Trabajo de Fin de Grado

# Paralelización del algoritmo A*


Universidad de Murcia

Autor:
Miguel Ángel Rodríguez García

Directores:
A. Javier Cuenca Muñoz y José Matías Cutillas Lozano

2025

# Resumen

En este trabajo exploramos la paralelización del algoritmo de búsqueda A*. Se trata de un algoritmo heurísitco que resuelve el problema de encontrar el camino más corto entre dos vértices de un grafo, y que a día de hoy es ampliamente utilizado en una gran variedad de campos como el desarrollo de videojuegos, el transporte o la robótica. A lo largo de los últimos años, se han propuesto diversas estrategias para obtener una versión paralela de este algoritmo, tarea que no ha sido facil debido al fuerte carátacter secuecial que presenta.

Comenzaremos introduciendo los conceptos matemáticos necesarios relacionados con la teoría de grafos para entender la terminología utilizada en el resto del trabajo. A continuación, introduciremos los distintos paradigmas de algoritmos búsqueda que han surgieron en la segunda mitad del siglo XX y explicaremos en detalle el funcionamiento de A*. También haremos un análisis del mismo en términos de admisibildad y optimalidad para resolver el problema del camino más corto.

A continuación, se presentarán las distintas estrategias de paralelización propuestas en la literatura, evaluando sus ventajas y limitaciones. Veremos cómo varias ideas clave han impulsado una evolución progresiva en este campo a lo largo de las últimas décadas, desembocando en dos paradigmas principales para el desarrollo de formulaciones paralelas del algoritmo.

Finalmente, se expondrá el desarrollo e implementación de tres versiones de A*: una secuecnial y dos paralela basadas en los dos principales paradigmas de paralelización, siendo uno de ellos el algoritmo del estado del arte actual. Analizaremos los resultados obtenidos y veremos como cambian radicalemente los resultados obtenidos según el enfoque utilizado.

# Extended Abstract

In this work, we explore the parallelization of the A* search algorithm. This is a heuristic algorithm that solves the problem of finding the shortest path between two vertices in a graph and is widely used today in a variety of fields such as video game development, transportation, and robotics. Over the past few years, various strategies have been proposed to obtain a parallel version of this algorithm—a task that has proven challenging due to the algorithm's inherently sequential nature.

We begin by introducing the necessary mathematical concepts from graph theory to understand the terminology used throughout the rest of the work. Then, we review the main search algorithm paradigms that emerged in the second half of the 20th century and provide a detailed explanation of how A* works. We also analyze the algorithm in terms of admissibility and optimality for solving the shortest path problem.

Next, we present the different parallelization strategies proposed in the literature, evaluating their advantages and limitations. We examine how several key ideas have driven steady progress in this field over recent decades, leading to two main paradigms for developing parallel formulations of the algorithm.

Finally, we describe the development and implementation of three versions of A*: one sequential and two parallel, based on the two main parallelization paradigms, one of which represents the current state of the art. We analyze the results obtained and observe how the performance outcomes vary significantly depending on the chosen approach.

# Índice

# 1. Introducción

El Algoritmo A* es un tipo de algoritmo de búsqueda en grafos heurístico (o informado) que garantiza, bajo unas determinadas condiciones, encontrar el camino de coste mínimo entre un par de nodos. Este problema, conocido como el problema del camino más corto (*Shortest Path Problem*), puede parecer trivial a simple vista, pero es fácil darse cuenta de que, a medida que se consideran un mayor número de nodos y aristas, la dificultad de resolverlo aumenta rápidamente. Podemos pensar que este tipo de problemas tienen una relevancia natural relacionada con las necesidades de los seres humanos, los animales o las plantas. Localizar la ruta más corta para, por ejemplo, conseguir comida o refugio, podría representar una ventaja en la adaptación de una especie a su entorno.

Lo cierto es que sí podemos encontrar en la naturaleza varios ejemplos de estrategias y acercamientos a este tipo de problemas. Uno de ellos es la forma en que las hormigas utilizan las feromonas para encontrar la ruta óptima entre la colonia y una fuente de alimento [5]. Tras explorar el espacio aleatoriamente, cuando una de las hormigas encuentra alimento, vuelve a la colonia de forma más o menos correcta, dejando un rastro de feromonas. Este rastro será seguido por otras hormigas, que lo reforzarán al volver si efectivamente encontraron alimento. Otro ejemplo son los túbulos que produce el moho *Physarum polucephalum* para conectar distintas colonias con sus fuentes de alimentos. Con este último se realizó un experimento en el que, sobre una placa translúcida que replicaba los accidentes geográficos de Tokio, y colocando alimento en las ubicaciones de Tokio y sus ciudades periféricas, se observó que el moho construyó una red de túbulos muy cercana a la actual red ferroviaria de Tokio [53]. Este experimento les valió a sus investigadores el premio Ig Nobel de Planificación de Transporte en 2010.

Para los humanos no ha sido diferente. Con el auge de las civilizaciones, comienzan a surgir escenarios en los que la optimización de rutas es cada vez más relevante. En Mesopotamia y Egipto, la construcción de caminos y redes fluviales eficientes permitió el transporte de mercancías y personas. El Imperio Romano destacó por su vasta red de calzadas que conectaban todo su territorio, optimizando el comercio y la movilidad militar.

A pesar de todo esto, la investigación matemática formal en este problema comenzó relativamente tarde, sobre todo si la comparamos con la de otros problemas de optimización combinatoria, como el del árbol de expansión mínimo [2]. La primera referencia encontrada en la literatura sobre el problema del camino más corto la encontramos en [55], donde se refiere en particular al problema de encontrar la salida en un laberinto. Se cree que esta aparición tardía puede deberse a que es un problema relativamente fácil de resolver, lo que además explica el hecho de que, cuando este problema comenzó a adquirir popularidad, varios investigadores independientes habían desarrollado métodos similares [49]. Otra

razón para explicar esta aparición tardía es el hecho de que hubo que esperar hasta el siglo XVIII para contar con un marco teórico matemático sobre el que trabajar este tipo de problemas. Se trata de la teoría de grafos, desarrollada por Leonhard Euler, y que utilizó para resolver el famoso problema de los puentes de Königsberg [12].

Desde entonces, la teoría de grafos ha evolucionado significativamente y, junto con los avances en ciencias de la computación, ha motivado la investigación en el área de los algoritmos de búsqueda. Además, con ayuda de conceptos como el de la representación atómica para el modelado de problemas [48, p. 57], el ámbito de aplicación se ha extendido mucho más allá de la búsqueda de un camino más corto, siendo de especial interés en áreas como la inteligencia artificial, la bioinformática, la toma de decisiones o la optimización de redes de transporte.

## 1.1. Conceptos matemáticos

En esta sección vamos a introducir algunas definiciones matemáticas que nos ayudarán a entender el formalismo sobre el que se sustentan los métodos de búsqueda en grafos, así como una descripción del problema del camino más corto y los conceptos esenciales para comprender el funcionamiento del algoritmo.

**Definición 1.** *Un **grafo** $G$ es un par ordenado $G = (V, E)$, donde $V$ es un conjunto de elementos $\{n_i\}$ llamados vértices (o nodos), y $E$ es un conjunto de elementos $\{e_{ij}\}$ llamados aristas (o arcos). Si el elemento $e_{pq}$ pertenece al conjunto $\{e_{ij}\}$, entonces decimos que existe un arco del nodo $n_p$ al nodo $n_q$.*

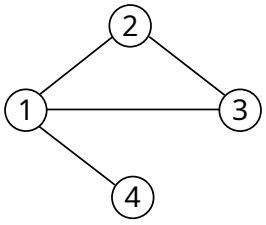



**Figura 1:** Grafo no dirigido

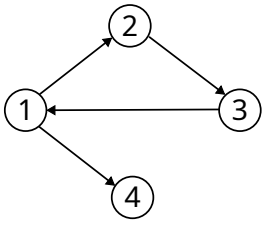



**Figura 2:** Grafo dirigido

Podemos hablar de **grafo no dirigido** (Figura 1) cuando los elementos de $E$ son pares no ordenados, de manera que $e_{pq} = e_{qp}$, y de **grafo dirigido** (Figura 2) cuando se trata de pares ordenados (en general, $e_{pq} \neq e_{qp}$). En estos últimos, la existencia del elemento $e_{pq}$ no implica la existencia del elemento $e_{qp}$. Para poder realizar una buena representación del problema del camino más corto, trabajaremos con **grafos dirigidos ponderados**.

**Definición 2.** *Un **grafo dirigido ponderado** es un grafo dirigido $G = (V, E)$, en el que cada elemento $e_{ij} \in E$ tiene asociado un coste $c_{ij} \in \mathbb{R}$.*

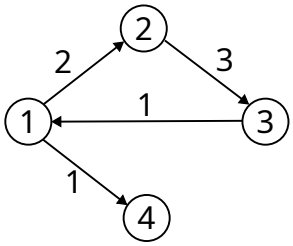



**Figura 3:** Caption

Consideraremos solo aquellos grafos dirigidos ponderados para los que el coste de cualquier arista es positivo y no nulo.

Esta forma explícita de definir un grafo a partir de sus conjuntos de nodos y aristas atiende a la manera convencional en teoría de grafos, pero cuando se trata con estos problemas desde el punto de vista de la computación, es usual utilizar otra forma equivalente. Se trata de una definición implícita a partir de un conjunto de nodos fuente $S \subset V$ y un operador de vecindad $\Gamma$ definido sobre el conjunto de elementos $\{n_i\}$ cuyo valor para cada $n_i$ es un conjunto de pares $\{(n_j, c_{ij})\}$. De esta forma, a partir de un nodo inicial y un operador de vecindad podemos obtener el grafo completo.

**Definición 3.** *Un **camino** de $n_1$ a $n_k$ es un conjunto ordenado de nodos $P = (n_1, n_2, \ldots, n_k)$ donde cada nodo $n_{i+1}$ es sucesor del nodo $n_i$. En un grafo ponderado, el camino entre tendrá un coste asociado $C(P) = \sum_{i=0}^{k-1} c_{i,i+1}$.*

**Definición 4.** *El **camino óptimo** del nodo $n_i$ al nodo $n_k$ es el camino con el menor coste de entre todos los posibles caminos de $n_i$ a $n_k$. A este coste lo representamos como $h(n_i, n_k)$.*

Por simplicidad, y cuando el nodo objetivo $n_k$ es único, podemos representar el coste del camino óptimo desde un nodo $n_i$ al nodo objetivo como $h(n_i)$. Los algoritmos heurísticos hacen uso de esta función, denominada **heurística** para discriminar cuál de las siguientes opciones es más prometedora (eligiendo aquella que minimiza o maximiza su valor). Diremos que un algoritmo es **admisible** si está garantizado que encontrará la solución óptima.

Ahora, ya podemos formular el problema a resolver y explicar el funcionamiento del algoritmo.

**Definición 5.** *Dado un grafo dirigido con pesos positivos y dos nodos $s$ y $t$, el **problema del camino más corto** consiste en encontrar una secuencia de nodos $P = (n_1, n_2 \ldots, n_k)$ tal que $n_0 = s$ y $n_k = t$, cada par consecutivo $n_i, n_{i+1}$ está conectado por una arista del grafo, y el coste total del camino es el mínimo posible de entre todos los caminos que conectan $s$ con $t$.*

## 1.2. Algoritmos de búsqueda

Un algoritmo de búsqueda es un procedimiento sistemático que explora un espacio de posibles soluciones con el objetivo de encontrar una o varias que cumplan ciertas condiciones, típicamente una solución óptima o satisfactoria, a un problema. Estos algoritmos están estrechamente relacionados con el problema del camino más corto, pues son la base para las estrategias que se llevan a cabo en este problema.

A día de hoy podemos clasificar los algoritmos de búsqueda en dos categorías, los no informados y los informados o heurísticos. La diferencia principal está en que los algoritmos no informados no poseen ningún tipo de conocimiento sobre el dominio del problema, mientras que los informados disponen de alguna forma de evaluar si una opción es más prometedora que otra. Esta limitación de los algoritmos no informados con respecto a los informados hace que, en general, su eficiencia sea mucho menor.

Las distintas estrategias para realizar búsquedas no informadas se basan en explorar el espacio de búsqueda de alguna manera más o menos ordenada, hasta dar con el objetivo, sin tener más información que el poder saber si el estado actual es el objetivo o no. Las dos principales estrategias son la búsqueda en profundidad (DFS) y la búsqueda en anchura (BFS), que se diferencian en el orden en que se exploran los estados vecinos. El objetivo de estos algoritmos es el de encontrar si existe dicho estado objetivo o no, pero no resuelven el problema del camino más corto. Para ello, debemos tener en cuenta el coste acumulado desde el inicio hasta el nodo objetivo. Ejemplos de estos algoritmos son la búsqueda de coste uniforme (UCS) o el algoritmo de Dijkstra [9].

Los algoritmos de búsqueda informados se basan en el diseño de una función heurística. Esta función permite evaluar los distintos estados del espacio de búsqueda para obtener alguna medida sobre lo prometedores que son, en cuanto a encontrar el objetivo se refiere. Uno de los principales focos de investigación en este área es precisamente el diseño de estas funciones heurísticas, y puede demostrarse para ciertos problemas que, si esta función cumple unas determinadas propiedades, el algoritmo encontrará la solución óptima. Un ejemplo de algoritmo de búsqueda heurístico es el algoritmo primero el mejor (GBFS), que selecciona en cada paso el nodo con el valor heurístico más bajo, es decir, el que parece estar más cerca del objetivo según la estimación. Sin embargo, este algoritmo no garantiza optimalidad, ya que puede quedar atrapado en soluciones subóptimas debido a que ignora el coste acumulado desde el nodo inicial.

