

UNIVERSIDAD DE OVIEDO  
ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN

TRABAJO DE FIN DE GRADO

---

**Optimización de Algoritmos de Búsqueda  
en Grafos:  
Implementación y Comparación de  
Rendimiento en FPGA**

---

*Autor:*

Alejandro Rodríguez López

*Tutor:*

Juan José Palacios Alonso

*Memoria entregada en cumplimiento con los requisitos indicados por Trabajo de  
Fin de Grado de Grado de Ingeniería Informática en Tecnologías de la Información*

24 de junio de 2024

UNIVERSIDAD DE OVIEDO

## *Resumen*

Escuela Politécnica de Ingeniería de Gijón  
Informática

Grado de Ingeniería Informática en Tecnologías de la Información

### **Optimización de Algoritmos de Búsqueda en Grafos: Implementación y Comparación de Rendimiento en FPGA**

por **Alejandro Rodríguez López**

La gran mayoría de ordenadores hoy en día poseen varios núcleos en sus procesadores, gracias a ellos se nos permite realizar diversas tareas de forma concurrente sin percatarnos de que un único procesador sólo puede hacer una tarea a la vez. La mayoría de programas deberían aprovechar este principio para acelerar y presentar resultados a sus usuarios más rápido.

En algunos casos esto no es así, tal vez porque sus desarrolladores no se han dignado a explorar los beneficios del paralelismo. Otra posibilidad es que el simple uso de varios hilos no implica una mejora en rendimiento, el diseño del algoritmo y sus secciones paralelas son de crucial importancia para obtener beneficios tangibles.

Algunos algoritmos son más propensos al paralelismo, en ellos resulta más fácil hallar zonas en las que el uso de múltiples hilos es sencillo de implementar y no existe mucho sobrecoste. Desafortunadamente esto no sucede en todos los algoritmos. En esta investigación se analiza el algoritmo A\*, un claro ejemplo de cómo el paralelismo requiere una muy detenida atención a la estrategia para que sea rentable.

# Índice general

<b>Resumen</b>	<b>I</b>
<b>1. Hipótesis de Partida y Alcance: <i>Estado del Arte</i></b>	<b>1</b>
1.1. Objeto de la investigación	1
1.2. Estado del arte	2
1.3. Requisitos	3
1.4. Alcance	3
<b>2. Problema a investigar: <i>Análisis, implementación y optimización</i></b>	<b>4</b>
2.1. Job Shop Scheduling Problem	4
2.2. Método a resolver	5
2.2.1. Algoritmo	5
2.2.1.1. Componentes A*	6
2.2.1.2. Pseudocódigo	8
2.2.2. Equipo de Estudio	8
2.2.2.1. Arquitectura x86	8
2.2.2.2. FPGA	9
2.3. Método de comparativas	10
2.4. Implementación	11
2.4.1. Task	11
2.4.2. State	11
2.5. Optimización	13
2.5.1. A*	13
2.5.1.1. State	13
2.5.1.2. Coste H - Heurístico	14
2.5.2. Paralelización	16
2.5.2.1. First Come First Serve (FCFS) Solver	16
2.5.2.2. Batch Solver	18
2.5.2.3. Recursive Solver	19
2.5.2.4. Hash Distributed A* (HDA*) Solver	20
<b>3. Experimentos: <i>Trabajo y Resultados</i></b>	<b>21</b>
3.1. Conjuntos de datos	21
3.2. Equipos de Prueba	22
3.2.1. Arquitectura x86	22
3.2.1.1. OS	22
3.2.1.2. CPU	22
3.2.1.3. RAM	22
3.3. Método de medición	23
3.4. Métricas	25
3.4.1. Speedup	25
3.4.2. Makespan	25
3.5. Resultados y Análisis	26

3.5.1. Complejidad del problema . . . . .	26
3.5.2. Comparativa de heurísticos . . . . .	27
3.5.3. Cuellos de botella en secciones críticas . . . . .	28
3.5.4. Comparativa de algoritmos . . . . .	30
3.5.5. Casos particulares . . . . .	31
3.5.5.1. Varios estados iniciales . . . . .	31
3.5.5.2. Estados solución intermedios . . . . .	31
<b>4. Conclusiones: <i>Observaciones y Trabajos futuros</i></b>	<b>32</b>
<b>A. Código Fuente:</b>	<b>33</b>
<b>B. Resultados y Métricas:</b>	<b>34</b>
<b>Bibliografía</b>	<b>35</b>
<b>Índice alfabético</b>	<b>36</b>

# Índice de figuras

2.1. Comparativa entre algoritmos Dijkstra y A* . . . . .	5
2.2. Representación de la estrategia FCFS . . . . .	16
2.3. Comparativa entre algoritmos monohilo y multihilo . . . . .	16
2.4. Representación de la estrategia Batch . . . . .	18
2.5. Representación de la estrategia HDA* . . . . .	20
3.1. Tiempo de ejecución de un único problema. . . . .	26
3.2. Tiempo de ejecución de un único problema (escala logarítmica). . . . .	26
3.3. Métricas heurístico lento. . . . .	27
3.4. Métricas heurístico rápido. . . . .	27
3.5. <i>Speedup</i> en un problema de gran tamaño. . . . .	28
3.6. Métricas de paralelismo en FCFS. . . . .	29
3.7. <i>Speedup</i> de HDA*. . . . .	29
3.8. Tiempo de ejecución de todos los algoritmos (1 hilo). . . . .	30
3.9. Métricas con 4 hilos. . . . .	30
3.10. Métricas con 8 hilos. . . . .	31

# Índice de cuadros

3.1. Equipos de prueba, arquitectura x86: OS . . . . .	22
3.2. Equipos de prueba, arquitectura x86: CPU . . . . .	22
3.3. Equipos de prueba, arquitectura x86: RAM . . . . .	22

# Lista de Abreviaturas

<b>CPU</b>	<b>Central Processing Unit</b>
<b>CSV</b>	<b>Comma Separated Value</b>
<b>FCFS</b>	<b>First Come First Serve</b>
<b>FPGA</b>	<b>Field Programmable Gate Array</b>
<b>HDA*</b>	<b>Hash Distributed A*</b>
<b>HDL</b>	<b>Hardware Description Language</b>
<b>HLS</b>	<b>High Level Synthesis</b>
<b>HTTP</b>	<b>Hyper Text Transfer Protocol</b>
<b>JSP</b>	<b>Job Shop Problem</b>
<b>JSSP</b>	<b>Job Shop Scheduling Problem</b>
<b>OS</b>	<b>Operative System</b>
<b>SoC</b>	<b>System on Chip</b>

## Capítulo 1

# Hipótesis de Partida y Alcance

*Estado del Arte*

### 1.1. Objeto de la investigación

El presente proyecto tiene como objetivo principal descubrir los beneficios del paralelismo aplicado al algoritmo A\*. Para estudiar el rendimiento de las implementaciones desarrolladas se utilizará el Job Shop Scheduling Problem. Adicionalmente, se observará el rendimiento de una implementación de la misma solución en una FPGA.

