentes* (véase la Sección II). Los modelos de ACO existentes en la literatura para problemas de optimización combinatoria trabajan sobre grafos con un número de nodos conocido y suficientemente pequeño como para poder almacenar en memoria los rastros de feromona asociados al grafo. Estos modelos asumen además que el número de componentes de que consta una solución es siempre el mismo o se encuentra acotado por un valor conocido. Es decir, el camino recorrido por las hormigas tiene una longitud máxima conocida. Estas suposiciones que hacen los modelos tradicionales de ACO representan una limitación para el conjunto de problemas susceptibles de ser resueltos con ellos. De hecho, existen problemas que pueden plantearse como una búsqueda de caminos mínimos en grafos y a los que, sin embargo, no se les puede aplicar un ACO de los existentes en la literatura. El motivo es que dichos problemas tienen asociados grafos subyacentes de extensión desconocida a priori y/o cuyas soluciones son caminos en el grafo de los que no se conoce ninguna cota superior razonablemente pequeña.

Un ejemplo de este tipo de problemas es la refutación de propiedades de seguridad en sistemas concurrentes. Dejando de lado los detalles (que se verán en la Sección IV) podemos decir que este es un problema de búsqueda en grafos cuyo fin es encontrar un camino entre el nodo inicial del grafo y algún nodo que cumpla cierta condición (nodo objetivo). El tamaño del grafo depende del sistema concurrente que se está verificando y, generalmente, crece de forma exponencial con respecto al tamaño del sistema. Además, no es habitual que pueda almacenarse el grafo completo en memoria y no se conoce de antemano el número de nodos que posee. Puesto que tampoco se sabe dónde se pueden encontrar los nodos objetivos, no se puede estimar una cota superior para la longitud de las soluciones que sea de utilidad¹. Para resolver este problema se han em-

¹En realidad, puede establecerse una cota superior para el número de nodos del grafo, a saber: el producto de las cardinalidades de los dominios de todas las variables usadas en el sistema concurrente. Esto es también una cota superior para la longitud de las soluciones, pero normalmente estará muy alejada de la cota superior mínima y no tendrá utilidad.

pleado algoritmos deterministas clásicos de búsqueda en grafos tales como la búsqueda en profundidad, la búsqueda en anchura o el algoritmo A*. Estos algoritmos tienen el inconveniente de que requieren mucha memoria y puede pasar mucho tiempo antes de que encuentren una traza de error en un sistema de gran tamaño. En estas situaciones las técnicas metaheurísticas han demostrado ser muy eficaces encontrando soluciones de buena calidad en un tiempo razonable. Los ACOs son las metaheurísticas que de forma natural pueden aplicarse a este problema. Sin embargo, los modelos de ACO que podemos encontrar en la literatura adolecen de las restricciones arriba mencionadas, lo cual hace que sea imposible su aplicación.

En este artículo presentamos ACOhg, un nuevo modelo de ACO que no posee las limitaciones de los modelos existentes. ACOhg puede aplicarse a problemas con grafos de tamaño desconocido y/o demasiado grandes para ser almacenados en memoria y no exige un tamaño máximo para la longitud de las soluciones. Por este motivo, puede aplicarse a un conjunto más amplio de problemas de optimización combinatoria. En particular, nuestro modelo puede aplicarse al problema de la refutación de propiedades de seguridad en sistemas concurrentes, tal y como veremos en la sección experimental. Por tanto, las aportaciones de este artículo son principalmente dos: la presentación de un nuevo modelo de ACO y su aplicación al dominio de la verificación de software. El artículo está organizado como sigue. La Sección II presenta un breve repaso de los ACOs. Nuestro modelo se detalla en la Sección III. En la Sección IV ilustramos la aplicación del modelo propuesto y analizamos la influencia de uno de sus parámetros en las soluciones. Por último, las conclusiones así como el trabajo futuro se señalan en la Sección V.

II. BREVE REPASO DE ACOs

Un problema de optimización combinatoria puede ser representado por una tripleta (S, f, Ω) , donde S es el conjunto de soluciones candidatas, f es la *función de fitness* que asigna un valor real a cada solución candidata relacionado con su calidad, y Ω es un conjunto de restricciones que debe satisfacer la solución final. El objetivo es encontrar una solución que minimice o maximice la función f , según el problema sea de minimización o de maximización (puesto que un problema de maximización puede transformarse trivialmente en uno de minimización, asumimos en lo sucesivo que el objetivo es minimizar f). En un ACO las soluciones candidatas se representan mediante una secuencia de *componentes* escogidos de un conjunto de componentes C . De hecho, una cuestión importante cuando se resuelve un problema de optimización con un ACO es la elección del conjunto de componentes C y el modo en que las soluciones se construyen usando estos componentes. En algunos problemas esta cuestión es trivial porque las

soluciones son ya una secuencia de elementos. Este es el caso del TSP donde una solución es una secuencia de ciudades. Para otros problemas, como el entrenamiento de redes neuronales, la representación no es tan directa [4].