El algoritmo A* fue propuesto en 1968 por Peter Hart, Nils Nilsson y Bertram Raphael [22] en el contexto del proyecto Shakey, un robot pionero en el campo de la inteligencia artificial con capacidad para razonar sobre sus propias acciones. En esta publicación introducen formalmente el uso de la información heurística

sobre el dominio de un problema en las estrategias de búsqueda, y proporcionan la definición de los conceptos de admisibilidad y optimalidad. Se dice que un algoritmo de búsqueda es admisible cuando garantiza encontrar la solución óptima, y es óptimo cuando la encuentra explorando el mínimo número de estados necesarios para llegar a dicha solución. Bajo este marco, demuestran la admisibilidad y optimalidad del algoritmo A* en términos de dos propiedades que debe cumplir la función heurística utilizada: la admisibilidad y la consistencia. Decimos que una heurística es admisible si nunca sobrestima el coste real, y que es consistente si el valor heurístico de cualquier estado es menor que el coste de moverse a un estado vecino más el valor heurístico del vecino si el estado solución es más cercano al vecino que al estado actual. En las Secciones 1.3.1 y 1.3.2 demostramos la admisibilidad y la optimalidad del algoritmo, y veremos cómo surgen naturalmente estas propiedades de la heurística.

## 1.3. El algoritmo A*

El algoritmo A* (A estrella) es un algoritmo de búsqueda informada que combina las ventajas de la búsqueda de coste uniforme y la búsqueda primero el mejor (velocidad mediante heurísticas). Su objetivo es encontrar el camino de menor coste desde un nodo inicial, $s$, hasta un nodo objetivo, $t$, en un grafo dirigido ponderado. El elemento diferenciador de este algoritmo es su función de evaluación, que se define como:

$$f(n) = g(n) + h(n) \tag{1}$$

donde $g(n)$ representa el coste real acumulado desde un nodo inicial hasta el nodo $n$ por un camino óptimo, y $h(n)$ el coste real para llegar al nodo objetivo desde el nodo $n$ por un camino óptimo. Si solo consideramos la función $g(n)$, el algoritmo A* sería equivalente al algoritmo de búsqueda de coste uniforme y, si solo consideramos $h(n)$, entonces es equivalente al algoritmo primero el mejor. El hecho de tener en cuenta ambas funciones hace que el algoritmo nos permita no solo encontrar la solución óptima, sino también de manera más eficiente.

El objetivo del algoritmo es encontrar el camino que minimiza el valor de $f$, pero no siempre podemos conocer con exactitud los valores de $h$ y $g$, por lo que tendremos que utilizar una estimación de estos, es decir:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n) \tag{2}$$

La propuesta de Hart et al. [22] para la elección de $\hat{g}(n)$ es el menor coste encontrado hasta el momento actual entre el nodo inicial y $n$. Esta elección implica que $\hat{g}(n) \geq g(n)$. La elección de $\hat{h}(n)$ es más complicada y requiere de un conocimiento explícito del dominio del problema. En algunos casos, como por ejemplo,

encontrar el camino entre dos puntos de un mapa, la distancia en línea recta sirve como estimación, pero en otros dominios puede no ser una elección trivial. De hecho, uno de los principales problemas que se presentan al querer aplicar este algoritmo en un problema real es determinar esta estimación de $h$.

La manera de proceder del algoritmo consiste en aplicar el operador de vecindad al estado actual para obtener sus estados vecinos y explorar aquel con un valor de $\hat{f}$ más pequeño. Para tener en cuenta aquellos estados que ya han sido visitados, y los que quedan por visitar, se utilizan dos estructuras de datos llamadas lista abierta, $Q$, y lista cerrada $H$ [1]. La lista abierta almacena aquellos estados a los que podemos llegar pero aún no hemos explorado, y la lista cerrada aquellos que ya han sido explorados y, además, sus vecinos se han introducido en la lista abierta. Decimos que un estado está *abierto* o *cerrado* si está en la lista abierta o cerrada, respectivamente.

El algoritmo, tal como se define en [22], opera de la siguiente manera:

1. Marcar $s$ como abierto y calcular $\hat{f}(s)$.
2. Seleccionar el nodo abierto $n$ con el mínimo valor de $\hat{f}(n)$.
3. Si $n = t$, marcar $n$ como cerrado y terminar el algoritmo.
4. Si no, marcar $n$ como cerrado y aplicar el operador $\Gamma$ a $n$. Calcular $\hat{f}(n)$ para cada vecino de $n$ y marcar como abiertos aquellos que no están marcados como cerrados. Remarcar como abiertos los sucesores ya marcados cerrados $n_i$ para los que el valor de $\hat{f}(n_i)$ es menor ahora que cuando fueron marcados cerrados.

A la hora de implementar el algoritmo, queda claro que son necesarias dos estructuras de datos principales, la lista abierta y cerrada. Para determinar la mejor implementación se deben revisar las principales operaciones que se realizan sobre ellas. Comenzando por la lista cerrada, al ser la encargada de almacenar los nodos expandidos, realizaremos la operación de inserción y, al expandir un nodo, debemos comprobar si cualquier nodo adyacente está en la lista, por lo que haremos una consulta (Paso 4). La estructura de datos más común es la tabla hash, que soporta ambas operaciones en tiempo constante en el mejor de los casos, esto es, cuando la función hash utilizada no provoca colisiones. El caso más sencillo y que podemos encontrar en algunas implementaciones es utilizar un array e identificar a cada nodo de forma unívoca por un número entero, que será su posición en el array. En cuanto a la lista abierta, es necesario extraer de ella el nodo con el menor valor $\hat{f}$, y también habrá que insertar los nuevos nodos que aún no han sido

[1] Las listas abierta y cerrada se llaman así por razones históricas. De hecho, no suelen implementarse como listas en la práctica.

expandidos (Paso 4). El hecho de que se extraigan los nodos de la lista abierta por orden de prioridad de $\hat{f}$ hace que la estructura de datos ideal sea una cola de prioridad. Según cómo se implemente la cola de prioridad, se obtienen órdenes de complejidad diferentes para las operaciones de inserción y extracción. Lo más habitual es implementar esta lista como un montículo binario, que permite realizar ambas operaciones en tiempo $O(\log n)$. Un montículo binario es un array en el que la información contenida mantiene una estructura de tipo árbol binario, de manera que cada nodo tiene un valor de prioridad menor (o mayor) que sus dos hijos. La forma de estructurar el array es la siguiente: cada nodo en la posición $n$ tendrá a sus hijos en las posiciones $2n$ y $2n+1$ si empezamos a contar las posiciones desde el uno, y en las posiciones $2n+1$ y $2n+2$ si empezamos a contar desde el cero (Figura 4).

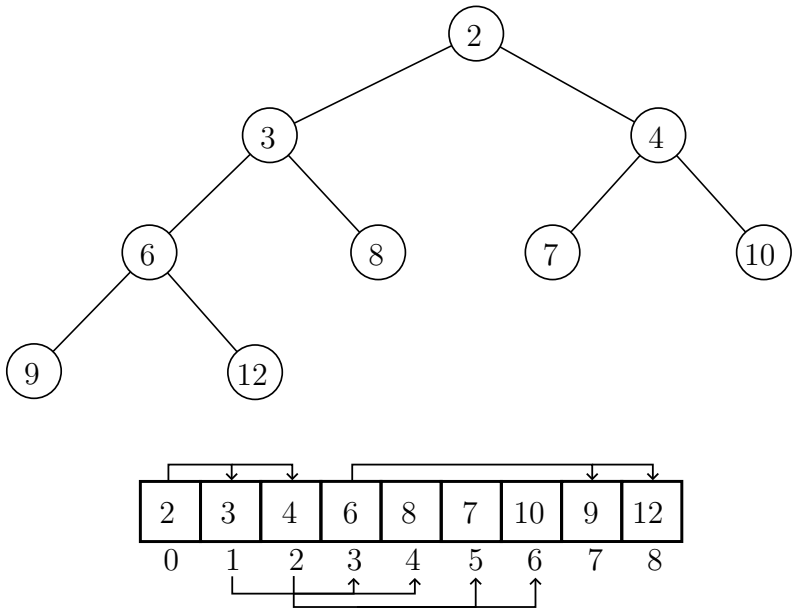



**Figura 4:** Representación en forma de árbol y array de un montículo binario. El valor representado en cada nodo indica su prioridad (coste $\hat{f}$) y las flechas apuntan a los hijos de cada nodo.

Aunque lo más habitual es implementar la cola de prioridad como un montículo binario, también se pueden encontrar otras estructuras de datos como montículos de Fibonacci [15]. Éstos, en particular, son útiles porque permite reubicar un nodo en el array en tiempo constante cuando se modifica su prioridad. A partir de ahora nos referiremos a la lista abierta también como la cola de prioridad de manera indistinta.

A continuación podemos ver el pseudocódigo del algoritmo. Los valores de $\hat{g}(n)$, $\hat{f}(n)$ y $parent(n)$ son los debemos almacenar para cada nodo. Las implementaciones más sencillas utilizan tres arrays distintos y prescinden de la lista

cerrada como estructura de datos en sí misma, pero lo más habitual es crear alguna estructura de datos que represente a los nodos, y almacenar en la lista cerrada los nodos con esos valores.

**Algoritmo 1** A*

```
1: Q ← {s};
2: H ← ∅;
3: while Q ≠ ∅ do
4:     n ← ExtractMin(Q);
5:     Insert(H, n);
6:     if n = t then
7:         return RetracePath(n);
8:     end if
9:     for all n' ∈ Neighbors(n) do
10:        g' ← ĝ(n) + c(n, n');
11:        if n' ∈ H then
12:            if g' < ĝ(n') then
13:                Remove(H, n');
14:                Insert(Q, n');
15:            else
16:                continue;
17:            end if
18:        else
19:            if n' ∉ Q then
20:                Insert(Q, n');
21:            else if g' ≥ ĝ(n') then
22:                continue;
23:            end if
24:        end if
25:        ĝ(n') ← g';
26:        f̂(n') ← g(n') + h(n');
27:        parent(n') ← n;
28:    end for
29: end while
30: return error Q is empty;
```

El algoritmo comienza creando la lista abierta, que incluye al nodo de inicio, y la lista cerrada, inicialmente vacía (líneas 1-2). Acto seguido entra en el bucle principal (líneas 3-29), que ejecutará mientras la lista abierta contenga elementos. Se extrae el nodo $n \in Q$ con menor coste $\hat{f}$ y se inserta en la lista cerrada (líneas 4-5). Si el nodo extraído es el objetivo, el algoritmo reconstruye el camino desde $n$ y termina su ejecución (líneas 6-8). En caso de no ser el objetivo, se expande

el nodo y se consideran sus vecinos (líneas 9-28). Se calcula el nuevo coste para alcanzar al vecino a partir del nodo actual $g'$ (línea 10) y se comprueba si ya ha sido cerrado. En ese caso, si el nuevo coste $g'$ mejora al coste anterior $\hat{g}(n')$, el nodo debe ser reabierto, esto es: se elimina de la lista cerrada, se inserta en la lista abierta (líneas 13-14) y se actualizan sus valores (líneas 15-17). En caso de no mejorar, podemos descartar el nodo y seguir (línea 16). Si el nodo vecino no ha sido cerrado, debemos comprobar si está o no en la lista abierta pendiente de expandir (línea 19). Si no está, se trata de un nuevo nodo, por lo que lo insertamos (línea 20) y actualizamos sus valores (líneas 25-27). Si está en lista abierta y el nuevo coste no mejora al anterior, podemos descartarlo (línea 22).

Cabe destacar que cuando la heurística es consistente, ningún nodo será reabierto, pues esta propiedad nos garantiza que cuando un nodo es cerrado, el coste óptimo para llegar a él ha sido encontrado, $\hat{g}(n) = g(n)$. Podemos encontrar otras versiones del pseudocódigo en las que se sustituye esta sección por un *continue* (líneas 12-17) cuando esta condición de consistencia está garantizada.

Al ser un algoritmo heurístico, la eficiencia de éste dependerá mucho de las característica de la función elegida como heurística. Cuanto más parecida sea la estimación al valor real de $h$, menos nodos expandirá el algoritmo. Sabemos que, como mucho, A* expandirá todos aquellos nodos con un valor $f(n) \leq C^*$, siendo $C^*$ el coste del camino óptimo. Dorian y Michie [10] proponen una medida llamada *penetrancia*, $P$, que se define como la longitud del camino óptimo, $L$, entre el número total de nodos generados durante la búsqueda, $T$, incluyendo al nodo objetivo pero no al inicial.

$$P = \frac{L}{T} \tag{3}$$

Cuando este valor es cercano a uno, quiere decir que el algoritmo ha expandido pocos nodos que no están en el camino óptimo y, cuando es cercana a cero, nos encontraremos que el algoritmo se comporta casi como una búsqueda no informada. Este valor depende tanto de la dificultad del problema como de la eficiencia del método de búsqueda. En la literatura más moderna en relación con la inteligencia artificial, se puede ver la eficiencia del algoritmo expresada en el peor caso como $O(b^d)$, donde $b$ es el factor de ramificación y $d$ la profundidad del nodo objetivo. En cuanto a la complejidad en espacio, A* guarda todos los nodos generados en la lista cerrada y los nodos por explorar en la lista abierta, y en cada iteración visita un nodo, por lo que la complejidad en espacio es también exponencial con respecto a la longitud del camino $d$.

En la práctica, éste es posiblemente el mayor problema que tiene el algoritmo, y ha llevado a desarrollar versiones limitadas en memoria para dominios de problema en los que el espacio de búsqueda crece de forma exponencial. Algunos de estos algoritmos son: IDA* [31], que realiza una búsqueda hasta una profundidad

limitada y que incrementa iterativamente si no encuentra la solución; SMA* [47], que elimina los nodos menos prometedres cuando la memoria está cerca de llenarse; RA* [13], que utiliza una operación llamada *retracción* para almacenar la información de los nodos de la frontera en sus padres, ahorrando memoria. Además de estas optimizaciones, como la eficiencia del algoritmo está íntimamente relacionada con la naturaleza del problema, se han creado a lo largo de los años otras versiones enfocadas a dominios de problema específicos: Anytime A* [20], Any-angle A* [6], D* [51] o LPA* [30].