La motivación principal para la realización de este proyecto emana en un interés personal por el diseño, implementación y optimización de algoritmos aplicables a problemas reales.

El Job Shop Scheduling Problem es un problema de optimización sobre la planificación de horarios. Este problema en particular es mundialmente conocido, ha sido resuelto utilizando un gran abanico de algoritmos diferentes y ha sido profundamente estudiado.

La resolución del Job Shop Scheduling Problem requiere el diseño e implementación de un algoritmo capaz de recibir como entrada las descripciones de una plantilla de trabajadores y un listado de trabajos y tareas a realizar puede ser aplicable en ámbitos industriales donde la automatización de la creación de planificaciones pueda ser de interés.



## 1.2. Estado del arte

Este proyecto abarca diversos tópicos, cuyas bibliografías (incluso en individual) tienen una gran extensión. A pesar de ello, resulta complicado hallar estudios previos sobre implementaciones paralelas del algoritmo A\* enfocadas a la resolución del Job Shop Scheduling Problem utilizando FPGAs. Así pues, el punto de partida de este proyecto se compone principalmente de estudios sobre los distintos tópicos de forma individual.

El Job Shop Scheduling Problem tiene su origen en la década de 1960, desde entonces ha sido utilizado frecuentemente (incluso hasta el día de hoy) como herramienta de medición del rendimiento de algoritmos que sean capaces de resolverlo [Man67].

A lo largo de los años, se han realizado numerosos trabajos con el objetivo de recoger distintos algoritmos que resuelvan el problema. Dichos algoritmos provienen de diferentes ámbitos de la computación. Entre ellos se pueden encontrar búsquedas en grafos, listas de prioridad, ramificación y poda, algoritmos genéticos, simulaciones Monte Carlo o métodos gráficos como diagramas Gantt y Pert. [Yan77], [Nil69], [KTM99], [BC22].

El algoritmo A\* fue diseñado a finales de la década de 1960 con el objetivo de implementar el enrutamiento de un robot conocido como "Shakey the Robot" [Nil84]. A\* es una evolución del conocido algoritmo de Dijkstra frecuentemente utilizado también para la búsqueda en grafos. La principal diferencia entre estos dos algoritmos es el uso de una función heurística en el A\* que 'guía' al algoritmo en la dirección de la solución. [HNR68], [MSV13], [Kon14].

El algoritmo A\* no es propenso a la paralelización, no existe una implementación simple que aproveche el funcionamiento de varios procesadores de forma simultánea. En su lugar existen varias alternativas que paralelizan el algoritmo, cada una de ellas con sus fortalezas y debilidades. Estas diferentes versiones serán estudiadas, implementadas y probadas en esta investigación. [Zag+17], [WH16].

El tópico sobre el que menor cantidad de documentación existe y que supone una mayor curva de aprendizaje es sin lugar a duda la implementación del algoritmo A\* en una FPGA. A diferencia de este proyecto, la principal fuente de bibliografía sobre este tópico desarrolla una implementación personalizada del algoritmo que reporta una aceleración de casi el 400%. [ZJW20].

### 1.3. Requisitos

El algoritmo a desarrollar en esta investigación deberá recibir como entrada una estructura de datos que contenga los trabajos, para cada trabajo la lista de tareas que lo compone, para cada tarea su duración y el listado de trabajadores cualificados para realizarla (generalmente, la longitud de esta lista será 1).

Como resultado, el algoritmo deberá retornar un listado de trabajos, para cada trabajo el instante de tiempo en el que se inicia cada una de sus tareas y un listado con los instantes de tiempo en los que cada trabajador quedará libre. El máximo elemento del listado de instantes de tiempo en los que cada trabajador queda libre se define como el *makespan*, el tiempo necesario para finalizar todos los trabajos.

### 1.4. Alcance

El presente documento describe el problema a resolver en detalle, los diferentes algoritmos implementados, observaciones sobre los mismos, casos de prueba utilizados para obtener métricas de rendimiento y observaciones sobre las métricas obtenidas.

Entre las observaciones tanto de los algoritmos como de las métricas obtenidas, se encontrarán razonamientos sobre los resultados así como explicaciones de las razones por las cuales un algoritmo presenta un rendimiento distinto a otro.

## Capítulo 2

# Problema a investigar

*Análisis, implementación y optimización*

### 2.1. Job Shop Scheduling Problem

Se estudia la implementación de una solución al problema *Job Shop Scheduling (JSP)* [Yan77]<sup>1</sup> resuelto utilizando el algoritmo A\*. Como su nombre indica, se trata de un problema en el que se debe crear una planificación. Claramente, el JSP es un problema de optimización.

El JSP busca una planificación para una serie de máquinas (o trabajadores) que deben realizar un número conocido de trabajos. Cada trabajo está formado por una serie de operaciones (o tareas), con una duración conocida. Las tareas de un mismo trabajo deben ser ejecutadas en un orden específico.

Existen numerosas variantes de este problema, algunas permiten la ejecución en paralelo de algunas tareas o requieren que alguna tarea en específico sea ejecutada por un trabajador (o tipo de trabajador) en particular. Por ello, es clave denotar las normas que se aplicarán a la hora de resolver el problema JSP:

1. Existen un número natural conocido de trabajos.
2. Todos los trabajos tienen el mismo número natural conocido de tareas.
3. A excepción de la primera tarea de cada trabajo, todas tienen una única tarea predecesora que debe ser completada antes de iniciar su ejecución.
4. Cada tarea puede tener una duración distinta.
5. La duración de cada tarea es un número natural conocido.
6. Existe un número natural conocido de trabajadores.
7. Cada tarea tiene un trabajador asignado, de forma que sólo ese trabajador puede ejecutar la tarea.
8. Una vez iniciada una tarea, no se puede interrumpir su ejecución.
9. Un mismo trabajador puede intercalar la ejecución de tareas de diferentes trabajos.