En un ACO las hormigas artificiales construyen la solución usando un procedimiento constructivo estocástico. Durante esta fase de construcción las hormigas caminan aleatoriamente por un grafo $G = (C, L)$ llamado *grafo de construcción*, donde L es un conjunto de *conexiones* (arcos) entre los componentes (nodos) de C . En general, el grafo de construcción está completamente conectado (es completo), pero algunas de las restricciones del problema (elementos de Ω) pueden modelarse eliminando algunos arcos del grafo completo. Este es el caso del problema que usamos en el estudio experimental (Sección IV). Cada arco l_{ij} tiene asociado un rastro de feromona τ_{ij} y puede tener asociado también un valor heurístico η_{ij} . Ambos valores se usan para guiar la fase de construcción estocástica que realizan las hormigas, pero hay una diferencia importante entre ellos: los rastros de feromona se modifican a lo largo de la búsqueda mientras que los heurísticos son valores fijos establecidos por fuentes externas al algoritmo (el diseñador). Los rastros de feromona pueden asociarse también a los nodos del grafo de construcción (componentes de la solución) en lugar de a los arcos (conexiones entre componentes). Esta variación es especialmente adecuada para problemas en los que el orden de los componentes no es relevante (problemas de subconjunto [5]).

El pseudocódigo de un ACO se presenta en la Figura 1. Consta de tres procedimientos que se ejecutan durante la búsqueda: **ConstructAntsSolutions**, **UpdatePheromones**, y **DaemonActions**. En el primer procedimiento cada hormiga sigue un camino en el grafo de construcción. La hormiga comienza en un nodo inicial y elige el siguiente nodo de acuerdo con el rastro de feromona y el heurístico asociado con cada arco (o nodo). La hormiga añade el nuevo nodo al camino andado y selecciona el siguiente nodo de la misma forma. Este proceso se repite hasta que se construye una solución candidata. En el procedimiento **UpdatePheromones** se modifican los rastros de feromona asociados con los arcos. Un rastro particular puede aumentar si el arco correspondiente ha sido usado por una hormiga para construir su solución y puede disminuir debido a la evaporación (un mecanismo que evita la convergencia prematura del algoritmo). El aumento en los rastros de feromona normalmente depende de la calidad de las soluciones candidatas construidas por las hormigas que pasan por el arco. Finalmente, el último (y opcional) procedimiento **DaemonActions** lleva a cabo acciones centralizadas que no realizan las hormigas individuales. Por ejemplo, puede implementarse dentro de este procedimiento un algoritmo de optimización local para mejorar las soluciones candidatas.

```

procedure ACOMetaheuristic
  ScheduleActivities
    ConstructAntsSolutions
    UpdatePheromones
    DaemonActions // optional
  end ScheduleActivities
end procedure

```

Fig. 1. Pseudocódigo de un ACO.

III. LA NUEVA VARIANTE

El esquema anterior de ACO se puede aplicar (y se ha aplicado) a problemas con un número de nodos n de varios millares. En estos problemas el grafo de construcción tiene un número de arcos del orden de n^2 , es decir, varios millones de arcos y la matriz de feromonas requiere para su almacenamiento varios megabytes de memoria. Sin embargo, el esquema descrito no es adecuado en problemas en los que el grafo de construcción tiene como mínimo del orden de 10^6 nodos (y 10^{12} arcos). Tampoco lo es cuando la cantidad de nodos no se conoce de antemano y los nodos y arcos del grafo de construcción se generan conforme avanza la búsqueda.

El problema de refutación de propiedades de seguridad en sistemas concurrentes, que abordaremos en la Sección IV, pertenece a este tipo de problemas. En él cada nodo representa un estado del sistema y el número de nodos del grafo es realmente alto: crece de forma exponencial con respecto al tamaño del sistema. Este hecho impide verificar sistemas grandes con técnicas clásicas como búsqueda en profundidad (DFS), búsqueda en amplitud (BFS), algoritmo A^* , etc. [6]. Una solución a este problema es un camino desde el nodo inicial (el estado inicial del sistema concurrente) hasta un nodo objetivo (cuando se verifican propiedades de seguridad un nodo objetivo es un estado que viola una proposición lógica). Otros problemas que pueden plantearse como una búsqueda en grafos desconocidos o de muy gran tamaño son la búsqueda de movimientos óptimos en juegos y la demostración automática de teoremas. La cantidad de estados posibles de un juego (ajedrez por ejemplo) es realmente elevada y es imposible almacenarlos o explorarlos todos para decidir el siguiente movimiento a realizar. En cuanto a la demostración automática de teoremas, algunos demostradores descomponen el teorema en cláusulas y usan el método de resolución para conseguir una cláusula vacía. En cada paso estos demostradores deben elegir una cláusula de entre las disponibles para aplicar resolución. Cada una de estas cláusulas puede ser vista como un arco que lleva a un nuevo estado de la demostración y de esta forma el problema puede ser planteado como una búsqueda en un grafo (el objetivo es encontrar un estado que contenga la cláusula vacía). En este caso el número de estados del grafo puede ser infinito.