Esta gran variedad de *sabores* diferentes del algoritmo a dado lugar a la aplicación y perfección de éste en dominios específicos. En el ámbito del desarrollo de videojuegos, es habitual representar el espacio de búsqueda como una rejilla [57, 21, 50, 1] y, además, a veces el camino debe buscarse para varios personajes [14]. Una de las variedades de A* que más se aplica en los videojuegos a día de hoy es LRTA* [3] ya que se trata de un algoritmo en tiempo real y que utiliza espacios de búsqueda locales en lugar del grafo completo para ahorrar memoria. En el artículo de [41] podemos ver como las estrategias basadas en heurísticas han ido sobreponiéndose a las basadas en algoritmos genéticos en los videojuegos. También en el campo de la robótica son útiles estas técnicas para resolver problemas como la planificación de rutas [37].

### 1.3.1. Admisibilidad de A*

En esta Sección vamos a discutir la admisibilidad del algoritmo, es decir, probaremos que dado un problema, encuentra la solución óptima. Nos basaremos en la prueba dada por uno de los autores del algoritmo en un libro publicado posteriormente [38, p. 59]. En la Sección 1.2 introducíamos el concepto de algoritmo admisible como aquel que garantiza encontrar la solución óptima. Esta propiedad está directamente ligada a las características de la función heurística y a la elección de la estimación $\hat{h}$. Más concretamente, probaremos que el algoritmo A* es admisible si $\hat{h}(n) \leq h(n)$ para todo $n$. Empezaremos demostrando el siguiente lema:

**Lema 1.** *Si $\hat{h}(n) \leq h(n)$ para todo $n$, entonces en cualquier momento antes de que A* finalice y para cualquier camino $P$ desde el nodo $s$ hasta el nodo $t$, existe un nodo abierto $n'$ en $P$ con $\hat{f}(n') \leq f(s)$.*

*Demostración.* Sea un camino óptimo $P = (s = n_0, n_1, n_2, \ldots, n_k = t)$. Debe haber al menos un nodo $n' \in P$ abierto, pues si $n_k$ fuera cerrado, el algoritmo habría terminado. Por definición de $\hat{f}$ tenemos:

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n)$$

Sabemos que A* ha encontrado el camino óptimo a $n'$, pues $n' \in P$ y todos sus antecesores en $P$ están cerrados. Por tanto $\hat{g}(n') = g(n')$ y:

$$\hat{f}(n') = g(n') + \hat{h}(n')$$

Como hemos asumido $\hat{h}(n) \leq h(n)$ podemos escribir:

$$\hat{f}(n') \leq g(n') + h(n') = f(n')$$

Pero el valor $f$ de cada nodo en un camino óptimo es igual a $f(s)$, el coste mínimo, y por tanto $\hat{f}(n') \leq f(s)$. ■

Podemos probar ahora que el algoritmo es admisible si se cumple que $\hat{h}$ es una estimación optimista de $h$.

**Teorema 1.** *Si $\hat{h}(n) \leq h(n)$ para todos los nodos $n$, y si todos los costes de los arcos son mayores o iguales que un número positivo $\delta$, entonces el algoritmo A∗ es admisible.*

*Demostración.* Probaremos este teorema por reducción al absurdo. Asumiremos que el algoritmo termina sin encontrar el camino óptimo. Entonces hay tres casos a considerar: termina sin encontrar el nodo objetivo, no termina nunca o termina pero no encuentra el camino de coste mínimo.

*Caso 1:* Termina sin encontrar el nodo objetivo. Este caso contradice la condición de terminación del algoritmo (Paso 3), por lo que no se considera. El único otro caso en el que puede terminar el algoritmo se da cuando la lista abierta está vacía, pero por el Lemma 1 sabemos que esta situación no puede darse si existe un camino entre el nodo inicial y el final.

*Caso 2:* No termina nunca. Supongamos un nodo objetivo $t$ accesible desde un nodo inicial $s$ en un número finito de pasos y con coste asociado $f(s)$. Dado que el valor de cada arco es positivo y mayor que cero, entonces para cada nodo $n$ a $M = f(s)/\delta$ pasos de $s$ se tiene $\hat{f}(n) \geq \hat{g}(n) \geq g(n) > M\delta = f(s)$. De esta expresión podemos ver que ningún nodo $n$ a más de $M$ pasos del nodo inicial $s$ será expandido nunca, dado que por el Lemma 1, habrá un nodo $n'$ en la lista abierta con $\hat{f}(n') \leq f(s) < \hat{f}(n)$, y A* elegirá a $n'$ antes que a $n$. Aun así, el A* podría no terminar debido a reabrir continuamente nodos a menos de $M$ pasos de $s$. Sea $\chi(M)$ el conjunto de nodos accesibles desde $s$ en $M$ pasos, y sea $\nu(M)$ el número de nodos en $\chi(M)$. Cada nodo $n \in \chi(M)$ podrá ser reabierto a lo sumo un número finito de veces $\tilde{\rho}(n, M)$, dado que hay un número finito de caminos de $s$ a $n$ pasando solamente por nodos a $M$ pasos de $S$. Sea

$$\rho(M) = \max_{n \in \chi(M)} \tilde{\rho}(n, M)$$

El máximo número de veces que un nodo puede reabrirse. Por tanto, tras como mucho $\nu(M)\rho(M)$ expansiones, todos los nodos en $\chi(M)$ habrán sido cerrados. Dado que ningún nodo fuera de $\chi(M)$ será nunca expandido, A* termina.

*Caso 3:* Termina pero no encuentra el camino de coste mínimo. Supongamos que A* termina en algún nodo objetivo $t$ con $\hat{f}(t) = \hat{g}(t) > f(s)$. Por el Lemma 1, habrá un nodo en la lista abierta $n'$ con $\hat{f}(n) \leq f(s) < \hat{f}(t)$, por lo que A* elegirá a $n'$ antes que a $t$, contradiciendo que el algoritmo terminaría en un camino no óptimo.

■

La prueba del Teorema 1 queda terminada. Hemos completado la prueba del Teorema 1 y podido comprobar, como anunciábamos al comienzo de la sección, que cuando la estimación de la heurística es un límite inferior de ésta, $\hat{h}(n) \leq h(n)$, para todo $n$, entonces A* es admisible, y encuentra siempre la solución óptima.

### 1.3.2. Optimalidad de A*

La optimalidad de A* fue una cuestión que condujo a confusión, y hubo que esperar al desarrollo de un marco de trabajo más completo para algoritmos de búsqueda informada. Como se ha mencionado anteriormente, decimos que un algoritmo de búsqueda es óptimo cuando expande el mínimo número de nodos necesarios para encontrar la solución óptima. La primera prueba de la optimalidad de A* la ofrecieron Hart et al. [22], que posteriormente sintetizaron en el libro de Nilsson [38]. En su prueba, imponían la condición de consistencia a la función heurística. Esto es, dado un nodo $n$ y un nodo $n'$ tal que $h(n) < h(n')$, entonces debe cumplirse:

$$\hat{h}(n) < c(n, n') + \hat{h}(n') \tag{4}$$

Años más tarde, publicaron una corrección [23] en la que sostenían que la condición de consistencia no era necesaria, pues no hacían uso de ella en su demostración, pero fueron refutados por Dechter y Pearl [8, 7]. Se basaron en la observación de Gelperin [18], quien había notado que para discutir sobre la optimalidad del algoritmo había que compararlo con una clase más amplia de algoritmos igualmente informados, no solo aquellos que hacían uso de la heurística en la forma $f = g + h$. Dechter y Pearl se dieron cuenta de que la definición de Gelperin de *igualmente informado* no incluía a aquellos algoritmos que, teniendo la misma información disponible que A*, pudieran hacer un mejor uso de ésta. El resultado de su trabajo es una jerarquía de cuatro tipos de optimalidad para tres tipos de algoritmos y cuatro dominios de problemas. No entraremos en los detalles de las demostraciones por la complejidad que presentan en comparación

con la prueba de admisibilidad de algoritmo, pero vamos a repasar los resultados Dechter y Pearl, así como las conclusiones más notables.

Una instancia de un problema se define como una cuádrupla $I = (G, s, \Gamma, \hat{h})$, donde $G$ es el grafo, $s$ el nodo inicial, $\Gamma$ el operador de vecindad y $\hat{h}$ las estimaciones de la heurística. Dechter y Pearl proponen los siguientes conjuntos de instancias relacionados con la admisibilidad y la consistencia de la heurística:

$$\begin{aligned} \mathbf{I}_{AD} &= \{(G, s, \Gamma, h) \,|\, \hat{h} \leq h \text{ en } (G, \Gamma)\} \\ \mathbf{I}_{CON} &= \{(G, s, \Gamma, h) \,|\, \hat{h} \text{ es consistente en } (G, \Gamma)\} \end{aligned} \tag{5}$$

Se puede demostrar que una heurística consistente es a su vez admisible, por lo que $\mathbf{I}_{CON} \subseteq \mathbf{I}_{AD}$. Además, de cada conjunto se toma un subconjunto especial denominado *instancias no patológicas* que aquí no tomaremos en cuenta. Al conjunto de algoritmos que encuentran la solución óptima cuando hacen uso de una heurística admisible lo denotaremos como $\mathbf{A}_{ad}$. De esta clase de algoritmos se estudian dos subclases: $\mathbf{A}_{gc}$ representa a los algoritmos globalmente compatibles, esto es, aquellos que encuentran una solución óptima cuando A* lo hace, incluso cuando $\hat{h} > h$. $\mathbf{A}_{bf}$ representa a los algoritmos de búsqueda primero el mejor admisibles. En la Figura 5 se pueden observar los diagramas de Venn correspondientes a los tipos de instancias de problemas y a las clases de algoritmos.

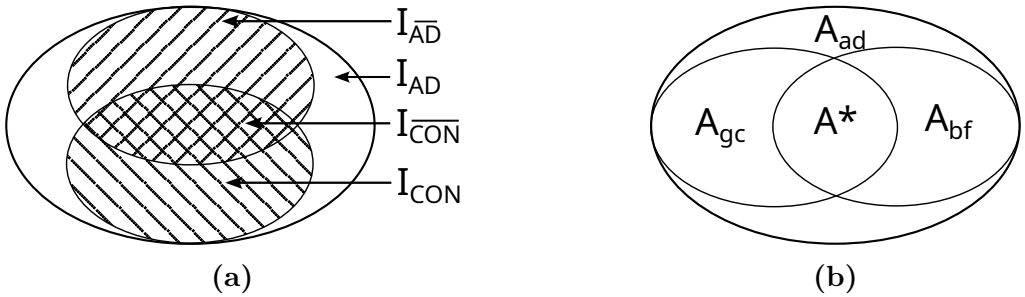



**Figura 5:** Diagramas de Venn de las instancias de problemas (a) y las clases de algoritmos considerados (b) [8].

Otro concepto necesario para definir la optimalidad es el de dominancia [39]:

**Definición 6.** *Decimos que un algoritmo A domina a otro algoritmo B en un conjunto de instancias de problemas* $\boldsymbol{I}$ *si y solo si, para cada instancia* $I \in \boldsymbol{I}$*, el conjunto de nodos expandido por A es un subconjunto de los nodos expandidos por B. A domina estrictamente a B si y solo si A domina a B y B no domina a A. Esto es, existe al menos una instancia donde A no expande un nodo que B sí, y no existe ninguna instancia donde ocurra lo contrario.*

Según este concepto de dominancia, se define la optimalidad de un algoritmo:

**Definición 7.** *Un algoritmo A es óptimo sobre una clase* $\boldsymbol{A}$ *de algoritmos si y solo si A domina a cada miembro de* $\boldsymbol{A}$.

Con esta definición de optimalidad podemos clasificar la optimalidad de un algoritmo sobre otra clase de algoritmos con respecto a un tipo de instancias de problemas, pero debemos recordar que A* no es un único algoritmo, sino una familia de algoritmos diferenciados por lo que se conoce como regla de desempate. Esta regla de desempate es una forma de elegir el próximo nodo a expandir si en la lista abierta hay más de un nodo con el mismo valor de $f$. Teniendo esto en cuenta, las cuatro clases de optimalidad definidas para A* son las siguientes:

- *Tipo 0.* A* es 0-óptimo sobre **A** en **I** si y solo si, en cada instancia de problema, $I \in \mathbf{I}$, cualquier regla de desempate expande un subconjunto de los nodos expandidos por cualquier miembro de **A**.
- *Tipo 1.* A* es 1-óptimo sobre **A** en **I** si y solo si, para cada instancia de problema, $I \in \mathbf{I}$, existe al menos una regla de desempate que expande un subconjunto de los nodos expandidos por cualquier miembro de $A$.
- *Tipo 2.* A* es 2-óptimo sobre **A** en **I** si y solo si no existe ninguna instancia de problema $I \in \mathbf{I}$ para la que algún miembro de **A** expande un subconjunto propio de los nodos expandidos por cualquier regla de desempate en A*.
- *Tipo 3.* A* es 3-óptimo sobre **A** en **I** si y solo si se cumple que: Si existe una instancia de problema $I_1 \in \mathbf{I}$ donde algún algoritmo $B \in \mathbf{A}$ no expande algún nodo expandido por alguna regla de desempate en A*, entonces existe alguna instancia $I_2 \in \mathbf{I}$ donde esa regla de desempate no expande un nodo expandido por B.

| | $\mathbf{A}_{ad}$ | $\mathbf{A}_{gc}$ | $\mathbf{A}_{bf}$ |
|---|---|---|---|
| $\mathbf{I}_{AD}$ | No óptimo | 1-óptimo | 1-óptimo |
| $\mathbf{I}_{CON}$ | 1-óptimo | 1-óptimo | 1-óptimo |

**Tabla 1:** Tipo de optimalidad de A* sobre otras clases de algoritmos en distintas instancias de problemas [8].

En la Tabla 1 podemos observar el tipo de optimalidad de A* en comparación con las tres clases de algoritmos en los cuatro dominios de problemas. Los resultados más destacables se concentran en la primera columna: A* no es óptimo

sobre $\mathbf{A}_{ad}$ para aquellos problemas en los que la heurística solo es admisible, es más, ningún algoritmo óptimo puede existir a menos que la heurística sea también consistente. Recordemos por los tipos de optimalidad que esto quiere decir que pueden existir algoritmos que consiga superar a A* en algunas instancias de esta clase de problemas. Por la segunda y tercera columna podemos ver que A* sí que es óptimo sobre las clases de algoritmos $\mathbf{A}_{gc}$ y $\mathbf{A}_{bf}$.