---

<sup>1</sup>El problema también es conocido por otros nombres similares como *Job Shop Scheduling Problem (JSSP)* o *Job Scheduling Problem (JSP)*.

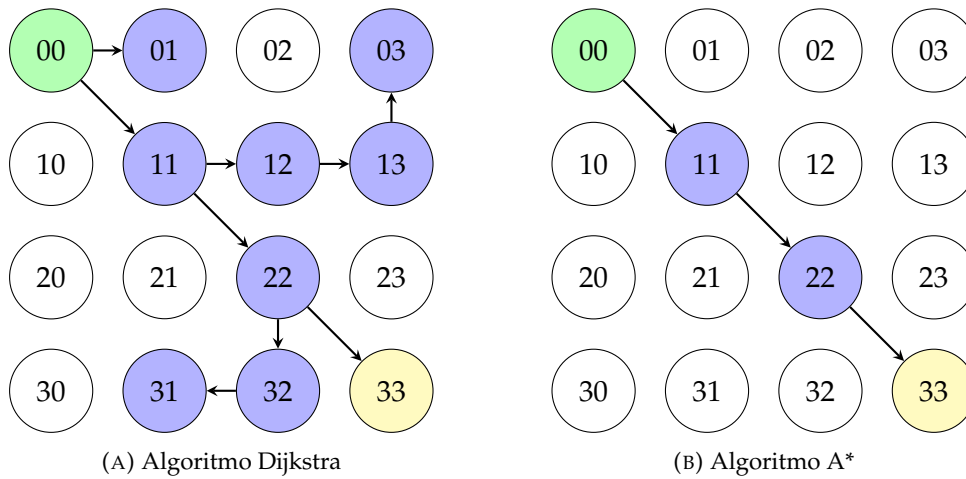


FIGURA 2.1: Comparativa entre algoritmos Dijkstra y A\*

10. Un trabajador sólo puede realizar una tarea al mismo tiempo.
11. Los tiempos de preparación de un trabajador antes de realizar una tarea son nulos.
12. Los tiempos de espera entre la realización de una tarea y otra son nulos.

## 2.2. Método a resolver

### 2.2.1. Algoritmo

Existen numerosos algoritmos capaces de resolver el problema del JSP. Estrategias más simples como listas ordenadas en función de la duración de los trabajos, algoritmos genéticos, técnicas gráficas, algoritmos *Branch and Bound* y heurísticos. En este estudio se utilizará un algoritmo heurístico, A\* (*A star*) [HNR68] para resolver el problema JSP.

El A\* es una evolución del algoritmo de Dijkstra. Su principal diferencia es la implementación de una función heurística que se utiliza para decidir el siguiente nodo a expandir. De esta forma, se podría decir que el algoritmo A\* va 'guiado' hacia la solución, mientras que el algoritmo de Dijkstra sigue los caminos con menor coste (Véase figura 2.1).

El algoritmo A\* utiliza varios componentes para resolver problemas de optimización. A continuación se describe cada uno de ellos.

### 2.2.1.1. Componentes A\*

#### 2.2.1.1.1. Estado

El estado es una estructura de datos que describe la situación del problema en un punto determinado. Estos estados deben ser comparables, debe ser posible dados dos estados conocer si son iguales o distintos. En el caso del JSP, el estado podría estar formado por la siguiente información:

- Instante de tiempo en el que comienza cada tarea planificada.
- Instante de tiempo futuro en el que cada trabajador estará libre (i.e. finaliza la tarea que estaba realizando).

El resultado final del JSP será un estado donde todas las tareas han sido planificadas.

#### 2.2.1.1.2. Costes

El algoritmo A\* utiliza 3 costes distintos para resolver el problema de optimización:

##### 2.2.1.1.2.1. Coste G

El coste G (de ahora en adelante  $cost_g$ ) es el coste desde el estado inicial hasta el estado actual. Este coste es calculado buscando el mayor tiempo de fin de las tareas ya planificadas.

##### 2.2.1.1.2.2. Coste H

El coste H (de ahora en adelante  $cost_h$ ) es el coste estimado desde el estado actual hasta el estado final. Este coste es calculado utilizando una función heurística, que estima el coste.

##### 2.2.1.1.2.3. Coste F

El coste F (de ahora en adelante  $cost_f$ ) es el coste estimado desde el estado inicial hasta el estado final pasando por el estado actual.

Por lo tanto,

$$cost_f = cost_g + cost_h$$

### 2.2.1.1.3. Generación de sucesores

El algoritmo  $A^*$  debe generar un número de estados sucesores dado un estado cualquiera <sup>2</sup>. Por lo que será necesario una función que dado un estado retorne un listado de estados.

Dado un estado donde existen  $N$  tareas por ejecutar ( $T_0 \dots T_N$ ) y un trabajador cualquiera sin tarea asignada tendrá  $N$  estados sucesores. En cada uno, el trabajador libre tendrá asignada cada una de las tareas, desde  $T_0$  hasta  $T_N$  <sup>3</sup>.

### 2.2.1.1.4. Listas de prioridad

El algoritmo  $A^*$  utiliza dos listas de estados: la lista abierta y la lista cerrada. La lista cerrada contiene los estados que ya han sido estudiados mientras que la lista abierta contiene los estados que aún están por estudiar.

Cada vez que se estudia un estado de la lista abierta, se obtienen sus sucesores que son añadidos a la lista abierta (siempre y cuando no estén en la lista cerrada) mientras que el estado estudiado pasa a la lista cerrada.

Los estados de la lista abierta están ordenados en función de su  $cost_f$ , de menor a mayor. De esta forma, se tiene acceso inmediato al elemento con menor  $cost_f$ .

---

<sup>2</sup>Dependiendo de la implementación, puede existir alguna excepción a esta norma, como el estado objetivo que puede no tener sucesores.