Este tipo de problemas no puede resolverse con los modelos de ACO existentes en la literatura debido a varios motivos. En primer lugar, si permitimos a las hormigas caminar por el grafo sin repetir nodo hasta que encuentren un nodo objetivo podrían llegar a un nodo sin sucesores no visitados (un nodo sin salida para las hormigas). Incluso si encuentran un nodo objetivo puede ser necesario mucho tiempo y memoria para construir una solución candidata. Por tanto, no es viable, en general, trabajar con soluciones completas como hacen los actuales modelos. Es necesario poder trabajar con soluciones parciales. Por otro lado, algunos modelos de ACO asumen que el número de nodos del grafo de construcción se conoce de antemano y la cantidad inicial de feromona de cada arco depende de este número de nodos. También debemos tener cuidado con la implementación de los rastros de feromona. En los modelos previos la matriz de feromonas se almacena en un array, pero esto requiere conocer el número de nodos. En nuestro caso, incluso si conociéramos dicho número, no podríamos almacenar la matriz de feromonas en arrays debido a la gran cantidad de memoria requerida.

Para solventar las dificultades que surgen al trabajar con grafos de gran dimensión, proponemos aquí una variante, llamada ACOhg, que es capaz de abordar problemas con un grafo de construcción subyacente de tamaño desconocido. Las principales cuestiones que tenemos que resolver están relacionadas con la longitud de los caminos de las hormigas, la cantidad de memoria usada para almacenar la matriz de feromona y el grafo de construcción. Abordaremos estas cuestiones a continuación. El resto de los detalles del modelo tales como la actualización de feromonas o la fase de construcción son iguales que en los modelos existentes.

A. Longitud de los Caminos de las Hormigas

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

La propuesta anterior resuelve el problema de las “hormigas errantes” pero introduce un nuevo problema: tenemos un nuevo parámetro para el algoritmo (λ_{ant}). No es fácil establecer a priori cuál es el mejor valor para este nuevo parámetro. Si se elige un valor menor que la profundidad² de todos los nodos obje-

²Definimos la *profundidad* de un nodo en el grafo de cons-

tivo el algoritmo no encuentra ninguna solución. Por tanto, debemos escoger un valor mayor que la profundidad de algún nodo objetivo (de hecho, veremos en la sección experimental que este valor debe ser a veces varias veces mayor que la profundidad del nodo objetivo). Esto no es difícil cuando conocemos la profundidad de un nodo objetivo, pero normalmente se da la situación contraria, en cuyo caso podemos usar dos alternativas que dan lugar a dos variantes del modelo.

La primera consiste en incrementar λ_{ant} durante la búsqueda si no se encuentra ningún nodo objetivo. Al principio se asigna un valor bajo a λ_{ant} y se incrementa en una determinada cantidad cada cierto número de pasos del algoritmo. De esta forma, en algún momento la longitud de los caminos será suficientemente larga como para alcanzar algún nodo objetivo. Este mecanismo puede ser útil cuando la profundidad de los nodos objetivos no es muy alta. En caso contrario, la longitud de los caminos de las hormigas crecerá mucho y lo mismo sucederá con el tiempo y la memoria requeridos para construir estos caminos.

La segunda alternativa consiste en comenzar la construcción del camino de las hormigas en nodos diferentes durante la búsqueda. Al principio las hormigas se colocan en el nodo inicial del grafo y el algoritmo se ejecuta durante un número dado de pasos. Si no se encuentra ningún nodo objetivo, los últimos nodos de los caminos construidos por las hormigas se usan como nodos iniciales para las siguientes hormigas. El algoritmo continúa su ejecución y las nuevas hormigas comienzan su exploración al final de los caminos de las hormigas previas, intentando ir más allá en el grafo. La longitud de estos caminos se mantiene constante durante la búsqueda y los rastros de feromona pueden olvidarse de una etapa a otra para mantener casi constante la cantidad de recursos computacionales (memoria y CPU) en todas las etapas. La elección del nodo inicial para las hormigas se realiza en dos fases. Primero, necesitamos elegir los nodos finales de la etapa previa que vamos a usar como nodos de comienzo en la nueva. Para esto tenemos varias opciones: podemos elegir los mejores caminos de forma determinista (de acuerdo con el valor heurístico de su último nodo o la calidad de la solución parcial que representa) o podemos elegir los caminos aleatoriamente (asignando una probabilidad de selección a cada camino que dependa de nuevo del heurístico o del fitness). Una vez que tenemos el conjunto de nodos de comienzo tenemos que asignar las nuevas hormigas a esos nodos. Aquí podemos asignar a cada nodo una probabilidad de ser seleccionado. En la Figura 2 ilustramos gráficamente la forma en que trabajan estas dos alternativas presentadas.

trucción como la distancia del camino más corto que llega a él partiendo del nodo inicial.