# 2. Estado del arte

El algoritmo A* es eficaz para encontrar caminos óptimos en grafos, pero su rendimiento se ve limitado cuando se aplica a problemas de gran escala, debido a la gran cantidad de memoria que requiere para mantener la información de los nodos visitados. Para mitigar este problema, además de las modificaciones limitadas en memoria de la versión secuencial, se han propuesto diversas estrategias para paralelizar el algoritmo.

Las primeras propuestas para paralelizar un algoritmo de búsqueda aparecen en las décadas de los 70 y 80. Wyllie [56] propone en su tesis doctoral paralelizar la estrategia de búsqueda en anchura y, aunque otros autores desestimaron su idea afirmando que no era posible [11, 44, 45], a principios de la década de los 80 comenzaron a surgir avances en éste área y que podemos encontrar en orden cronológico en el artículo de Freeman [16], demostrando la viabilidad de la idea. Aunque estas ideas no incorporaban la información heurística en el problema como A*, sentaron las bases para muchos de los principios que se aplicarían después. Rao y Kumar [42, 33] también trabajaron con la búsqueda en profundidad y, como resultado de su trabajo, unieron la búsqueda en profundidad con el algoritmo IDA*, consiguiendo una aumento en la velocidad superlineal [43].

Estos primeros trabajos se enfocaban en dividir el espacio de búsqueda entre distintos procesadores, cada uno de los cuales exploraba una región diferente del grafo o árbol de búsqueda. En general, se trataba de problemas en los que el espacio de estados se puede representar como una estructura tipo árbol. De ésta manera, los subárboles generados a partir de cada nodo son independientes, y la paralelización es casi inmediata. Sin embargo, en la mayoría de aplicaciones, los subárboles generados son irregulares, provocando un desbalanceo en la carga de trabajo entre los procesadores.

Antes de abordar las técnicas concretas de paralelización de A*, es necesario introducir brevemente los principales modelos de computación paralela que, además, representan fielmente las dos principales estrategias con las que se intenta paraleliza el algoritmo. En primer lugar, tenemos el modelo de memoria compartida, en el que varios hilos se ejecutan en paralelo y acceden a una memoria común, lo que facilita el intercambio de información pero requiere mecanismos de sincronización para evitar condiciones de carrera. Por otro lado, tenemos el modelo de memoria distribuida, que se basa en procesos que no comparten memoria y se comunican entre sí mediante el envío de mensajes. Este segundo modelo implica un diseño más explícito de la coordinación entre procesos, pero ofrece una mejor escalabilidad en arquitecturas con muchos núcleos o nodos.

Al diseñar algoritmos de búsqueda paralelos, es importante tener en cuenta tres problemas principales: la sobrecarga de búsqueda, que se refiere a los estados que se generan en paralelo pero que no serían explorados en una búsqueda

secuencial; la sobrecarga de comunicación, que surge al distribuir el espacio de búsqueda entre distintos hilos; y la sobrecarga de coordinación, que tiene que ver con el coste de sincronizar correctamente los hilos durante la ejecución. Las diferentes estrategias que se han propuesto en la literatura se diferencian principalmente en cómo tratan estos problemas. Kumar [32] propone dos enfoques para paralelizar las búsquedas primero el mejor, diferenciados por cómo usan y gestionan la lista abierta entre los diferentes procesos. En la revisión de Fukunaga [17] encontramos una imagen que se ha traducido al castellano para este trabajo y que representa muy bien las distintas estrategias y a qué algoritmos han dado lugar (Figura 6).

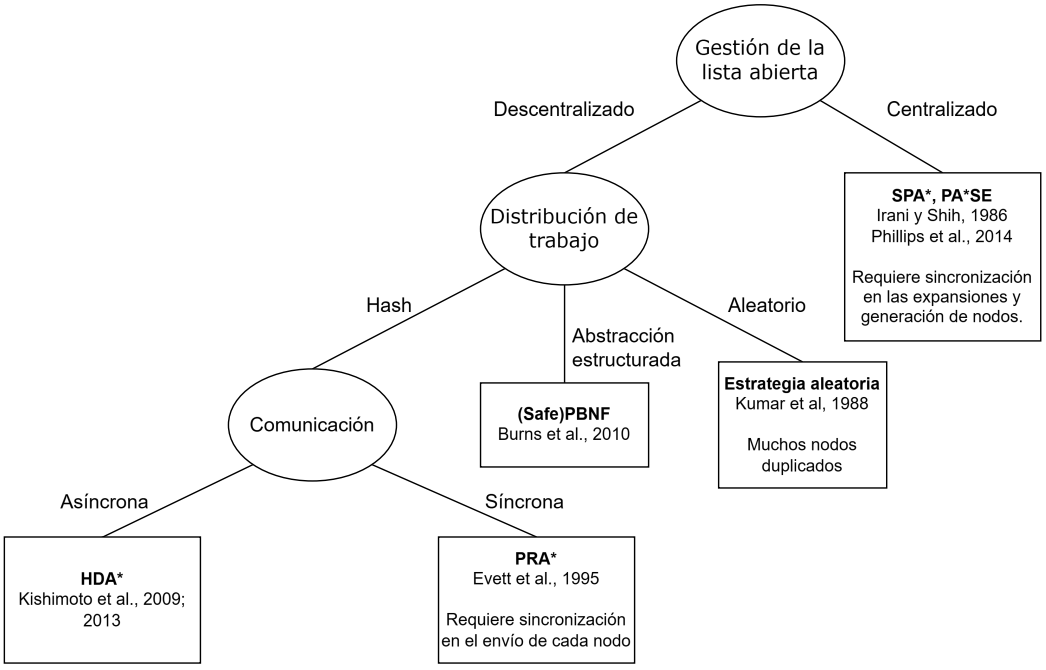



**Figura 6:** Clasificación de las distintas estrategias de paralelización [17].

## 2.1. A* centralizado

La idea más sencilla para paralelizar A* es extraer tantos nodos de la lista abierta como procesos estén participando en la búsqueda. Esta estrategia es conocida como la estrategia centralizada, pues tanto la lista abierta como la cerrada se mantiene en una memoria global compartida por todos los procesos (Figura 7). Este enfoque presenta dos desafíos principalmente, el primero es que el criterio de terminación del algoritmo secuencial deja de ser válido. No podemos asegurar que cuando un nodo es extraído de a lista abierta, éste sea la mejor solución, pues puede haber otro proceso que en su próxima expansión encuentra una solución mejor. Esto ocurre cuando hay más de un nodo objetivo, y el criterio de

terminación puede modificarse para que el algoritmo termine únicamente cuando encuentra la mejor solución [35]. Por otro lado, los continuos accesos a la lista abierta generan una zona de contención entre los hilos, ya que ésta debe protegerse mediante un cerrojo para evitar condiciones de carrera. Esta contención limitaría el *speedup* obtenido en la versión paralela a $(T_{exp}+T_{access})/T_{access}$, donde $T_{exp}$ es el tiempo promedio para una expansión, y $T_{access}$ es el tiempo promedio accediendo a la lista abierta por cada expansión [24]. También es necesario proteger la lista cerrada, pues diferentes procesos pueden estar leyendo y escribiendo a la vez para comprobar si el nodo ya ha sido visitado o si es un nodo nuevo. Lo más habitual es establecer un cerrojo por nodo, de manera que no se bloquee la estructura entera para el resto de procesos que puedan estar expandiendo otros nodos diferentes.

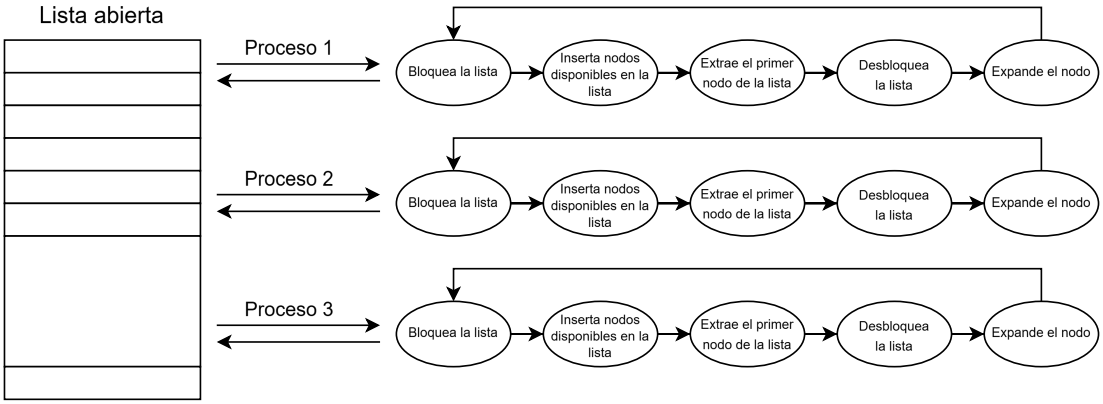



**Figura 7:** Estrategia centralizada con lista abierta compartida en memoria global. Cada proceso bloque la lista, inserta los nodos expandidos en la iteración anterior, extrae el de menor coste y la desbloquea para el siguiente proceso [19].

Este algoritmo recibe el nombre de *Simple Parallel A** o SPA* (Algoritmo 2). En el pseudocódigo no se muestran los cerrojos de la lista cerrada y una variable atómica extra que es necesaria para detectar la terminación. Se comentarán estas diferencias a lo largo de la explicación, pero se ha decidido no incluirlas para no hacerlo excesivamente largo y difícil de entender a simple vista.

La estructura general es similar a A*, inicializa la lista abierta con el nodo inicial y la lista cerrada vacía, pero ahora cuenta con un cerrojo para la lista abierta y otro para $m$, que representa la mejor solución encontrada hasta ahora (líneas 1-4). En esta parte también se inicializarían los cerrojos de la lista cerrada y el contador atómico. El bucle principal del algoritmo se ejecuta en paralelo para cada proceso $p$ (líneas 6-41). En esta versión no podemos terminar el algoritmo cuando $Q \neq \emptyset$ (línea 7), ya que puede darse el caso de que la lista esté vacía y otro proceso esté expandiendo un nodo sin haber introducido todavía los vecinos en la lista, algo que suele ocurrir muy a menudo al comienzo del algoritmo. Además, se

debe comprobar también que el próximo nodo no empeora el coste del objetivo actual, pues se harían expansiones innecesarias. El valor estimado para $m$ cuando aún no hemos encontrado el objetivo debería ser un valor lo suficientemente alto, esto es, $\hat{f}(\text{NULL}) = \infty$. Cuando se cumple alguna de las condiciones, se debe comprobar que ningún proceso puede descubrir una mejor solución. Para detectar la correcta terminación del algoritmo se utiliza variable atómica que cada proceso aumenta cuando se cumple alguna de las condiciones y que vuelve a decrementar si no se cumplen en alguna iteración posterior. Al inicio de cada iteración del bucle, TerminateDetection() se encarga de comprobar si el valor de esta variable es igual al número de procesos, en cuyo caso se puede dar por terminado el algoritmo.

A continuación los procesos se disponen a extraer el siguiente nodo de la lista abierta, protegiendo el acceso con un cerrojo para evitar condiciones de carrera (líneas 10-12). La inserción en la lista cerrada (línea 13) puede hacerse sin ningún cerrojo, ya que cada proceso habrá extraído un nodo diferente. La comprobación del objetivo es similar a la de la versión secuencial, pero protegiendo la variable $m$ con un cerrojo (líneas 14-20). El resto del algoritmo, es decir, la expansión del nodo actual, se realiza igual que en el algoritmo secuencial (líneas 21-41), pero en este caso, tanto los accesos a $Q$ como a $H$ deben protegerse. Puede darse el caso de que dos procesos obtengan el mismo vecino desde dos nodos diferentes, siendo necesaria la sincronización a la hora de comprobar cuál de los dos debe actualizar los valores del nodo si éste está cerrado y se han encontrado mejores caminos. Cuando el bucle principal termina, se comprueba si se ha encontrado el nodo objetivo y, si es el caso, se devuelve el camino (líneas 42-46).

SPA* es muy sencillo de implementar en un modelo de memoria compartida, pero ha probado ser poco eficiente [34], incluso peor que la versión secuencial para problemas en los que la operación de expansión de un nodo es rápida, debido a que la mayor parte del tiempo de ejecución de los procesos se desperdicia mientras esperan a obtener un nodo de la lista abierta. Aun así, ha servido de base para otras propuestas como PA*SE [40], que introduce un mecanismo para evitar reexpansiones innecesarias en SPA*, o PKBFS [54], que realiza la expansión paralela de los $k$ mejores nodos en cada iteración.

**Algoritmo 2** Simple Parallel A* (SPA*)

```
1: Q ← {s};
2: H ← ∅;
3: Lock l_o, l_m;
4: m ← NULL;
5: En paralelo, para cada proceso p, ejecutar:
6: while TerminateDetection() do
7:     if Q = ∅ or PeekMin(Q) ≥ f̂(m) then
8:         continue;
9:     end if
10:    AcquireLock(l_o);
11:    n ← ExtractMin(Q);
12:    ReleaseLock(l_o);
13:    Insert(H, n);
14:    if n = t then
15:        AcquireLock(l_m);
16:        if ĝ(n) < ĝ(m) then
17:            m ← n;
18:        end if
19:        ReleaseLock(l_m);
20:    end if
21:    for all n' ∈ Neighbors(n) do
22:        g' ← ĝ(n) + c(n, n');
23:        if n' ∈ H then
24:            if g' < ĝ(n') then
25:                Remove(H, n');
26:                Insert(Q, n');
27:            else
28:                continue;
29:            end if
30:        else
31:            if n' ∈ Q then
32:                Insert(Q, n');
33:            else if g_1 ≥ ĝ(n') then
34:                continue;
35:            end if
36:        end if
37:        ĝ(n') ← g';
38:        f̂(n') ← ĝ(n') + ĥ(n');
39:        parent(n') ← n;
40:    end for
41: end while
42: if m = NULL then
43:    return error
44: else
45:    return RetracePath(m);
46: end if
```

## 2.2. A* descentralizado

La estrategia descentralizada surge para aliviar la sobrecarga debida a los accesos constantes a las listas abierta y cerrada en SPA*. En contraste con la estrategia centralizada, cada proceso tiene su propia lista abierta. Al inicio, el proceso padre expande el primer nodo y reparte los sucesores entre el resto de procesos hijos, que empezará una búsqueda en cada uno de ellos. Al descentralizar la lista abierta, se eliminan las condiciones de carrera al intentar acceder a una estructura global, pero surge la necesidad de balancear la carga de trabajo.