<sup>3</sup>Suponiendo que:

1. Todas las tareas  $T_x \in (T_0 \dots T_N)$  pueden ser ejecutadas por el trabajador libre.
2. Todas las tareas  $T_x \in (T_0 \dots T_N)$  pueden ser ejecutadas en este instante (e.g. cada una es de un trabajo diferente).

### 2.2.1.2. Pseudocódigo

```
1      lista_abierta = SortedList()
2      lista_abierta.append(estado_inicial)
3
4      g_costes = {}
5      f_costes = {}
6
7      g_costes[estado_inicial] = 0
8      f_costes[estado_inicial] = calcular_h_coste(estado_inicial)
9
10     while (not lista_abierta.empty()):
11         estado_actual = lista_abierta.pop()
12
13         if (estado_actual == estado_final):
14             return estado_actual
15
16         estados_sucesores = calcular_sucesores(estado_actual)
17
18         for estado_sucesor in estados_sucesores:
19             sucesor_g_coste = calcular_g_coste(estado_sucesor)
20             if (sucesor_g_coste < g_costes[estado_sucesor]):
21                 g_costes[estado_sucesor] = sucesor_g_coste
22                 f_costes[estado_sucesor] = sucesor_g_coste +
calcular_h_coste(estado_sucesor)
23                 if (estado_sucesor not in lista_abierta):
24                     lista_abierta.append(estado_sucesor)
```

### 2.2.2. Equipo de Estudio

Este algoritmo es implementado y optimizado en diversas arquitecturas. Posteriormente, se realizan comparaciones entre ellas.

#### 2.2.2.1. Arquitectura x86

Inicialmente, se realiza una implementación del algoritmo utilizando **Python**. Esta versión permite comprobar rápidamente el correcto funcionamiento del mismo así como llevar a cabo pruebas rápidas sin necesidad de compilación y estudiar los posibles cuellos de botella del algoritmo.

Posteriormente, se desarrolla una nueva versión del mismo algoritmo utilizando C++, un lenguaje compilado, imperativo y orientado a objetos que facilita la paralelización gracias a librerías como **OpenMP**.

Una vez desarrolladas ambas versiones monohilo, se comienza la implementación de versiones multihilo que serán posteriormente comparadas.

#### **2.2.2.2. FPGA**

Finalmente, se desarrolla una implementación del algoritmo diseñado para ser ejecutado en una FPGA. Esta aceleradora, se encuentra embebida en una placa SoC Zybo Z7 10 acompañada de un procesador ARM.

Para realizar esta implementación, se utiliza el software propio de Xilinx (AMD), Vitis HDL. Este programa ofrece entre muchas otras herramientas un sintetizador capaz de transpilar código C++ a Verilog que puede ser entonces compilado para ejecutarse en la FPGA.



## 2.3. Método de comparativas

Las comparativas entre las diferentes implementaciones del algoritmo se realizan en base a varias características. Principalmente:

1. Tiempo de ejecución.
2. Calidad de la solución.

Como es lógico, el algoritmo es ejecutado utilizando distintos datos de entrada múltiples veces.

### NOTA

La segunda ejecución de cualquier algoritmo suele tender a requerir menos tiempo debido al funcionamiento de la caché.

Para evitar este fenómeno, el algoritmo se ejecuta siempre  $N$  veces ignorando las métricas de la primera ejecución.

## 2.4. Implementación

Tanto **Python** como C++ son lenguajes orientados a objetos. Esta sección contiene las descripciones de las diferentes clases diseñadas para dar soporte al algoritmo.

### 2.4.1. Task

La clase `Task` corresponde a una tarea a realizar. Una instancia de esta clase está definida por los atributos:

- `unsigned int duration`: Duración de la tarea.
- `std::vector<int> qualified_workers`: Listado de trabajadores que pueden realizar la tarea.

### 2.4.2. State

La clase `State` corresponde a un estado (o nodo). Una instancia de esta clase está definida por los atributos:

- `std::vector<std::vector<Task>> jobs`: Lista de trabajos y tareas a ejecutar.
- `std::vector<std::vector<int>> schedule`: Planificación actual.
- `std::vector<int> workers_status`: Instantes en los que cada trabajador queda libre.

#### NOTA

Nótese que el atributo `std::vector<std::vector<Task>> jobs` será el mismo en todos los estados de un mismo problema. Por lo que no será necesario revisarlo en `State::operator==` ni `State::operator()`. Si el consumo de memoria fuese de importancia, sería posible utilizar una referencia para evitar almacenar esta estructura múltiples veces.

El algoritmo A\* requiere que se creen estructuras de datos que contendrán instancias de la clase `State`. Estas estructuras necesitan que se proporcionen implementaciones para los operadores `State::operator==` y `State::operator()` de la clase `State`. Para diseñar las implementaciones de estos operadores se estudian previamente los atributos que componen la clase `State`:

- `std::vector<std::vector<Task>> jobs`: Es igual en todas las instancias de `State`, por lo que será ignorado.
- `std::vector<std::vector<int>> schedule`: Proporciona información crucial sobre el estado ( $cost_g$  y  $cost_h$ ).

- `std::vector<int> workers_status`: No proporciona información alguna sobre los costes, pero es necesario para distinguir dos estados diferentes ya que es posible que dos estados tengan los mismos costes pero a través de planificaciones distintas.

Por ello, será necesario definir dos operadores `State::operator()`: uno que sea indiferente al atributo `std::vector<int> workers_status` (`StateHash::operator()`) y otro que sí lo tenga en cuenta para distinguir diferentes instancias de `State` (`FullHash::operator()`).

## 2.5. Optimización

### 2.5.1. A\*

La siguiente subsección estudia la optimización del algoritmo A\* sin tener en cuenta el paralelismo, esto es, se trata de optimizar el rendimiento monohilo del mismo.

#### 2.5.1.1. State

La operación `operator()` es ejecutada varias veces para cada `State`, este método tiene una complejidad de  $O(n^2)$ , por lo que su valor se almacena tras calcularlo por primera vez en un atributo del propio `State`.

Los operadores `operator==` necesarios se implementan utilizando los `operator()` correspondientes. Las funciones hash utilizadas en los `operator()` son resistentes a colisiones, esto es,  $h(State_a) \neq h(State_b) \iff State_a \neq State_b$  por lo que se pueden utilizar para comparar elementos en `operator==`.