B. La Matriz de Feromonas

En ACOhg la matriz de feromonas es sustituida por una matriz dispersa donde sólo se almacenan los valores de feromona de los arcos por los que pasan las hormigas. Este es el modo natural de implementar la matriz de feromona en el nuevo modelo. No obstante, conforme la búsqueda progresa, la memoria requerida para dicha matriz puede crecer hasta cantidades inadmisibles. Podemos evitar esto eliminando de la matriz de feromona los rastros con poca influencia en la fase de construcción, es decir, los arcos con un valor bajo de feromona asociada. Definimos τ_θ como el valor umbral para eliminar arcos. Todos los arcos con un rastro de feromona por debajo de τ_θ se eliminan de la matriz de feromona.

La eliminación de feromona se puede aplicar cuando cierta condición se cumpla, por ejemplo, cuando la memoria libre está por debajo de un cierto umbral o se haya alcanzado un número predefinido de iteraciones desde la última vez que se eliminó feromona. También se puede aplicar de forma continua cada vez que un rastro es actualizado. Es decir, los rastros de feromona por debajo de τ_θ son eliminados automáticamente cuando se actualizan. Esta última alternativa tiene la ventaja de que evitamos la búsqueda de rastros de feromona por debajo de τ_θ en la matriz de feromona (que puede ser una labor muy costosa). Además, la eliminación continua de feromona equivale a una inicialización continua de rastros poco influyentes, lo cual da una segunda oportunidad a caminos poco prometedores en principio.

C. El Grafo de Construcción

Una última cuestión que hay que tener en cuenta en el nuevo modelo es la generación del grafo de construcción. A diferencia de otros modelos de ACO, en el nuestro el grafo se genera conforme se realiza la exploración. Esto significa que conforme avanza la búsqueda será necesaria más memoria para almacenar los nodos del grafo. Los arcos se almacenan implícitamente en la matriz de feromona. Es necesario cuidar la implementación para ahorrar memoria. Por ejemplo, si el tamaño de los nodos en memoria supera el tamaño de un puntero se puede mantener una sola copia del nodo y hacer referencia a ella allá donde se necesite. En este caso hay que evitar duplicados de un nodo en memoria. No obstante, si el tamaño de los nodos es igual o inferior al de un puntero el mecanismo anterior no es útil y convierte la implementación en ineficiente. Otro mecanismo que permite ahorrar memoria y que se utiliza en model checking es la compresión de los nodos. Por otro lado, si se emplea algún mecanismo de eliminación de feromona, los nodos extremos de los arcos eliminados podrían ser eliminados si no son referenciados por ningún otro arco u hormiga.

Las alternativas arriba mencionadas son mecanismos de ahorro de memoria que deben ser tenidos

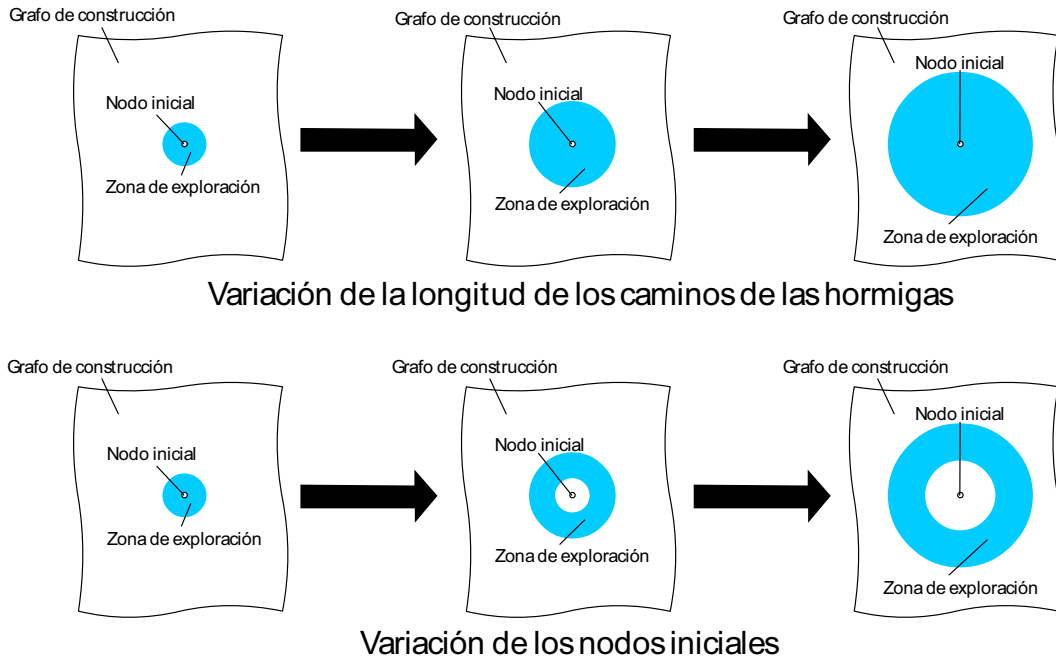


Fig. 2. Las dos variantes del modelo en diferentes etapas de la búsqueda.

en cuenta en la implementación del algoritmo para maximizar la región del grafo que puede almacenarse en memoria, ya que es en esta región donde el algoritmo puede trabajar con mayor eficacia para maximizar la calidad de las soluciones parciales.