En una estrategia distribuida, cada proceso extrae un nodo de su lista local. Cuando el espacio de búsqueda tiene forma de árbol, esta estrategia divide el trabajo de forma equitativa entre los procesos, pero puede llevar a que alguno de ellos explore una parte del árbol que hubiese sido descartada por la versión secuencial. Varios autores han propuesto estrategias de comunicación para repartir los nodos entre los procesos y aliviar esta sobrecarga en la búsqueda, lo que representa el clásico intercambio entre computación y comunicación en las arquitecturas distribuidas. Cuando el espacio de búsqueda es un grafo, además, surge el problema de detectar la duplicación de nodos, es decir, dos procesos pueden obtener el mismo nodo como resultado de la expansión de dos nodos diferentes.

En una estrategia descentralizada (Algoritmo 3), cada proceso cuenta con su propia lista abierta (línea 1), cerrada y un *buffer* de recepción. La función ComputeRecipient($n$) devuelve el índice del proceso al que debe mandarse el nodo $n$. Al entrar en el bucle principal (líneas 5-42), cada proceso comprueba si hay algún elemento en el *buffer*. Extrae la información sobre el nodo, su nuevo coste y su padre (línea 7). A continuación comprueba si es un duplicado, si mejora el coste anterior, si es un nuevo nodo o si se puede descartar, al igual que en las otras versiones centralizada y secuencial (líneas 8-24). Una vez vaciado el *buffer*, comprueba si debe terminar (líneas 26-28). Extrae un nodo de su lista abierta y lo inserta en su lista cerrada (líneas 29-30). Si es un nodo objetivo, actualiza el valor de $m$ (líneas 31-37). En caso contrario, expande el nodo, calculando el nuevo coste para alcanzarlo y enviando al propietario la información.

Gran parte de la eficiencia de estas estrategias depende de la función ComputeRecipient(n) y de cómo reparta los nodos entre procesos. Karp y Zhang [26, 27] proponen una estrategia de distribución de trabajo aleatoria en la que cada vez que se obtiene un nodo al expandir, se calcula a qué proceso debe asignarse. Kumar et al. [32] proponen otro tipo de estrategia de distribución aleatoria a la que denominan *blackboard*, en la que los procesadores comparten información a través de un estructura común. Otro ejemplo de estrategia distribuida es la denominada búsqueda bidireccional, que consiste en la ejecución del algoritmo de búsqueda desde el nodo inicial hacia el final y desde el nodo final hacia el inicial [46].

**Algoritmo 3** A* descentralizado

```
1: Q ← {Q_i}_{i=1}^p;
2: INSERT(Q_{COMPUTERECIPIENT(s)}, s);
3: m ← NULL;
4: En paralelo, para cada proceso p, ejecutar:
5: while TERMINATEDETECTION() do
6:     while BUFFER_P ≠ ∅ do
7:         (n', g', n) ← EXTRACT(BUFFER_p);
8:         if n' ∈ H_p then
9:             if g' < ĝ(n') then
10:                REMOVE(H_p, n');
11:                INSERT(Q_p, n');
12:            else
13:                continue;
14:            end if
15:        else
16:            if n' ∉ Q_p then
17:                INSERT(Q_p, n');
18:            else if g' ≥ ĝ(n') then
19:                continue;
20:            end if
21:        end if
22:        ĝ(n') ← g';;
23:        f̂(n') ← ĝ(n') + ĥ(n');
24:        parent(n') ← n;
25:    end while
26:    if Q_p = ∅ or PEEKMIN(Q_p) ≥ f̂(m) then
27:        continue;
28:    end if
29:    n ← EXTRACTMIN(Q_p);
30:    INSERT(H_p, n);
31:    if n = t then
32:        AcquireLock(l_m)
33:        if ĝ(n) < ĝ(m) then
34:            m ← n;
35:        end if
36:        ReleaseLock(l_m);
37:    end if
38:    for all n' ∈ NEIGHBORS(n) do
39:        g' ← ĝ(n) + c(n, n');
40:        INSERT(BUFFER_{COMPUTERECIPIENT(n')}, (n', g', n));
41:    end for
42: end while
43: if m = NULL then
44:    return ERROR
45: else
46:    return RETRACEPATH(m);
47: end if
```

El problema con las estrategias aleatorias es que los nodos duplicados solo son detectados si se mandan fortuitamente al mismo proceso. Otro enfoque más simple para distribuir la carga de trabajo entre los procesos fue introducido por Evett et al. [13]. En su artículo presentan PRA*, una versión paralela de A* distribuida en la que el reparto de nodos entre los procesos se realiza mediante una función *hash* que asigna cada nodo a único proceso. Al expandir un nodo, cada proceso debe calcular a qué proceso debe mandar cada sucesor de manera síncrona. Cada proceso cuenta con un *buffer* que el resto deben bloquear cada vez que envían un nodo. Esta idea de asignar el trabajo a los procesos mediante una función *hash* también fue usada por Mahapatra [36] para paralelizar una versión de A* llamada SEQ_A*, pero tanto Evett et al. como Mahapatra no explorar esta técnica lo suficiente en sus trabajos, ya que se centraban en otros aspectos del problema, como la retracción de nodos en el caso de Evett.

## 2.3. Hash Distributed A*

Kishimoto, Fukunawa y Botea [28, 29] retoman esta idea de repartir los nodos entre procesos mediante una función *hash* años mas tarde y proponen HDA* (Hash-Distributed A*), una versión simple, paralela y escalable de A*, en la que a cada proceso le pertenece una parte del espacio de búsqueda. HDA* opera de manera similar a la expuesta en el Algoritmo 3. La principal diferencia con la propuesta de PRA* es que la comunicación entre los procesos se realiza de manera asíncrona, lo que evita tener que bloquear los *buffers* de recepción del resto de procesos se le deben enviar nuevos nodos. Las primeras implementaciones de PRA* mostraban un rendimiento muy bajo, incluso peor que la versión secuencial, en problemas como la búsqueda en *grids* [4] debido a la sobrecarga introducida por los bloqueos. En su lugar, en HDA*, la comunicación entre procesos se implementa mediante paso de mensajes incluso en entornos de memoria compartida.

Los primeros trabajos sobre HDA* no evaluaron de forma cuantitativa el impacto que tiene la elección de la función *hash*, lo que llevó a que investigaciones posteriores arrojaran resultados poco precisos. Posteriormente, Jinnai y Fukunaga [25] realizaron una comparación entre diferentes funciones de dispersión utilizadas en la literatura, demostrando que tanto la función *hash* de Zobrist como su versión mejorada, conocida como Abstract Zobrist hashing, superan con claridad a otras alternativas empleadas anteriormente.

# 3. Objetivos

El objetivo de este trabajo es implementar y evaluar tres versiones del algoritmo A*: una versión secuencial y dos versiones paralelas basadas en las estrategias centralzada y descentralizada que se han explicado en la Sección 2.

# 4. Metodología

La segunda mitad de este trabajo consiste en el desarrollo y la evalución de distintas versiones del algoritmo A*, en concreto, una versión secuencial, una versión centralizada y una versión descentralizada, que se evaluarán en distintos escenarios. El objetivo principal es obtener una mejora en la versiones paralela con respecto a la versión secuencial. Todo el código se desarrollará en el lenguaje de programación C, y se compilará con la herramienta `gcc`. Para implementar el paralelismo se ha utilizado OpenMP. La versión centralizada estará basada en SPA*, y la descentralizada en HDA*. Aunque HDA* está pensado para implementarse en un entorno de memoria distribuida mediante MPI, en este trabajo se propone una implementación con OpenMP que simula la comunicación asíncrona entre hilos.

# 5. Diseño e implementación

En esta sección se explicará en detalle el proceso seguido para obtener tanto la versión secuencial como las versiones paralelas a partir de ésta. Finalmente, realizaremos una comparación entre los tres tipos de implementaciones en distintos escenarios y analizaremos los tiempos obtenidos.

La implementación cuenta con los siguientes ficheros:

- `astar.h`: Cabecera con las estructuras y métodos comunes a las tres implementaciones. También define el método para iniciar la versión secuencial.
- `astar.c`: Implementación secuencial.
- `spastar.h`: Cabecera que define el método para iniciar SPA*.
- `spastar.c`: Implementación de SPA*.
- `hdastar.h`: Cabecera que define las estructuras necesarias para la versión descentralizada. También define el método para iniciar HDA*.
- `hdastar.c`: Implementación de HDA*.

Cabe mencionar antes de comenzar con las implementaciones una modificación que es muy frecuente encontrar en las implementaciones más modernas de A*. Se trata de utilizar una lista de visitados en lugar de una lista cerrada. La diferencia fundamental es que cuando un nodo es reabierto, no se elimina de la lista de visitados. Esta situación, que se da cuando la heurística es admisible pero no consistente, conlleva en la versión original eliminar al nodo de la lista cerrada e insertarlo de nuevo en la lista abierta. El problema que esto presenta es que si el nodo no está cerrado pero sí está abierto, la operación de buscarlo en la lista abierta es bastante costosa: $O(n)$ si se implementa como un montículo binario. La solución pasa por no eliminar el nodo de la lista cerrada y añadir un campo a la estructura que representa a los nodos que nos indique si el nodo está abierto o no. Al no eliminar nunca los nodos, esta pasa a llamarse lista de visitados.

Esta modificación conlleva un cambio en el pseudocódigo del Algoritmo 1. En concreto, no se eliminaría el nodo de la lista cerrada en la línea 13 y las líneas 18 a 23 serían sustituidas por la inserción del nodo en la lista de visitados y en la lista de abiertos. Se puede ver esta modificación en el Algoritmo 4, donde $V$ representa la lista de visitados.

**Algoritmo 4** Modificación de A* (Algoritmo 1, líneas 9-28)

```
 9: for all n' ∈ NEIGHBORS(n) do
10:     g' ← ĝ(n) + c(n, n');
11:     if n' ∈ V then
12:         if g' < ĝ(n') then
13:             INSERT(Q, n');
14:         else
15:             continue;
16:         end if
17:     else
18:         INSERT(V, n')
19:         INSERT(Q, n');
20:     end if
21:     ĝ(n') ← g';
22:     f̂(n') ← g(n') + h(n');
23:     parent(n') ← n;
24: end for
```

## 5.1. Implementación secuencial

En primer lugar veremos la implementación de la versión secuencial. Empezaremos por las estructuras de datos utilizadas. Para representar a los nodos hemos utilizado una estructura `node_t` que podemos encontrar en `astar.h`:

```
struct {
    int id;
    float gCost;
    float fCost;
    int parent;
    int open_index;
} node_t;
```

Cuenta con los valores de $\hat{g}(n)$, $\hat{f}(n)$ y $parent(n)$ que hemos visto antes y, además, con un identificador que nos permite diferenciar a los nodos y un campo `open_index` que nos indica la posición en la lista abierta. Este valor será $-1$ cuando el nodo no esté abierto. Para iniciar los algoritmos es necesario definir una estructura `AstarSource`. Esta estructura cuenta con tres parámetros: el número máximo de nodos en el espacio de búsqueda, `max_size`, y dos punteros a funciones. La primera de ellas recibe como parámetros los identificadores de dos nodos y devuelve la estimación de la distancia entre ellos, la heurística. La segunda función se encarga de calcular los vecinos. Ambas deben ser implementadas por el usuario para el dominio específico en el que se quiera ejecutar los algoritmos.

```
struct {
    int max_size;
    float (*heuristic)(int id_1, int id_2);
    void (*get_neighbors)(neighbors_list *neighbors, int id);
} AStarSource;
```

Al definir la función `get_neighbors`, el usuario debe añadir los nodos vecinos con la función:

```
void add_neighbor(neighbors_list *neighbors, int n_id, float
    cost);
```

indicando el índice del vecino, `n_id`, y el coste para alcanzarlo,`cost`. La lista de vecinos, `neighbors`, será la que recibe como parámetro en `get_neighbors`. El algoritmo irá haciendo llamadas a este método en cada iteración para obtener la lista de vecinos del nodo actual. La estructura de la lista de vecinos es la siguiente:

```
struct {
    int capacity;
    int count;
    int *nodeIds;
    float *costs;
} neighbors_list;
```

Para iniciar la búsqueda búsqueda con el algoritmo secuencial, se debe hacer una llamada al siguiente método, que recibe como parámetro la estructura `AStarSource` y los identificadores del nodo inicial y el final:

```
path *astar_search(AStarSource *source, int s_id, int t_id);
```

que devuelve un puntero a una estructura `path` en caso de encontrar el camino, o `NULL` si no existe el camino. La estructura que representa los caminos tiene la siguiente forma:

```
struct {
    int count;
    int *nodeIds;
    float cost;
} path;
```

donde `count` representa el número de nodos en el camino, `nodeIds` es la lista de identificadores de dichos nodos, en orden, y `cost` es el coste total del camino.

En cuanto a las estructuras de datos propias del algoritmo, la lista abierta se ha implementado con un montículo binario en los ficheros `heap.h` y `heap.c`. Las tres versiones utilizan la misma implementación. La lista cerrada consiste en un array de punteros a estructuras `node_t` y se inicializada con el tamaño especificado por `max_size`. Ambas estructuras, junto con la lista de vecinos, se crean al comienzo del algoritmo.