### 2.5.1.2. Coste H - Heurístico

La principal decisión que afectará al tiempo de ejecución del algoritmo se encuentra en la Implementación de la función heurística encargada de calcular el coste H. Este coste se utiliza para seleccionar el siguiente nodo a expandir, por lo que un buen heurístico es aquel que mejor dirige al algoritmo en la dirección del nodo objetivo.

El rendimiento y calidad del resultado del algoritmo dependerán en gran medida de la función seleccionada. En algunos casos la implementación retornará resultados óptimos (o cercanos al óptimo) pero requerirá un mayor tiempo de ejecución, mientras que otras implementaciones requerirán un menor tiempo de ejecución pero sus resultados no serán óptimos. Dependiendo del problema a resolver será conveniente implementar una función heurística de un tipo u otro.

#### 2.5.1.2.1. Heurístico para optimalidad

La siguiente implementación de la función heurística dirigirá al algoritmo hacia nodos solución que sean óptimos o se encuentren relativamente cerca del óptimo.

```
1 unsigned int State::calculate_h_cost() const
2 {
3     std::vector<int> h_costs;
4     for (size_t job_idx = 0; job_idx < this->m_jobs.size(); job_idx++)
5     {
6         h_costs.emplace_back(0);
7         std::vector<Task> job = this->m_jobs[job_idx];
8         for (size_t task_idx = 0; task_idx < job.size(); task_idx++)
9             if (this->get_schedule()[job_idx][task_idx] == -1)
10                 h_costs[job_idx] += job[task_idx].get_duration();
11     }
12     auto max_element = std::max_element(h_costs.begin(), h_costs.end());
13     ;
14     return max_element == h_costs.end() ? 0 : *max_element;
15 }
```

La función calcula el tiempo necesario para completar cada trabajo y retorna el tiempo mayor.

### 2.5.1.2.2. Heurístico para tiempo

La siguiente implementación de la función heurística dirigirá al algoritmo hacia cualquier nodo solución independientemente de si es óptimo o no.

```
1 unsigned int State::calculate_h_cost() const
2 {
3     unsigned int unscheduled_tasks_count = 0;
4     for (std::size_t job_idx = 0; job_idx < this->m_jobs.size();
5         job_idx++)
6         for (std::size_t task_idx = 0; task_idx < this->m_jobs[job_idx]
7             .size(); task_idx++)
8             if (this->m_schedule[job_idx][task_idx] == -1)
9                 unscheduled_tasks_count += this->m_jobs[job_idx][
10                    task_idx].get_duration();
11     return unscheduled_tasks_count;
12 }
```

La función calcula el tiempo necesario para completar las tareas restantes si se ejecutasen una a una y retorna esta suma.

#### NOTA

A pesar de que ambas implementaciones tienen la misma complejidad ( $O(n^2)$ ), un algoritmo A\* utilizando de ellas tardará varias magnitudes de tiempo más que si utilizase la otra aunque retornará resultados notablemente mejores en algunos casos.

### 2.5.2. Paralelización

En la siguiente subsección se estudia la paralelización del algoritmo A\*. Este estudio está compuesto por la descripción y comparación de distintas alternativas discutidas en la literatura.

#### 2.5.2.1. First Come First Serve (FCFS) Solver

Sería posible desarrollar una implementación que paralelice el procesamiento de los nodos, asignando uno a cada hilo de forma que para  $N$  hilos se procesen  $N$  nodos de forma simultánea. Esta estrategia permite que los hilos procesen los nodos a medida que entran en el `open_set` (Véase Figura 2.2).

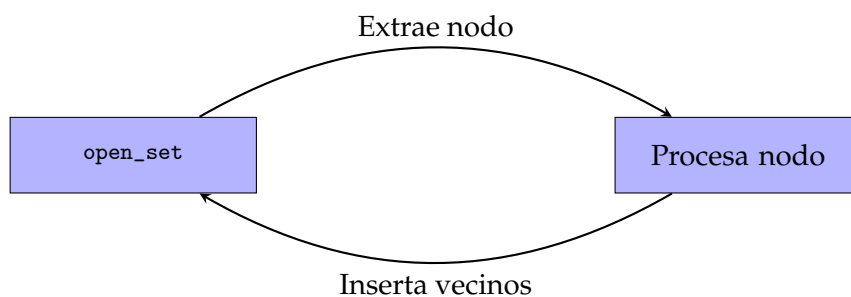


FIGURA 2.2: Representación de la estrategia FCFS

De cualquier forma, este diseño en particular no tiene por qué reducir el tiempo requerido para hallar un nodo solución, simplemente tiene la oportunidad de reducirlo en algunos casos específicos. Esto se debe a que la única diferencia entre las versiones monohilo y multihilo es que en la multihilo se procesan más nodos en el mismo tiempo. (Véase Figura 2.3)

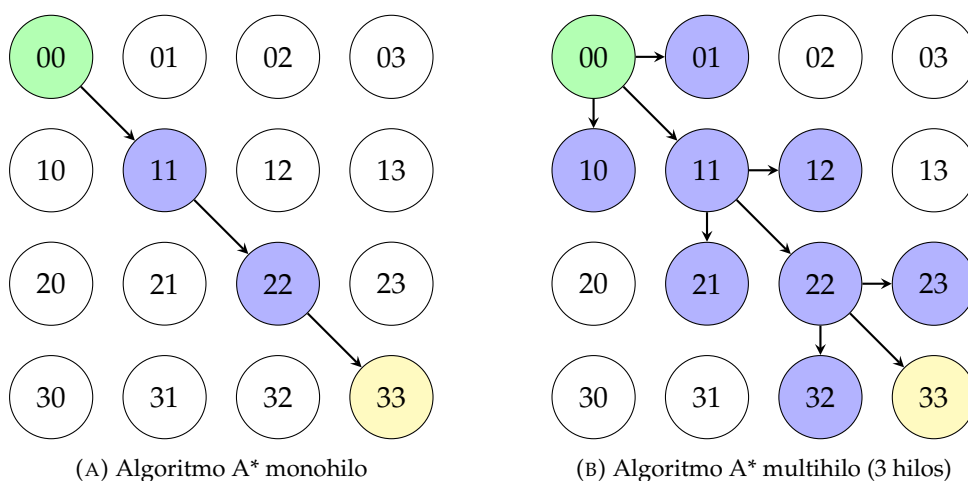


FIGURA 2.3: Comparativa entre algoritmos monohilo y multihilo

**NOTA**

Nótese que este acercamiento no tiene sincronización entre diferentes iteraciones, esto implica que la solución está sujeta a una condición de carrera (i.e. la solución depende de qué hilo finalice primero su ejecución). Por lo que sería posible ejecutar  $N$  veces este algoritmo y obtener  $N$  soluciones diferentes.