IV. ESTUDIO EXPERIMENTAL

En esta sección ilustraremos la aplicación de un ACOhg al problema de refutación de propiedades de seguridad en sistemas concurrentes. En primer lugar introduciremos el problema y después daremos algunos detalles sobre el software implementado para realizar los experimentos. Por último, mostraremos los resultados obtenidos con nuestro algoritmo cuando tratamos de buscar errores en un sistema que modela el conocido problema de los filósofos.

A. Verificación de Sistemas

Desde el comienzo de la informática los programadores e ingenieros del software se han interesado en técnicas que les permita saber si un determinado módulo software cumple un conjunto de requisitos (la especificación). El software moderno es muy complejo y estas técnicas se han convertido en una necesidad en la mayoría de las empresas de software. Una de estas técnicas es *model checking* [7], que consiste en analizar exhaustivamente (de una forma directa o indirecta) todos los posibles estados de un sistema concurrente para demostrar o refutar que el programa satisface una propiedad dada. Esta propiedad se especifica usando una lógica temporal como LTL [8] o CTL [9]. Una de las herramientas más conocidas que realizan model checking es SPIN [10]. En este model checker el modelo de entrada se debe codificar en el lenguaje PROMELA y

la propiedad se debe especificar en LTL. SPIN transforma el modelo y la negación de la propiedad en autómatas de Büchi para, posteriormente, realizar el producto síncrono de ambos autómatas y explorar el nuevo autómata en busca de un ciclo de estados alcanzable desde el estado inicial y que contenga un estado de aceptación. Si lo encuentra, existe una ejecución del sistema que incumple la propiedad dada (véase [11] para más detalles). Si no lo encuentra la verificación acaba con éxito. El algoritmo que sigue SPIN para realizar la exploración del grafo es una búsqueda en profundidad anidada (Nested-DFS).

Las propiedades de seguridad de un sistema concurrente se pueden comprobar buscando solamente un estado de aceptación en el autómata, es decir, no es necesario encontrar un ciclo adicional que contenga un estado de aceptación. Esto significa que la verificación de propiedades de seguridad se puede transformar en un problema de búsqueda de un nodo objetivo en un grafo y el camino desde el nodo inicial hasta el nodo objetivo representa una ejecución del sistema concurrente en la que no se cumple la propiedad de seguridad comprobada. Este hecho ha sido utilizado en trabajos previos para verificar propiedades de seguridad usando algoritmos clásicos de exploración de grafos (tales como A* o Weighted A*) [12], [13] e incluso algoritmos genéticos [14], [15]. Si el algoritmo consigue encontrar un camino hasta un nodo objetivo la propiedad ha sido refutada y el camino es un contraejemplo, pero verificar que el sistema posee la propiedad requiere explorar todos los posibles caminos para asegurarse de que no hay nodos objetivo. La mayoría de los algoritmos metaheurísticos canónicos, por su carácter aproximado, no pueden asegurar que la propiedad se verifica pero

pueden refutarla. Por este motivo hablamos de un problema de refutación de propiedades en lugar de verificación. Para guiar la búsqueda se suele asociar a cada estado del autómata un valor heurístico que es una cota inferior de la distancia a la que se encuentra un nodo objetivo. El cálculo de este valor heurístico puede estar basado en la fórmula temporal a verificar [13] o en el estado objetivo (si se conoce de antemano) [16].

B. Integración de ACOhg y SPIN

La implementación de nuestro modelo se ha realizado dentro de la biblioteca MALLBA [17], usando como base una implementación previa y general de los modelos ACO existentes realizada por Guillermo Ordóñez [18]. La incorporación del modelo dentro de MALLBA tiene la ventaja, entre otras, de que puede paralelizarse sin mucho esfuerzo y de forma transparente a los usuarios finales.

Por otro lado, existe un model checker desarrollado por Stefan Edelkamp y Alberto Lluch-Lafuente llamado HSF-SPIN que integra una biblioteca de algoritmos de exploración de grafos (HSF) y SPIN permitiendo la aplicación de métodos de búsqueda heurística a la verificación de sistemas modelados en PROMELA [12]. Nosotros hemos incorporado MALLBA dentro de HSF-SPIN para poder usar la implementación de nuestro modelo de ACO. De esta forma, podemos despreocuparnos de los detalles relacionados con la representación de modelos (interpretación del lenguaje PROMELA, fórmulas LTL, etc.) y disponemos además de una gran cantidad de funciones heurísticas listas para usar (ya implementadas dentro de HSF-SPIN).