## 5.2. Implementación centralizada

La versión paralela centralizada sigue la misma estructura que la versión secuencial. En este caso se utilizan estructuras `omp_lock_t` para proteger la lista abierta y cada uno de los nodos de la lista de visitados.

```
omp_lock_t open_lock;
omp_lock_t *visited_locks;
```

Cada vez que un hilo va a extraer un nodo de la lista abierta debe adquirir el cerrojo, y en el caso de encontrar un mejor camino para uno de los nodos ya en la lista de visitados (abierto o no) también debe bloquear ese nodo. OpenMP ofrece estos mecanismos mediante dos métodos que reciben como parámetro la dirección del cerrojo a adquirir o liberar:

```
omp_set_lock(omp_lock_t *lock);
omp_unset_lock(omp_lock_t *lock);
```

La sección concurrente se declara justo antes del bucle `while` principal, con la directiva OpenMP:

```
# pragma omp parallel if(k > 1) num_threads(k) shared(open,
    open_lock, visited, visited_locks, m, m_lock, terminated,
    terminated_lock)
```

Las variables compartidas entre hilos corresponden, en el orden en que aparecen en la directiva, a la lista abierta y su cerrojo, la lista de visitados y sus cerrojos, la mejor solución actual y su cerrojo, el contador de terminación y su cerrojo. Esta sección solamente se ejecutará de manera concurrente si el número de hilos es mayor a uno, que se especifica como parámetro al iniciar la búsqueda:

```
path *spastar_search(AStarSource *source, int start_id, int
    goal_id, int k)
```

Las principales diferencias de esta versión con la secuencial son dos. En primer lugar, la adquisición y liberación de cerrojos al extraer nodos de la lista abierta y en la evaluación de los vecinos de cada nodo, como ya hemos comentado. La segunda diferencia es la condición de terminación que podemos implementar mediante el uso de un contador atómico (Sección 2.1). Este contador es la variable `terminated`, y los accesos atómicos se controlan mediante el cerrojo `terminated_lock`, aunque se podrían haber encapsulado también las modificaciones dentro de una directiva `critical` de OpenMP. El código para la detección de la terminación (5.2) se ejecuta al comenzar cada iteración del bucle principal. En primer lugar se comprueba si todos los hilos han terminado (líneas 2-7). Si no es así, se obtiene el valor del mejor camino encontrado actualmente (líneas 9-11) con exclusión mutua mediante el cerrojo `m_lock`. Se debe comprobar si la lista abierta está vacía o si el próximo nodo a extraer es peor que la mejor solución

encontrada hasta ahora (línea 14). Esta operación también se hace con exclusión mutua sobre la lista abierta, pues puede haber otros hilos insertando nodos en ese momento. Si se cumple alguna de las condiciones, el hilo aumenta el valor de la variable `terminated` también con exclusión mutua. Se utiliza una variable auxiliar privada para cada hilo, `waiting`, que evita que el mismo hilo aumente el contador varias veces (líneas 16-21). Finalmente, si no se cumple ninguna de las condiciones el hilo puede continuar, pero si había aumentado el contador anteriormente, ahora debe decrementarlo (líneas 25-30).

Detección de terminación en A* centralizado

```
1  ...
2  omp_set_lock(&terminated_lock);
3  if (terminated == k) {
4      omp_unset_lock(&terminated_lock);
5      break;
6  }
7  omp_unset_lock(&terminated_lock);
8
9  omp_set_lock(&m_lock);
10 float m_cost = (m != NULL) ? m->fCost : FLT_MAX;
11 omp_unset_lock(&m_lock);
12
13 omp_set_lock(&open_lock);
14 if (heap_is_empty(open) || heap_min(open) >= m_cost) {
15     omp_unset_lock(&open_lock);
16     if (!waiting) {
17         waiting = 1;
18         omp_set_lock(&terminated_lock);
19         terminated++;
20         omp_unset_lock(&terminated_lock);
21     }
22     continue;
23 }
24
25 if (waiting) {
26     waiting = 0;
27     omp_set_lock(&terminated_lock);
28     terminated--;
29     omp_unset_lock(&terminated_lock);
30 }
31 ...
```

## 5.3. Implementación descentralizada

La versión descentralizada es la que más se diferencia de las otras dos en cuanto a la implementación se refiere. El bucle principal no es tan paracedido al de la versión secuencial como si lo era el de la versión centralizada. En esta estrategia, cada hilo cuenta con una seria fe *buffers* que utiliza para la comunicación con el resto de hilos. Estos *buffers* se implementan con las siguientes estructuras en los ficheros `buffer.h` y `buffer.c`:

```
struct {
    int id;
    float gCost;
    int parent;
} buffer_elem_t;

struct {
    int capacity;
    int size;
    buffer_elem_t *elems;
} buffer_t;
```

Principalmente, cada hilo contará con un *buffer* de recepción que el resto de hilos deberán boquear mediante un cerrojo para escribir en él los nodos que le pertenezcan, llamado `incomebuffer`. Como hemos comentado en la Sección 2.2, estos bloqueos sobre los `incomebuffer` provocan una sobrecarga en la comunicación y disminuyen el rendimiento general del algoritmo. Para aliviar estos bloqueos, se ha seguido la siguiente estrategia: cada hilo cuenta con una serie de *buffers* llamados `outgobuffer`, uno por cada hilo, en los que irá incluyendo los nodos que le pertenezcan a otros hilos si en el momento de mandárselos no puede obtener el cerrojo correspondiente, ya sea porque otro hilo está añadiendo nodos, o porque el propietario lo está vaciando. Esta estrategia consigue que los hilos esperen a mandar los nodos cuando tienen el `incombuffer` correspondiente disponible, pero puede provocar que un hilo haga trabajo innecesario, pues quien tiene que mandarle los nodos más prometedores no es capaz de obtener el cerrojo durante un largo periodo de tiempo. Para evitar esto, hemos establecido un número máximo de elementos, `outgo_threshold`, que puede haber en cada `outgobuffer`. Si el número de elementos alcanza este valor, el hilo no seguirá hasta no enviar los nodos a su propietario. Para conseguir este comportamiento, OpenMP nos proporciona el siguiente método:

```
int omp_test_lock(omp_lock_t *lock);
```

que devuelve 1 (*true*) si se logrado obtener el cerrojo y 0 (*false*) en caso contrario. Este mecanismo también se aplica cuando un hilo comprueba si ha recibido nodos. Si alguno otro sigue escribiendo nodos en su `incomebuffer`, éste no se bloquea

y sigue su ejecución. A continuación podemos ver las primeras líneas del bucle principal (5.3), en el que se hace esta comprobación (líneas 2-14):

Comprobación del *buffer* de recepción

```
1  ...
2  if (incomebuffers[tid]->size > 0) {
3      omp_set_lock(&terminate_lock);
4      terminate[tid] = 0;
5      omp_unset_lock(&terminate_lock);
6      if (incomebuffers[tid]->size >= income_threshold) {
7          omp_set_lock(&incomebuffers_locks[tid]);
8          fill_buffer(tmp_buffer, incomebuffers[tid]);
9          omp_unset_lock(&incomebuffers_locks[tid]);
10     } else if (omp_test_lock(&incomebuffers_locks[tid])) {
11         fill_buffer(tmp_buffer, incomebuffers[tid]);
12         omp_unset_lock(&incomebuffers_locks[tid]);
13     }
14 }
15
16 if (tmp_buffer->size > 0) {
17     for (int i = 0; i < tmp_buffer->size; i++) {
18         buffer_elem_t msg = tmp_buffer->elems[i];
19         if (visited[msg.node_id] != NULL) {
20             if (msg.gCost < visited[msg.node_id]->gCost) {
21                 visited[msg.node_id]->gCost = msg.gCost;
22                 visited[msg.node_id]->fCost = msg.gCost +
                       source->heuristic(msg.node_id, goal_id);
23                 visited[msg.node_id]->parent = msg.parent_id;
24                 if (visited[msg.node_id]->is_open) heap_update
                       (open, visited[msg.node_id]);
25                 else heap_insert(open, visited[msg.node_id]);
26             }
27         } else {
28             visited[msg.node_id] = node_create(msg.node_id,
                   msg.gCost, msg.gCost + source->heuristic(msg.
                   node_id, goal_id), msg.parent_id);
29             heap_insert(open, visited[msg.node_id]);
30         }
31     }
32     tmp_buffer->size = 0;
33 }
34 ...
```

El valor de `income_threshold` es el número máximo de elementos para forzar la recepción de nodos, al igual que en el envío. En este caso, para evitar bloquear el

*buffer* de recepción si otros hilos tienen que insertar nodos, se copia el contenido a un *buffer* temporal, y posteriormente se tratan como si de la expansión en la versión secuencial se tratara (líneas 16-33). En este caso, al contrario que en la versión centralizada, las listas abierta y cerrada son privadas, por lo que esta segunda parte se hace sin necesidad de obtener ningún cerrojo.

A continuación, se realiza la comprobación de la condición de terminación de la misma forma que en la versión centralizada (5.2). Finalmente, se expande el nodo obtenido de la lista abierta local (5.3). Para cada vecino se deberá calcular qué hilo es el propietario. Esta operación se implementa en HDA* mediante un función *hash*:

```
static inline int hash(int n_id, int k) {
    return n_id % k;
}
```

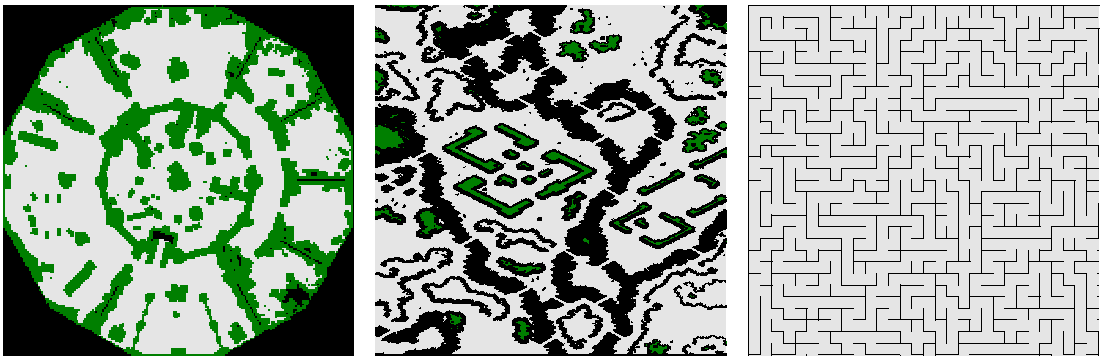
Si el nodo pertenece al propio hilo, se puede comprobar directamente si es duplicado o hay que insertarlo en las listas locales (líneas 6-18). En caso de pertenecer a otro hilo se sigue el esquema que se ha presentado antes: si el número de elementos a enviar es igual o mayor que el valor de umbral, se envían (líneas 19-23); si no lo supera pero ha podido obtener el cerrojo, también los envía (líneas 24-29); y si no es ninguno de los casos, los guarda en el `outgobuffer` correspondiente para enviarlos después (líneas 30-32).

Evaluación de los nodos vecinos en HDA*

```
1  ...
2  for (int i = 0; i < neighbors->count; i++) {
3      int n_id = neighbors->nodeIds[i];
4      float new_cost = current->gCost + neighbors->costs[i];
5      int owner = hash(n_id, k);
6      if (owner == tid) {
7          if (visited[n_id]) {
8              if (new_cost < visited[n_id]->gCost) {
9                  visited[n_id]->gCost = new_cost;
10                 visited[n_id]->fCost = new_cost + source->
                       heuristic(n_id, goal_id);
11                 visited[n_id]->parent = current->id;
12                 if (visited[n_id]->is_open) heap_update(open,
                       visited[n_id]);
13                 else heap_insert(open, visited[n_id]);
14             }
15         } else {
16             visited[n_id] = node_create(n_id, new_cost,
                   new_cost + source->heuristic(n_id, goal_id),
                   current->id);
17             heap_insert(open, visited[n_id]);
18         }
19     } else if (outgobuffers[owner]->size > outgo_threshold) {
20         omp_set_lock(&incomebuffers_locks[owner]);
21         buffer_insert(incomebuffers[owner], (buffer_elem_t){
               n_id, new_cost, current->id});
22         fill_buffer(incomebuffers[owner], outgobuffers[owner])
               ;
23         omp_unset_lock(&incomebuffers_locks[owner]);
24     } else if (omp_test_lock(&incomebuffers_locks[owner])) {
25         buffer_insert(incomebuffers[owner], (buffer_elem_t){
               n_id, new_cost, current->id});
26         if (outgobuffers[owner]->size > 0) {
27             fill_buffer(incomebuffers[owner], outgobuffers[
                   owner]);
28         }
29         omp_unset_lock(&incomebuffers_locks[owner]);
30     } else {
31         buffer_insert(outgobuffers[owner], (buffer_elem_t){
               n_id, new_cost, current->id});
32     }
33 }
```

# 6. Resultados

A continuación vamos a evaluar los tres algoritmos en un conjunto de escenarios y compararemos los tiempos obtenidos. La colección de ejemplos se ha obtenido de la página web del profesor Nathan R. Sturtevant[2][52]. Los escenarios de prueba consisten en *grids* de mapas de videojuegos conocidos, como *Dragon Age*, *Baldur's Gate II* o *Starcraft*, además de otros ejemplos sintéticos como laberintos o mapas aleatorios (Figura 8).



**(a)** *Dragon Age* **(b)** *StarCraft* **(c)** Laberinto

**Figura 8:** Ejemplos de los mapas de prueba.

Es precisamente en este tipo de escenarios representados por *grids* en los que las versiones paralelas de los algoritmos de búsquedas suelen tener más problemas en obtener una mejora con respecto a la versión secuencial. En el artículo de Burns et al. [4] en el que comparan distintas versiones paralelas de la búsqueda primero el mejor, entre las que se incluyen PRA* y HDA*, obtienen una probabilidad de que un nodo se bloquee cuando se permiten movimientos en las cuatro direcciones cardinales de un 30 %, y de un 45 % cuando los movimientos se permiten en las ocho direcciones cuando se trata de versiones centralizadas. Eso sumado a que en este tipo de problemas la operación de expansión es poco costosa, hace que los hilos pasen la mayor parte del tiempo en espera para obtener los cerrojos. Además, cuanto más cercano es el valor de la heurística al valor real, más tenderán los hilos a buscar todos por la misma zona.

Todos los tiempos que se mostrarán a continuación se han obtenido ejecutando los algoritmos sobre el mapa correspondiente en diez casos con un coste de camino similar y tomando la media de esas ejecuciones.

Vamos a empezar con un conjunto de mapas de tipo laberinto. En la Figura 9 podemos ver los resultados para las ejecuciones según crece el número de hilos.