**2.5.2.1.1. Secciones críticas**

El diseño paralelo propuesto no es sin inconvenientes, su implementación contiene varias secciones críticas que suponen una amenaza para el rendimiento del algoritmo. A continuación se observan cada una de estas secciones y se analizan las razones por las cuales son necesarias.

**2.5.2.1.1.1. Variables de control de flujo**

Primero, al paralelizar el algoritmo siguiendo esta estrategia, se han añadido variables de control compartidas por todos los hilos que sirven para conocer si se ha resuelto el problema o no (una variable donde se copia el resultado y otra que sirve como *flag*). El acceso a estas variables debe estar controlado para evitar el acceso simultáneo a las mismas. De cualquier forma, es improbable que dos hilos tengan la necesidad de acceder esta sección crítica ya que sólo se ejecuta una vez por lo que los efectos en el rendimiento serán nulos. Sería necesario que dos hilos hallasen dos soluciones diferentes al problema al mismo tiempo.

**2.5.2.1.1.2. `open_set`**

Segundo, el acceso al `open_set` también debe estar controlado de forma que sólo un hilo pueda interactuar con la estructura de datos compartida. Esta interacción se presenta al menos en dos instancias por cada iteración del bucle principal: una primera vez para acceder al nodo a procesar y otra para insertar los nuevos vecinos. Si bien la obtención del nodo a procesar se realiza en  $O(1)$  ya que el `open_set` está ordenado y siempre se accede al nodo en la cabeza de la lista, la inserción de vecinos no corre la misma suerte. Para que la lectura del nodo a procesar sea en  $O(1)$  el `open_set` se mantiene ordenado, esto implica que la inserción se haría en  $O(n)$ <sup>4</sup>.

**NOTA**

La complejidad de la sección crítica en la que se insertan elementos en el `open_set` tiene como factor la longitud del `open_set`. Esto es, a mayor tamaño tenga el `open_set`, mayor tiempo será necesario para resolver la sección crítica.

Nótese que a medida que avanza el programa, el tamaño del `open_set` crece, incrementando la duración de la sección crítica y reduciendo la paralelización del algoritmo.

<sup>4</sup>Esta implementación utiliza iteradores y `std::deque<T>` para hallar la posición de cada nuevo elemento e insertarlo en el mismo barrido.



### 2.5.2.2. Batch Solver

La siguiente estrategia es una evolución del FCFS Solver anterior, utiliza el mismo principio (los hilos exploran nodos del `open_set` a medida que éste se va llenando), pero en este caso se implanta una barrera de sincronización en cada iteración. Esta barrera obliga a los hilos a esperar al resto de sus compañeros antes de extraer otro nodo del `open_set`.

La diferencia más notable entre este acercamiento y el anterior es que las secciones críticas se ven reducidas porque las secciones paralelas son menores. Además, al sincronizar los hilos en cada iteración, ahora no existe ninguna condición de carrera que pueda alterar el resultado, por lo que para el mismo problema este algoritmo siempre retornará el mismo resultado y utilizará la misma ruta para llegar a él.

Se utiliza un `std::vector<State>` para almacenar los nodos a explorar por cada hilo y un `std::vector<std::vector<State>>` para que cada hilo almacene los vecinos que ha encontrado. Cada uno de estos vectores tiene una longitud igual al número de hilos de forma que el hilo con ID  $N$  tiene asignada la posición  $N$  de cada vector. Véase figura 2.4.

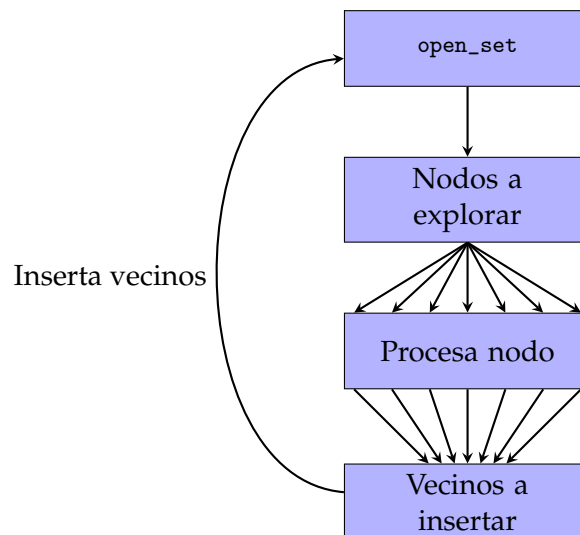


FIGURA 2.4: Representación de la estrategia Batch

#### 2.5.2.2.1. Secciones críticas

A diferencia de la estrategia FCFS, no es posible que dos hilos accedan al mismo recurso de forma simultánea. Las únicas estructuras de datos compartidas (los dos `std::vector`) ofrecen a los hilos un índice privado al que acceder. Las secciones críticas de la estrategia FCFS ahora son ejecutadas por un hilo: una al registrar los nodos a explorar y otra al retirar los vecinos e insertarlos en el `open_set`.

### 2.5.2.3. Recursive Solver

La estrategia propuesta en [Zag+17] se basa en la ejecución simultánea de varias instancias del algoritmo A\* en el mismo problema.

1. Calcular vecinos del estado inicial.
2. Asignar un hilo a cada vecino.
3. Para cada vecino, resolver el problema como si fuese el estado inicial.
4. Recoger resultados obtenidos.
5. Obtener resultado con menor coste.
6. Retornar.

Al igual que la solución por batches, no existe ninguna condición de carrera que permita al algoritmo retornar resultados diferentes en varias ejecuciones. Originalmente cada hilo utiliza sus propias estructuras de datos, por lo que no existen secciones críticas. No obstante, sería posible hacer una versión en la que los distintos hilos compartiesen las estructuras de costes y el `open_set`. Implementar el algoritmo de esta forma añadiría las tediosas secciones críticas que podrían ser más dañinas que beneficiosas.