C. Resultados Experimentales

En esta sección mostramos los resultados de aplicar un ACOhg a la búsqueda de interbloqueos (un tipo de error) en el problema de los filósofos de Edsger Dijkstra. Una breve descripción del problema es la siguiente. Un cierto número n de filósofos se sientan en una mesa circular para comer arroz. En la mesa hay n palillos chinos distribuidos de forma circular. Cada filósofo necesita dos palillos para comer, el de su izquierda y el de su derecha, los cuales toma en ese orden. Este escenario se puede modelar en un computador usando n procesos que toman los “palillos” para “comer” y los liberan tras saciarse, permitiendo comer a otros “filósofos”. En el modelo así descrito puede producirse un interbloqueo cuando todos los filósofos cogen el palillo de su izquierda y esperan a que su compañero suelte el palillo de su derecha. Hemos usado cuatro instancias del problema con 8, 12, 16 y 20 filósofos. En primer lugar, realizaremos un estudio sobre la influencia de λ_{ant} en los resultados cuando las hormigas parten siempre del nodo inicial. Posteriormente, comparamos los resultados obtenidos por el modelo con los que se obtienen al aplicar técnicas exactas como DFS y BFS.

En todos los casos, los mecanismos de construcción de soluciones y actualización de feromonas se corresponden con los de un MAX-MIN Ant System (MMAS), por lo que llamaremos al algoritmo resultante MMAShg. En la fase de actualización de feromona sólo la mejor solución encontrada es usada para aumentar el rastro de feromona del camino trazado (elitismo). En todo momento los rastros deben estar dentro de un intervalo $[min, max]$ donde max coincide con la inversa del menor valor de fitness encontrado y $min = max/a$. El valor de a es 5 en los experimentos. Si un rastro de feromona se encuentra temporalmente fuera del rango debido a una actualización, se ajusta al extremo más cercano del intervalo. El valor de fitness de un camino es la profundidad conocida del último nodo del camino. No se ha usado ninguna información heurística para guiar la búsqueda, de esta forma queremos estudiar el comportamiento de MMAShg en ausencia de información. La colonia empleada posee 10 hormigas y el algoritmo se detiene cuando encuentra un nodo objetivo o realiza un máximo de 10 pasos en cada ejecución (realizando como máximo un total de 100 evaluaciones de la función de fitness). Se ha tomado este bajo número de evaluaciones porque de esta forma se puede apreciar la influencia de λ_{ant} en la tasa de éxito³. Experimentos previos demostraron que usando un número más alto de evaluaciones la tasa de éxito ascendía al 100% en todos los casos. La máquina empleada fue un Pentium 4 a 2.8GHz con 512 MB.

C.1 Influencia de λ_{ant}

En esta sección vamos a estudiar cuál debe ser la longitud máxima del camino de las hormigas λ_{ant} para encontrar un nodo objetivo. Más concretamente, fijaremos λ_{ant} sin modificar el nodo inicial de las hormigas y analizaremos su influencia en la tasa de éxito del algoritmo. Para cada instancia hemos usado distintos valores de λ_{ant} en distintas ejecuciones. Concretamente, se han usado valores que van desde la profundidad del nodo objetivo más cercano d_{opt} (solución óptima) hasta $6d_{opt}$ aumentando en intervalos de $0.5d_{opt}$. Por comodidad en las discusiones siguientes llamaremos η al *coeficiente de aumento de longitud* calculado como el cociente λ_{ant}/d_{opt} . No hemos incorporado ningún mecanismo de ahorro de memoria para poder analizar la influencia de λ_{ant} aisladamente. Para cada instancia y valor de λ_{ant} se han realizado 100 ejecuciones independientes. En las Tablas I y II se presentan la tasa de éxito (número de ejecuciones que encontraron una solución) y la longitud media de las soluciones encontradas, respectivamente.

Podemos observar en las últimas filas de la Tabla I que la tasa de éxito aumenta al permitir a las hormi-

³En este problema, una ejecución tiene *éxito* cuando encuentra una traza que lleva a un estado de interbloqueo, refutando la propiedad de seguridad de ausencia de interbloqueos.

TABLA I
TASA DE ÉXITO (%) EN FUNCIÓN DE η .

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

TABLA II
LONGITUD MEDIA DE LA SOLUCIÓN ENCONTRADA EN FUNCIÓN DE η .

| Número de Filósofos | Coeficiente de aumento de longitud (η) | | | | | | | | | | |
|---------------------|---|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|
| | 1.0 | 1.5 | 2.0 | 2.5 | 3.0 | 3.5 | 4.0 | 4.5 | 5.0 | 5.5 | 6.0 |
| 8 | 10.00 | 10.00 | 11.68 | 16.80 | 14.08 | 16.32 | 16.80 | 12.28 | 17.20 | 13.88 | 14.48 |
| 12 | 14.00 | 16.04 | 17.52 | 18.00 | 18.00 | 17.72 | 17.76 | 18.00 | 18.00 | 18.00 | 18.00 |
| 16 | 18.00 | 21.54 | 25.40 | 25.64 | 32.68 | 34.68 | 30.08 | 38.84 | 47.72 | 51.64 | 60.08 |
| 20 | 22.00 | 27.69 | 31.14 | 38.68 | 45.44 | 49.68 | 56.60 | 47.52 | 74.36 | 72.64 | 76.92 |

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

El hecho de que el algoritmo encuentre más fácilmente la traza de error cuando λ_{ant} es mayor se deja notar también en la memoria y el tiempo empleado para encontrar dicha traza. En las Figuras 3 y 4 se muestra gráficamente la cantidad de memoria usada (en kilobytes) y el tiempo medio empleado (en milisegundos). Como se puede apreciar, tanto la memoria como el tiempo disminuyen al aumentar λ_{ant} . Esto es debido a que la probabilidad de que una hormiga encuentre un nodo objetivo es más alta y no son necesarios tantos pasos del algoritmo. De esta forma, además de reducir el tiempo de cómputo se reduce la cantidad de rastros de feromona que almacena la matriz.