[2] https://movingai.com/

Los dos primeros números del título en cada gráfica representan la altura y la anchura total del mapa, y el tercer número en este caso representa el ancho de los caminos del laberinto.

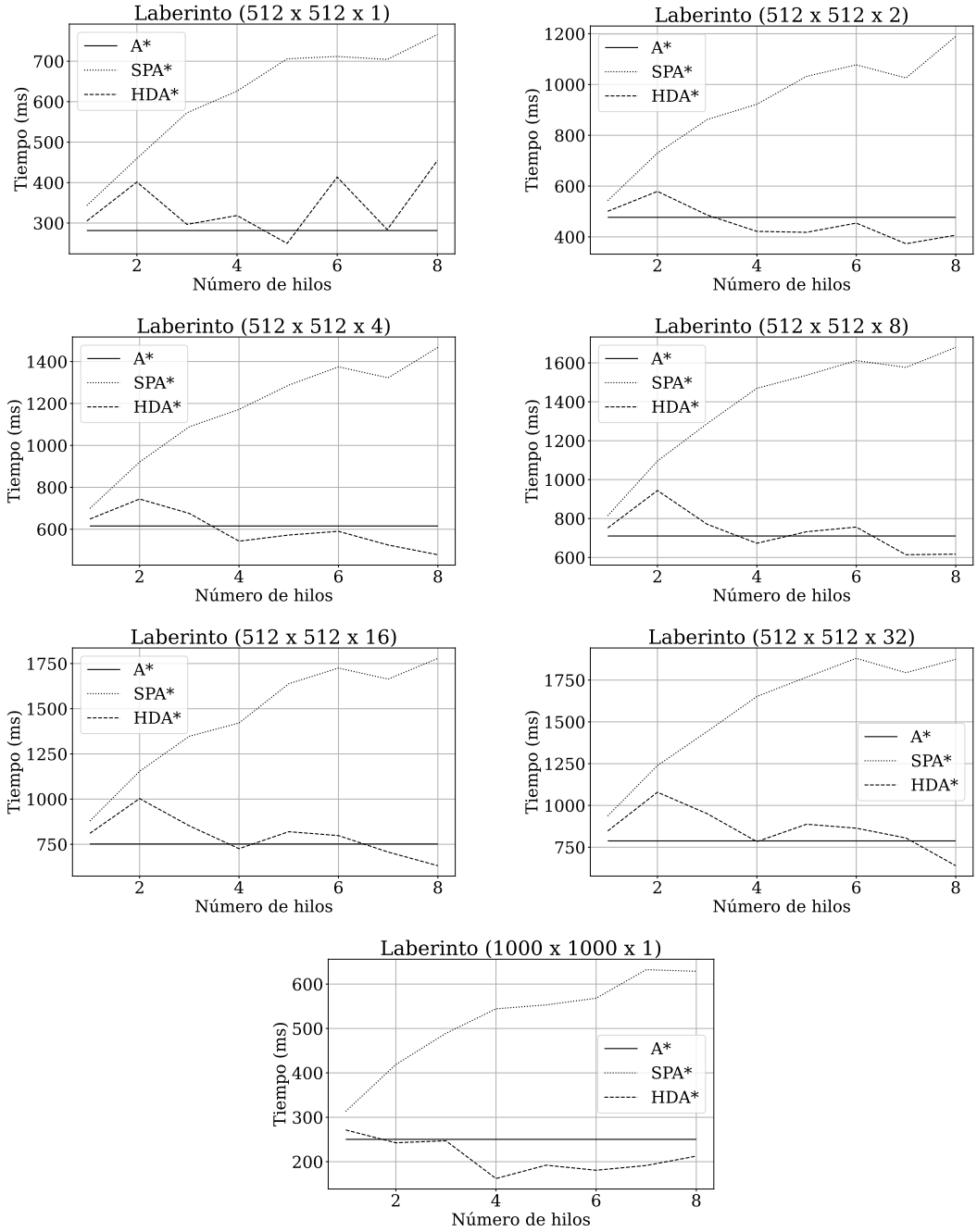



**Figura 9:** Resultados de tiempos de ejecución frente al número de hilos en mapas de tipo laberinto. El primer y segundo número en el título representan las dimensiones del mapa y el tercer número la anchura de los pasillos.

De estos resultados podemos extraer varias conclusiones que veremos replicadas en el resto de ejemplos más adelante. En primer lugar, el tiempo de SPA* no deja de aumentar con el número de hilos en todos los casos. Esto se debe a la sobrecarga en la sincronización entre procesos, y no deja de ser otra prueba de que una paralelización directa del algoritmo empeora enormemente la eficiencia con respecto a la versión secuencial, lo que en la literatura se conoce como un acercamiento ingenuo (*naive approach*). En cuanto a HDA* vemos que si que consigue alguna mejora con respecto a la versión secuencial en algunos casos. Estos casos dependen del número de hilos utilizados y de la estructura del mapa. En general se observa la tendencia a reducir el tiempo de ejecución cuando se aumentan el número de hilos, lo que le hace un buen candidato a una formulación paralela escalable del algoritmo. En algunos casos, como el primer laberinto, vemos que solamente consigue una mejora cuando se usan cinco hilos. Lo que ocurre es que HDA*, a costa de aliviar la sobrecarga de sincronización, introduce una sobrecarga de búsqueda y de comunicación. Esto puede ser determinante en el tiempo de ejecución, sobre todo cuando los mapas son pequeños. Como prueba de ello, podemos ver que cuando se utiliza un mapa más grande, como en el último caso, la mejora es mucho más notable, llegando a reducirse el tiempo de ejecución en un 35 % cuando se utilizan cuatro hilos.

En la Figura 10 podemos apreciar cómo afecta el tamaño del mapa a la mejora obtenida con HDA*. Estos mapas son los mostrados en las Figuras 8a y 8b correspondientes a los videojuegos *Dragon Age* y *Starcraft*. Al lado del nombre entre paréntesis se muestran las dimensiones de cada mapa.

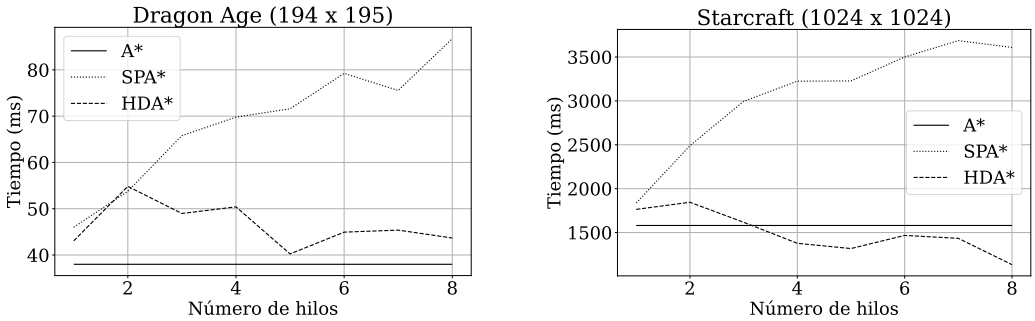



**Figura 10:** Tiempos de ejecución frente al número de hilos involucrados en la búsqueda.

Otro conjunto de mapas en el que se han probado estos algoritmos son mapas aleatorios (Figura 11). Consisten en cuadrículas de $512 \times 512$ en las que se colocan obstáculos de manera aleatoria. Contamos con siete escenarios correspondientes a diferentes porcentajes de ocupación por parte de obstáculos, desde 10 % hasta un 40 %.

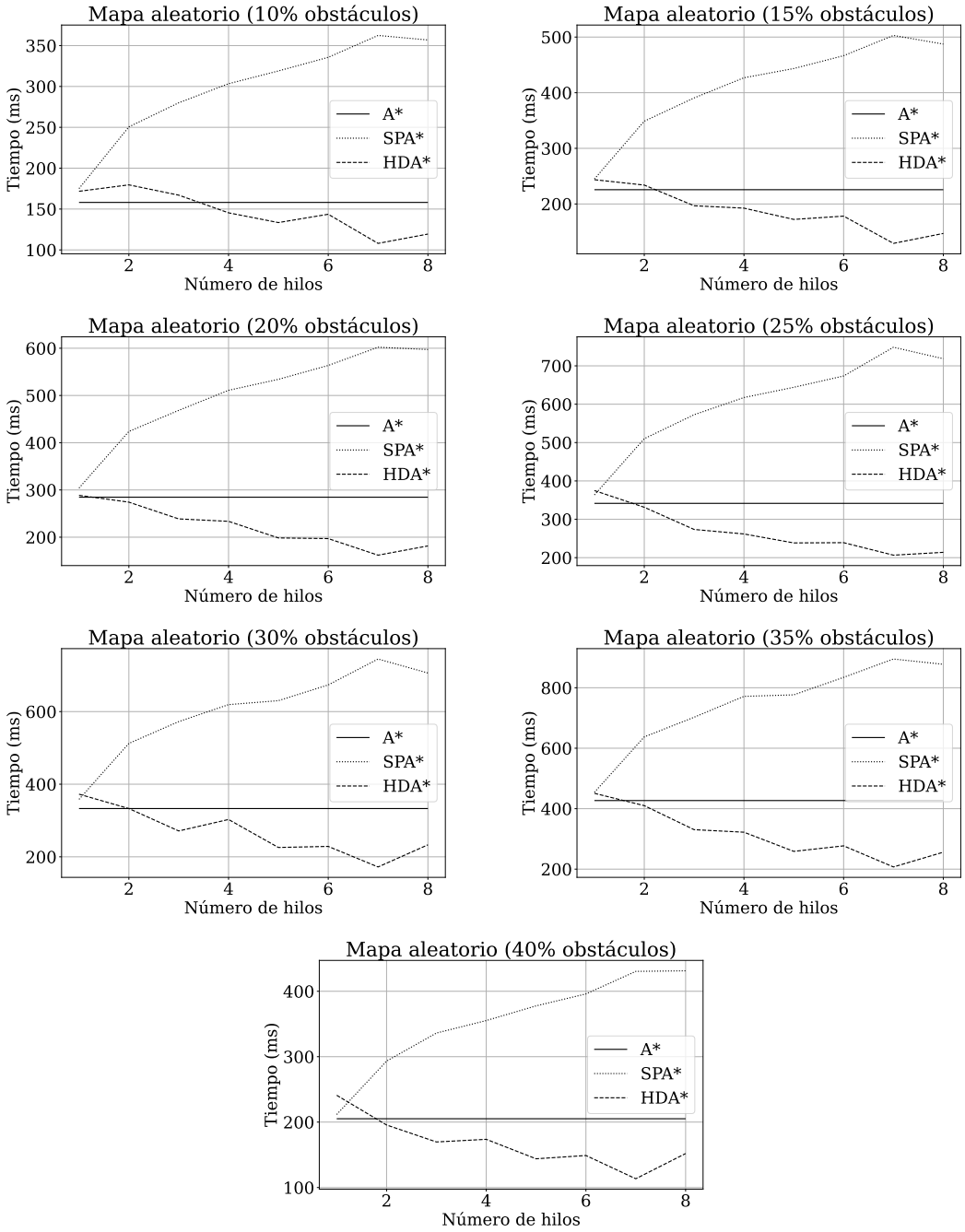



**Figura 11:** Resultados de tiempos de ejecución frente al número de hilos en mapas de tipo aleatorio de dimensiones $512 \times 512$.

En este tipo de mapas se observa una mejora mucho mas estable que en los mapas de laberintos cuando el número de hilos aumenta. En este caso el espacio de búsqueda se reparte mejor entre los distintos hilos, llegando incluso a reducir a la mitad el tiempo de búsqueda en algunos casos.

Finalmente, vamos a evaluar el rendimiento de los tres algoritmos en unos mapas sintéticos para mostrar cómo afecta la naturaleza y la estructura del espacio de búsqueda. Se han construido dos escenarios, uno de tamaño $5000 \times 5000$ sin ningún obstáculo, en el que el nodo de inicio se encuentra arriba a la izquierda y el objetivo abajo a la derecha; y otro de tamaño $954 \times 7$ en el que solamente hay dos caminos hacia el objetivo, ambos con el mismo coste. Podemos ver una representación de éste último en la Figura 12, donde el círuclo verde representa el inicio y el rojo el objetivo.

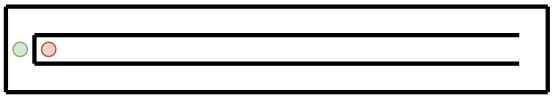


**Figura 12:** Escenario de prueba con dos caminos óptimos para alcanzar la solución.

En la Figura 13 podemos ver como ambos algoritmos funcionan peor en estos casos que la versión secuencial, además se puede observar que HDA* funciona peor que SPA* para estos escenarios.

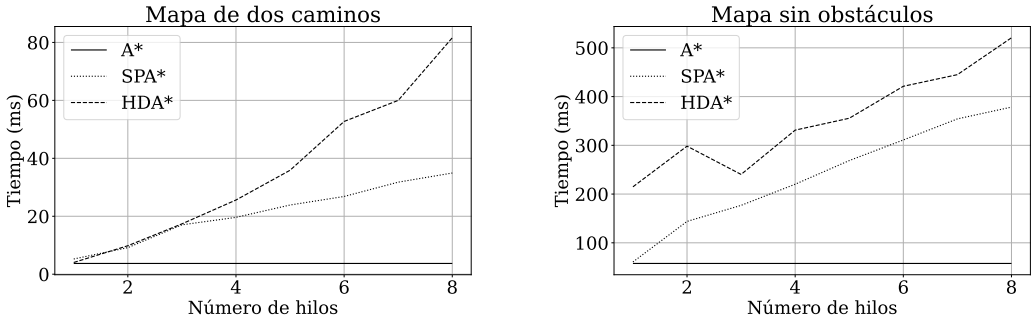



**Figura 13:** Resultados de tiempos de ejecución frente al número de hilos en mapas sintéticos. El primero corresponde al escenario de la Figura 12 y el segundo a un escenario sin ningún obstáculo.