Este diseño puede ser de gran interés para otros problemas distintos al JSP donde existan varios estados iniciales, ya que sería posible calcular una solución del A\* para cada uno de ellos. El algoritmo permitiría conocer el mejor estado inicial así como la ruta a seguir para llegar al estado objetivo.

Uno de los posibles problemas que puede presentar esta estrategia consiste en la posibilidad de que dos hilos terminen calculando caminos muy similares. Esto implica que uno de los hilos ha malgastado un tiempo que podría haber sido invertido en rutas diferentes. Para resolver este problema, sería necesario que los hilos compartiesen alguna estructura de datos para que sean conscientes de lo que está calculando cada uno.

#### 2.5.2.4. Hash Distributed A\* (HDA\*) Solver

El algoritmo propuesto en [KFB09] utiliza tantos `open_set` como hilos y una función hash para asignar cada nodo a uno de los `open_set`. Cada hilo es 'propietario' de uno de los `open_set` y por consiguiente, de los nodos que estén contenidos dentro del mismo. Cada hilo está encargado de explorar los nodos de su `open_set` y de añadir sus vecinos al `open_set` correspondiente.

El rendimiento de este acercamiento depende en gran medida de la función hash que se utilice para distribuir a los diferentes nodos. Una función hash que no sea uniforme<sup>5</sup> distribuirá los nodos de forma poco equitativa sobrecargando algunos hilos. Por otro lado, al utilizar varios `open_set`, el tiempo de inserción es menor porque tienen un menor tamaño.

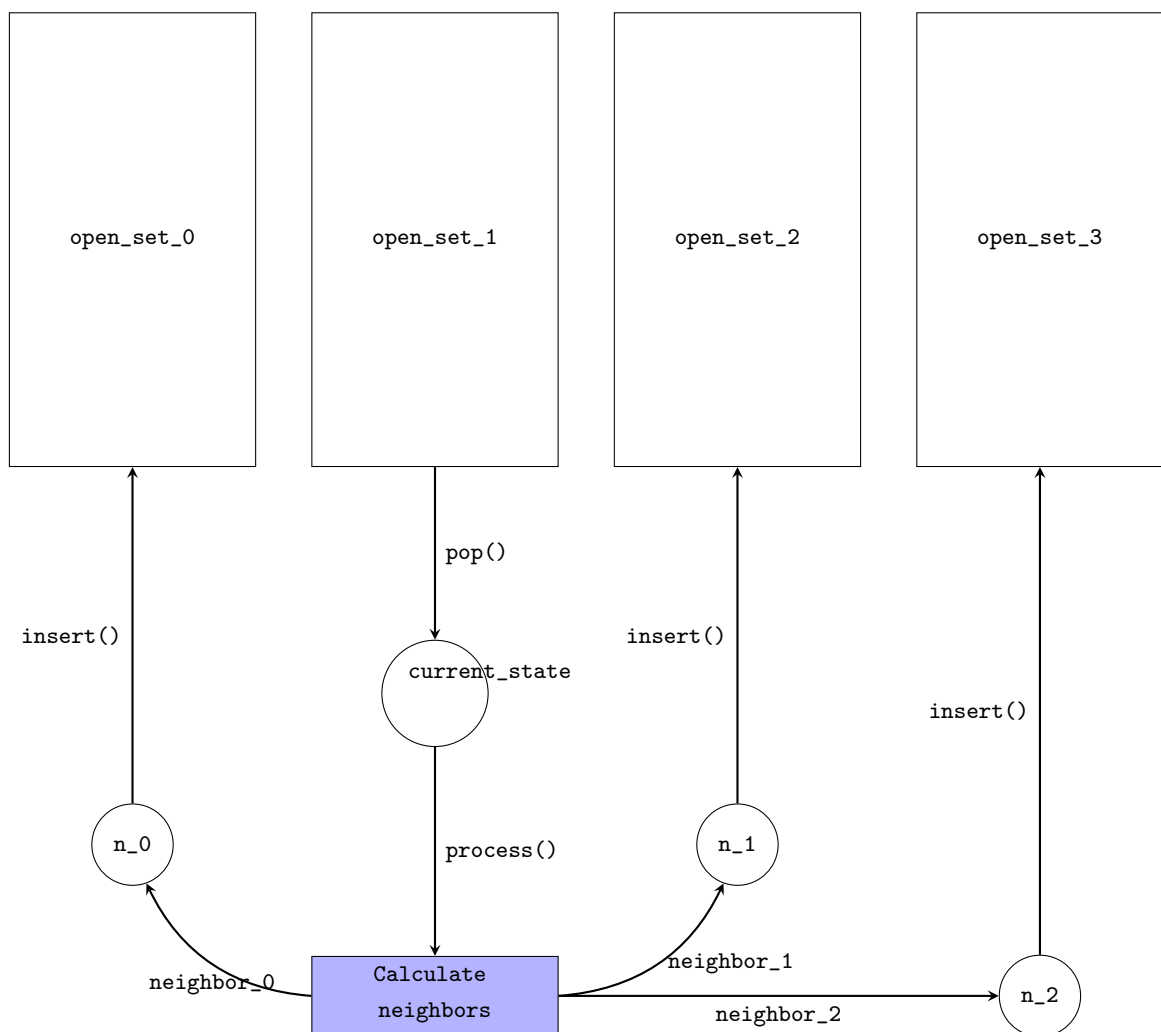


FIGURA 2.5: Representación de la estrategia HDA\*

<sup>5</sup>Una función hash es uniforme si los valores que retorna tienen la misma probabilidad de ser retornados.

## Capítulo 3

# Experimentos

### *Trabajo y Resultados*

### 3.1. Conjuntos de datos

Los diferentes conjuntos de datos utilizados para medir el rendimiento de las distintas implementaciones han sido obtenidos de **(HTTP) Jobshop Instances** o diseñados a mano.

Para desarrollar el programa se han utilizado conjuntos de datos personalizados hechos a mano para facilitar la creación de pruebas de software que comprueben el correcto funcionamiento del algoritmo.