C.2 Comparación con Algoritmos Exactos

En esta última sección compararemos los resultados obtenidos por nuestro \mathcal{MMAShg} con dos algoritmos exactos: búsqueda en profundidad (DFS) y búsqueda en anchura (BFS). El primero de ellos puede considerarse una simplificación para propiedades de seguridad del algoritmo que usa SPIN para realizar model checking (Nested-DFS). Por su parte, BFS asegura encontrar el camino más corto hasta un nodo objetivo (en nuestro caso esto es la traza de ejecución más corta que acaba en interbloqueo). Existen otros algoritmos exactos tales como A^* y WA^* que se guían influidos por infor-

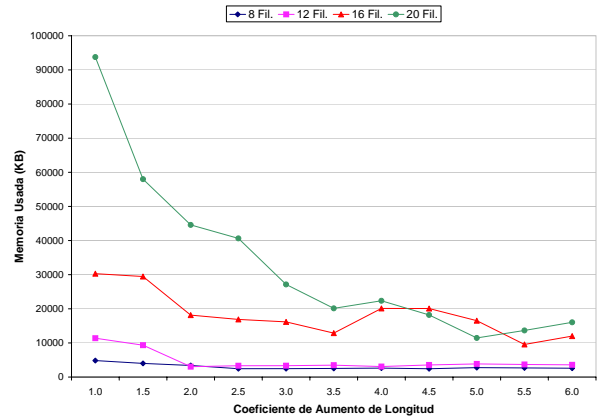


Fig. 3. Memoria máxima usada por el algoritmo.

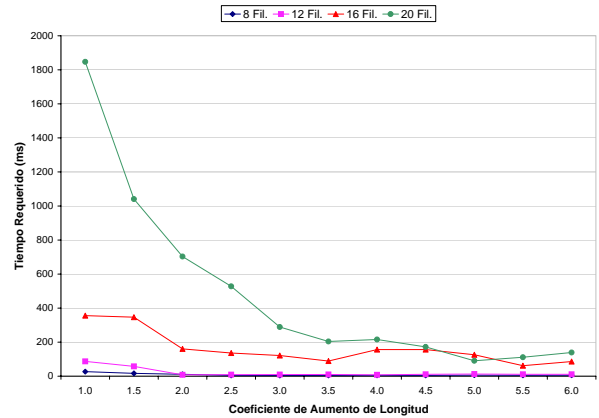


Fig. 4. Tiempo requerido por el algoritmo.

mación heurística y que han sido aplicados a un modelo de verificación con muy buenos resultados. Sin embargo, hemos escogido DFS y BFS porque, al igual que \mathcal{MMAShg} , no usan información heurística y podemos realizar una comparación justa. Mostramos en la Tabla III la longitud de la traza, el tiempo empleado y la memoria requerida por cada algoritmo para encontrar una traza de error en el problema de los 8 filósofos. En el caso de \mathcal{MMAShg} presentamos la versión con $\lambda_{ant} = 10$.

TABLA III
COMPARACIÓN ENTRE MMAShg Y ALGORITMOS EXACTOS.

| Algoritmo | Long. | Memoria (KB) | Tiempo (ms) |
|-----------|-------|--------------|-------------|
| MMAShg | 10 | 4830 | 27 |
| DFS | 1338 | 29696 | 70 |
| BFS | 10 | 17408 | 70 |

Como podemos observar, MMAShg encuentra la traza más corta usando menos memoria y tiempo que los algoritmos exactos. Además, no fue posible resolver el problema con DFS y BFS para el resto de las instancias (12, 16 y 20 filósofos) debido a limitaciones de memoria (el proceso era en todos los casos terminado por el sistema operativo). Estos resultados suponen un inicio prometedor para la aplicación de ACO en el dominio de verificación software.

V. CONCLUSIONES Y TRABAJO FUTURO

En el presente artículo hemos presentado un nuevo modelo de ACO que permite resolver problemas cuyo grafo subyacente sea de gran tamaño y deba construirse mientras se realiza la búsqueda. Este modelo supera las limitaciones que los otros modelos existentes de ACO poseen y que les impiden trabajar con este tipo de problemas.