El mal desempeño de HDA* en el mapa de dos caminos se explica debido a que los hilos pasan la mayor parte del tiempo esperando a recibir un nodo para expandir. Por cómo se ha implementado la función *hahs* para repartir los nodos entre hilos, cada nodo consecutivo del escenario pertenecerá a un proceso distinto, por lo que solo el hilo con el nodo actual en el camino óptimo estará realmente trabajando. Cabe destacar que podría obtenerse un mejor resultado diseñando una función *hash* específica para este escenario que repartiera el trabajo entre los hilos de otra manera, como por ejemplo asignando cada fila a un hilo. En el caso del mapa vacío, lo que ocurre es que la heurística guía al camino perfectamente, por lo que la versión secuencial solamente explora los nodos en el camino óptimo,

y cualquier otro trabajo extra realizado por HDA* afecta negativamente a su resultado, además del coste añadido por comunicar los nodos entre procesos.

Por último, vamos ver cómo afecta al rendimiento de HDA* el parámetro `threshold`, que recordemos que es el número máximo de nodos que puede acumular un hilo antes de enviárselos a su propietario. En realidad definíamos dos valores: uno para el *buffer* de recepción y otro para el de envío. Asignamos el mismo valor a ambos, por eso en la Figura 14 se nombra el eje de abscisas como `threshold`. Se ha ejecutado el algoritmo con cuatro y ocho hilos en el mapa con forma de laberinto de $1000 \times 1000 \times 1$.

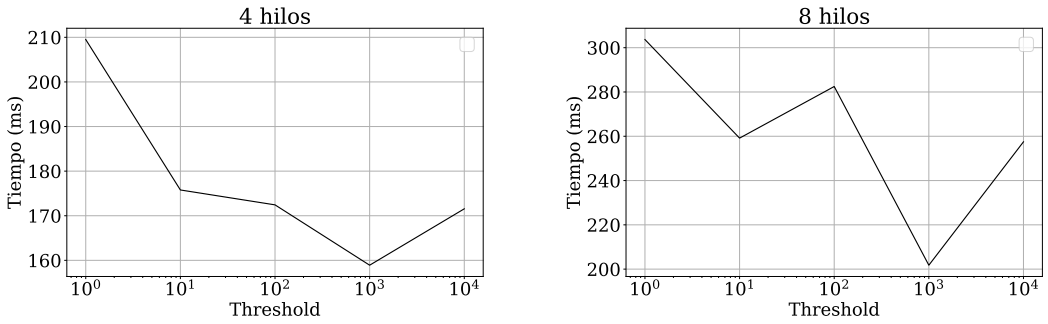



**Figura 14:** Efecto del parámetro `threshold` en el algoritmo HDA* con cuatro y ocho hilos. Eje de abscisas en escala logarítmica.

Se observa una tendencia a disminuir el tiempo de ejecución a medida que aumenta el valor de este parámetro, pero tambien se ve que aumentarlo demasiado puede ser contraproducente, debido a que algún hilo deba esperar demasiado a recibir los nodos que le llevarán a la solución óptima.

# 7. Conclusiones y Trabajo Futuro

## 7.1. Conclusiones

## 7.2. Trabajo Futuro

# Referencias


[1] Barnouti, N.H., Al-Dabbagh, S.S.M., Naser, M.A.S.: Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm. Journal of Computer and Communications **4**(11), 15–25 (Sep 2016). doi:10.4236/jcc.2016.411002, https://www.scirp.org/journal/paperinformation?paperid=70460, number: 11 Publisher: Scientific Research Publishing

[2] Borůvka, O.: O jistém problému minimálním. Práce Mor. Přírodověd. Spol. V Brně III **3**, 37–58 (1926)

[3] Bulitko, V., Lee, G.: Learning in real-time search: A unifying framework. Journal of Artificial Intelligence Research **25**, 119–157 (2006)

[4] Burns, E., Lemons, S., Ruml, W., Zhou, R.: Best-first heuristic search for multicore machines. Journal of Artificial Intelligence Research **39**, 689–743 (2010)

[5] Colorni, A., Dorigo, M., Maniezzo, V., et al.: Distributed optimization by ant colonies. In: Proceedings of the first European conference on artificial life. vol. 142, pp. 134–142. Paris, France (1991)

[6] Daniel, K., Nash, A., Koenig, S., Felner, A.: Theta*: Any-angle path planning on grids. Journal of Artificial Intelligence Research **39**, 533–579 (2010)

[7] Dechter, R., Pearl, J.: The optimality of A* revisited. In: Proceedings of the Third AAAI Conference on Artificial Intelligence. pp. 95–99 (1983)

[8] Dechter, R., Pearl, J.: Generalized best-first search strategies and the optimality of A*. Journal of the ACM (JACM) **32**(3), 505–536 (1985)

[9] Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**(1), 269–271 (Dec 1959). doi:10.1007/BF01386390, https://doi.org/10.1007/BF01386390

[10] Doran, J.E., Michie, D., Kendall, D.G.: Experiments with the Graph Traverser program. Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences **294**(1437), 235–259 (Jan 1997). doi:10.1098/rspa.1966.0205, https://royalsocietypublishing.org/doi/10.1098/rspa.1966.0205, publisher: Royal Society

[11] Eckstein, D.M., Alton, D.A.: Parallel graph processing using depth-first search. University of Iowa. Department of Computer Science (1977)

[12] Euler, L.: Solutio problematis ad geometriam situs pertinentis. Commentarii academiae scientiarum Petropolitanae pp. 128–140 (1741)

[13] Evett, M., Hendler, J., Mahanti, A., Nau, D.: PRA*: Massively parallel heuristic search. Journal of Parallel and Distributed Computing **25**(2), 133–143 (1995)

[14] Felner, A., Li, J., Boyarski, E., Ma, H., Cohen, L., Kumar, T.K.S., Koenig, S.: Adding Heuristics to Conflict-Based Search for Multi-Agent Path Finding. Proceedings of the International Conference on Automated Planning and Scheduling **28**, 83–87 (Jun 2018). doi:10.1609/icaps.v28i1.13883, https://ojs.aaai.org/index.php/ICAPS/article/view/13883

[15] Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM (JACM) **34**(3), 596–615 (1987)

[16] Freeman, J.: Parallel algorithms for depth-first search. Department of Computer and Information Science. University of Pennsylvania (1991)

[17] Fukunaga, A., Botea, A., Jinnai, Y., Kishimoto, A.: A survey of parallel A*. arXiv preprint arXiv:1708.05296 (2017)

[18] Gelperin, D.: On the optimality of A*. Artificial Intelligence **8**(1), 69–76 (1977)

[19] Grama, A., Kumar, V.: Parallel search algorithms for discrete optimization problems. ORSA Journal on Computing **7**(4), 365–385 (1995)

[20] Hansen, E.A., Zhou, R.: Anytime heuristic search. Journal of Artificial Intelligence Research **28**, 267–297 (2007)

[21] Harabor, D., Grastien, A.: Online Graph Pruning for Pathfinding On Grid Maps. Proceedings of the AAAI Conference on Artificial Intelligence **25**(1), 1114–1119 (Aug 2011). doi:10.1609/aaai.v25i1.7994, https://ojs.aaai.org/index.php/AAAI/article/view/7994

[22] Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Transactions on Systems Science and Cybernetics **4**(2), 100–107 (Jul 1968). doi:10.1109/TSSC.1968.300136, https://ieeexplore.ieee.org/document/4082128, conference Name: IEEE Transactions on Systems Science and Cybernetics

[23] Hart, P.E., Nilsson, N.J., Raphael, B.: Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". ACM SIGART Bulletin **37**, 28–29 (Dec 1972). doi:10.1145/1056777.1056779, https://dl.acm.org/doi/10.1145/1056777.1056779

[24] HUANG, S.R.: A tight upper bound for the speed-up of parallel best-first branch-and-bound algorithms. Center for Automation Research, University of Maryland (1987)

[25] Jinnai, Y., Fukunaga, A.: Abstract zobrist hashing: An efficient work distribution method for parallel best-first search. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 30 (2016)

[26] Karp, R., Zhang, Y.: A randomized parallel branch-and-bound procedure. In: Proceedings of the twentieth annual ACM symposium on Theory of computing. pp. 290–300 (1988)

[27] Karp, R.M., Zhang, Y.: Randomized parallel algorithms for backtrack search and branch-and-bound computation. Journal of the ACM (JACM) **40**(3), 765–789 (1993)

[28] Kishimoto, A., Fukunaga, A., Botea, A.: Scalable, parallel best-first search for optimal sequential planning. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 19, pp. 201–208 (2009)

[29] Kishimoto, A., Fukunaga, A., Botea, A.: Evaluation of a simple, scalable, parallel best-first search strategy. Artificial Intelligence **195**, 222–248 (2013)

[30] Koenig, S., Likhachev, M.: Incremental A*. Advances in neural information processing systems **14** (2001)

[31] Korf, R.E.: Depth-first iterative-deepening: An optimal admissible tree search. Artificial intelligence **27**(1), 97–109 (1985)

[32] Kumar, V., Ramesh, K., Rao, V.N.: Parallel best-first search of state-space graphs: A summary of results. In: AAAI. vol. 88, pp. 122–127. Citeseer (1988)

[33] Kumar, V., Rao, V.N.: Parallel depth first search. part ii. analysis. International Journal of Parallel Programming **16**(6), 501–519 (1987)

[34] Lai, T.H., Sahni, S.: Anomalies in parallel branch-and-bound algorithms. Communications of the ACM **27**(6), 594–602 (1984)

[35] Leifker, D.B., Kanal, L.N.: A Hybrid SSS*/Alpha-Beta Algorithm for Parallel Search of Game Trees. University of Maryland. Computer Science (1984)

[36] Mahapatra, N.R., Dutt, S.: Scalable global and local hashing strategies for duplicate pruning in parallel a* graph search. IEEE Transactions on Parallel and Distributed Systems **8**(7), 738–756 (2002)

[37] Martins, O.O., Adekunle, A.A., Olaniyan, O.M., Bolaji, B.O.: An Improved multi-objective a-star algorithm for path planning in a large workspace: Design, Implementation, and Evaluation. Scientific African **15**, e01068 (Mar 2022). doi:10.1016/j.sciaf.2021.e01068, https://www.sciencedirect.com/science/article/pii/S2468227621003690

[38] Nillsson, N.J.: Problem Solving Methods in Artificial Intelligence. McGraw-Hill Book Company, New York, New York (1971)

[39] Nilsson, N.J.: Principles of artificial intelligence. Morgan Kaufmann (2014)

[40] Phillips, M., Likhachev, M., Koenig, S.: Pa* se: Parallel a* for slow expansions. In: Proceedings of the International Conference on Automated Planning and Scheduling. vol. 24, pp. 208–216 (2014)

[41] Rafiq, A., Asmawaty Abdul Kadir, T., Normaziah Ihsan, S.: Pathfinding Algorithms in Game Development. IOP Conference Series: Materials Science and Engineering **769**(1), 012021 (Feb 2020). doi:10.1088/1757-899X/769/1/012021, https://iopscience.iop.org/article/10.1088/1757-899X/769/1/012021

[42] Rao, V.N., Kumar, V.: Parallel depth first search. part i. implementation. International Journal of Parallel Programming **16**(6), 479–499 (1987)

[43] Rao, V.N., Kumar, V.: Superlinear speedup in parallel state-space search. In: Foundations of Software Technology and Theoretical Computer Science: Eighth Conference, Pune, India December 21–23, 1988 Proceedings 8. pp. 161–174. Springer (1988)

[44] Reghbati, E., Corneil, D.G.: Parallel computations in graph theory. SIAM Journal on Computing **7**(2), 230–237 (1978)

[45] Reif, J.H.: Depth-first search is inherently sequential. Information Processing Letters **20**(5), 229–234 (1985)

[46] Rios, L.H.O., Chaimowicz, L.: Pnba*: A parallel bidirectional heuristic search algorithm. In: ENIA VIII Encontro Nacional de Inteligê ncia Artificial (2011)

[47] Russell, S.: E cient memory-bounded search methods. In: Proc of ECAI-92. Citeseer (1992)

[48] Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. pearson (2016)

[49] Schrijver, A.: On the history of the shortest path problem. In: Grötschel, M. (ed.) Documenta Mathematica Series, vol. 6, pp. 155–167. EMS Press, 1

edn. (jan 2012). doi:10.4171/dms/6/19, https://ems.press/doi/10.4171/dms/6/19

[50] Standley, T.S., Korf, R.: Complete Algorithms for Cooperative Pathfinding Problems. International Joint Conference on Artificial Intelligence (2011), https://www.semanticscholar.org/paper/Complete-Algorithms-for-Cooperative-Pathfinding-Standley-Korf/146061be1affd4af17b8996f1d0316ad147368f5

[51] Stentz, A.: Optimal and efficient path planning for partially-known environments. In: Proceedings of the 1994 IEEE international conference on robotics and automation. pp. 3310–3317. IEEE (1994)

[52] Sturtevant, N.: Benchmarks for grid-based pathfinding. Transactions on Computational Intelligence and AI in Games **4**(2), 144 – 148 (2012), http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf

[53] Tero, A., Takagi, S., Saigusa, T., Ito, K., Bebber, D.P., Fricker, M.D., Yumiki, K., Kobayashi, R., Nakagaki, T.: Rules for biologically inspired adaptive network design. Science **327**(5964), 439–442 (2010)

[54] Vidal, V., Bordeaux, L., Hamadi, Y.: Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In: Proceedings of the International Symposium on Combinatorial Search. vol. 1, pp. 100–107 (2010)

[55] Wiener, C.: Ueber eine aufgabe aus der geometria situs. Mathematische Annalen **6**(1), 29–30 (1873)

[56] Wyllie, J.C.: The complexity of parallel computations. Tech. rep., Cornell University (1979)

[57] Yap, P.: Grid-Based Path-Finding. In: Goos, G., Hartmanis, J., Van Leeuwen, J., Cohen, R., Spencer, B. (eds.) Advances in Artificial Intelligence, vol. 2338, pp. 44–55. Springer Berlin Heidelberg, Berlin, Heidelberg (2002). doi:10.1007/3-540-47922-8_4, http://link.springer.com/10.1007/3-540-47922-8_4, series Title: Lecture Notes in Computer Science