Para ilustrar el funcionamiento del modelo se ha escogido un problema con un gran interés en la ingeniería del software: la refutación de propiedades de seguridad en sistemas concurrentes. Se han buscado estados de interbloqueo en un sistema que implementa el problema de los filósofos. Hemos estudiado cómo influye la longitud máxima permitida a las hormigas en la tasa de éxito, la memoria y el tiempo requeridos. Los resultados muestran que la tasa de éxito aumenta con la longitud máxima. Lo mismo ocurre con la longitud media de las soluciones encontradas. Adicionalmente, la memoria y tiempo requeridos para encontrar una traza de error disminuyen al permitir a las hormigas construir caminos más largos. Por otro lado, en comparación con técnicas exactas como DFS y BFS, nuestro algoritmo consigue la solución óptima usando menos memoria y en un menor tiempo. Además es capaz de abordar instancias de un tamaño que resulta inalcanzable para los algoritmos exactos en una máquina corriente.

Con este artículo iniciamos dos líneas de trabajo futuro en direcciones ortogonales. Por un lado, es necesario estudiar el modelo más a fondo. Debemos explorar todas las alternativas mencionadas en este trabajo y estudiar sus ventajas e inconvenientes para poder identificar rápidamente cuál es la mejor opción según el problema a resolver. Además, podemos aplicar las ideas presentadas en la Sección III a otras metaheurísticas: aunque han sido planteadas para ACOs, dichas ideas son extensibles a otros algoritmos. Por otro lado, la aplicación de técnicas metaheurísticas al dominio de los métodos formales en la ingeniería del software y, concretamente, a la verificación es un campo poco explorado. Los tra-

bajos previos están basados en algoritmos genéticos y poseen limitaciones debidas fundamentalmente a la representación de las soluciones. En este sentido, debemos profundizar en la aplicación y optimizar el modelo presentado aquí para poder superar a los algoritmos usados en model checking actualmente.

AGRADECIMIENTOS

Este trabajo está parcialmente financiado por el Ministerio de Educación y Ciencia y FEDER con número de proyecto TIN2005-08818-C04-01 (proyecto OPLINK). Francisco Chicano disfruta de una beca de la Junta de Andalucía (BOJA 68/2003).

REFERENCIAS

- [1] C. Blum and A. Roli, "Metaheuristics in combinatorial optimization: Overview and conceptual comparison," *ACM Comp. Surveys*, vol. 35, no. 3, pp. 268–308, 2003.
- [2] M. Dorigo and T. Stützle, *Ant Colony Optimization*, The MIT Press, 2004.
- [3] M. Dorigo, V. Maniezzo, and A. Coloni, "Positive feedback as a search strategy," Tech. Rep. 91-016, Dipartimento di Elettronica, Politecnico di Milano, Italy, 1991.
- [4] K. Socha and C. Blum, *Metaheuristic Procedures for Training Neural Networks*, chapter 8, Ant Colony Optimization, pp. 153–180, Springer, 2006.
- [5] G. Leguizamón and Z. Michalewicz, "A new version of Ant System for subset problems," in *Proceedings of the 1999 Congress on Evolutionary Computation*, P.J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzal, Eds., Piscataway, New Jersey, USA, 1999, pp. 1459–1464, IEEE Computer Society Press.
- [6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," in *Informatics: 10 Years Back, 10 Years Ahead*, 2000, vol. 2000 of LNCS, pp. 176–194, Springer-Verlag.
- [7] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, The MIT Press, January 2000.
- [8] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs, Workshop*, London, UK, 1982, pp. 52–71, Springer-Verlag.
- [9] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.
- [10] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. on Soft. Eng.*, vol. 23, no. 5, pp. 1–17, May 1997.
- [11] G. J. Holzmann, *The SPIN Model Checker*, Addison-Wesley, 2004.
- [12] S. Edelkamp, A. Lluch-Lafuente, and S. Leue, "Directed Explicit Model Checking with HSF-SPIN," in *LNCS, 2057*, 2001, pp. 57–79, Springer.
- [13] S. Edelkamp, A. Lluch-Lafuente, and S. Leue, "Protocol Verification with Heuristic Search," in *AAAI-Spring Symposium on Model-based Validation Intelligence*, 2001, pp. 75–83.
- [14] E. Alba and J.M. Troya, "Genetic Algorithms for Protocol Validation," in *Proceedings of the PPSN IV International Conference*, Berlin, 1996, pp. 870–879, Springer.
- [15] P. Godefroid and S. Khurshid, "Exploring Very Large State Spaces Using Genetic Algorithms," in *LNCS, 2280*, 2002, pp. 266–280, Springer.
- [16] A. Lluch-Lafuente, "Symmetry Reduction and Heuristic Search for Error Detection in Model Checking," in *Workshop on Model Checking and Artificial Intelligence*, August 2003.
- [17] E. Alba and C.Cotta, "Optimización en entornos geográficamente distribuidos. proyecto MALLBA," in *Actas del Primer Congreso Español de Algoritmos Evolutivos y Bioinspirados*, Mérida, 2002, pp. 38–45.
- [18] E. Alba, G. Leguizamón, and G. Ordóñez, "Analyzing the behavior of parallel ant colony systems for large instances of the task scheduling problem," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, April 2005, IEEE Computer Society.