Para ejecutar las pruebas se han utilizado porciones de conjuntos de datos obtenidos de **(HTTP) Jobshop Instances**. En particular se ha utilizado el 50% del conjunto 'abz5' (5x5) o el 100% de 'ft06' (6x6).

#### NOTA

Un conjunto de datos tiene un tamaño de  $A \times B$  cuando está compuesto por  $A$  trabajos y  $B$  tareas.

### 3.2. Equipos de Prueba

#### 3.2.1. Arquitectura x86

Se han utilizado 3 equipos diferentes para tomar y contrastar las mediciones.

##### 3.2.1.1. OS

ID	Tipo	OS	Versión
0	Escritorio	Arch Linux	6.9.6-arch1-1
1	Servidor	Debian Linux 12 'Bookworm'	6.1.0-16-amd64
2	Portátil	Debian Linux 12 'Bookworm'	6.1.0-16-amd64

CUADRO 3.1: Equipos de prueba, arquitectura x86: OS

##### 3.2.1.2. CPU

ID	Familia	Modelo	Hilos	Reloj
0	Intel	Core I7-9700F	8c8t	4.7GHz
1	Intel	Xeon-E5450	4c4t	3GHz
2	AMD	Ryzen 7 5700U	8c16t	4.3GHz

CUADRO 3.2: Equipos de prueba, arquitectura x86: CPU

##### 3.2.1.3. RAM

ID	RAM	Formato	Tipo	Reloj
0	64GB	4x16	DDR4	3200MHz
1	8GB	2x4	DDR3	2666MHz
2	16GB	2x8	DDR4	3200MHz

CUADRO 3.3: Equipos de prueba, arquitectura x86: RAM

### 3.3. Método de medición

Todas las versiones imprimen por salida estándar datos que posteriormente son procesados en formato CSV:

- Lenguaje
- Número de hilos
- Porcentaje del trabajo resuelto
- Trabajos
- Tareas
- Trabajadores
- Tiempo de ejecución
- Planificación
- *Makespan*

La alta complejidad del JSP implica que un mínimo aumento en el tamaño del problema puede implicar que el algoritmo no finalice su ejecución en un tiempo polinomial. Por ello, en lugar de tomar una única medición al inicio y final de la ejecución se ha optado por utilizar un objeto personalizado que toma mediciones en intervalos predefinidos. De esta forma, de una única ejecución se podría obtener:

- Tiempo de inicio.
- Tiempo necesario para resolver el 10% del problema.
- Tiempo necesario para resolver el 20% del problema.
- ...
- Tiempo necesario para resolver el 90% del problema.
- Tiempo necesario para resolver el 100% del problema.

```

1  class Chronometer
2  {
3  private:
4      std::chrono::_V2::system_clock::time_point m_start_time;
5      std::map<unsigned short, bool> m_goals;
6      std::map<unsigned short, double> m_times;
7      std::string m_solver_name;
8
9  public:
10     Chronometer() : Chronometer(
11         std::map<unsigned short, bool>(),
12         "Unknown Solver") {}
13     explicit Chronometer(
14         const std::map<unsigned short, bool> &goals,
15         std::string const &solver_name) : m_goals(goals),
16                                             m_solver_name(solver_name) {}
17
18     void start()
19     {
20         this->m_start_time = std::chrono::high_resolution_clock::now();
21     }
22     std::chrono::duration<double> time() const
23     {
24         return (
25             std::chrono::high_resolution_clock::now() -
26             this->m_start_time
27         );
28     }
29
30     void process_iteration(const State &state);
31     void log_timestamp(unsigned short goal, const State &state);
32     void enable_goals();
33     std::map<unsigned short, double> get_timestamps() const;
34 };

```

Esta clase se utiliza para tomar las mediciones de todas las iteraciones <sup>1</sup> y posteriormente estos datos se utilizan para estudiar el rendimiento de cada implementación.

---

<sup>1</sup>Ignorando la primera para calentar la caché

### 3.4. Métricas

A continuación se listan las diferentes métricas observadas a la hora de estudiar y criticar las implementaciones del algoritmo:

- Tiempo de ejecución - Menor implica mejor.
- *Speedup* - Mayor implica mejor.
- *Makespan* - Menor implica mejor.
- Consumo de CPU <sup>2</sup>.

#### 3.4.1. *Speedup*

Sean  $T_1$  y  $T_0$  dos métricas del tiempo de ejecución de dos algoritmos distintos ( $A_0$  y  $A_1$ ), el *speedup* del algoritmo  $A_0$  respecto al  $A_1$  ( $S_{A_0,A_1}$ ) es:

$$S_{A_0,A_1} = T_1/T_0$$

#### EJEMPLO

Si el algoritmo  $A_0$  tardó 5s en finalizar y  $A_1$  tardó 10s, el *speedup* ( $S_{A_0,A_1}$ ):

$$S_{A_0,A_1} = T_1/T_0 = 10/5 = 2$$

$A_0$  es el doble de rápido que  $A_1$ .

Por lo tanto:

$$S_{A_0,A_1} = 1 \rightarrow T_1 = T_0$$

$$S_{A_0,A_1} > 1 \rightarrow T_1 > T_0$$

$$S_{A_0,A_1} < 1 \rightarrow T_1 < T_0$$

#### 3.4.2. *Makespan*

El *makespan* es el tiempo necesario para completar todos los trabajos de un conjunto de datos, es el 'resultado' que genera el algoritmo.

Lógicamente, existe un *makespan* óptimo para cada conjunto de datos, el mínimo. No obstante, el algoritmo desarrollado en este proyecto no tiene la certeza de obtener ese resultado siempre. Por ello, si una implementación  $A$  retorna resultados con un *makespan* que otra  $B$ ,  $A$  es mejor que  $B$  <sup>3</sup>.

<sup>2</sup>Dependiendo de otras métricas, el consumo de CPU se puede utilizar para 'desempatar' algoritmos que aparentemente tengan los mismos resultados, en cuyo caso menor implica mejor.

<sup>3</sup>Suponiendo que el resto de métricas entre  $A$  y  $B$  son lo suficientemente similares.



### 3.5. Resultados y Análisis

#### 3.5.1. Complejidad del problema

En la siguiente gráfica (figura 3.1), se puede ver el tiempo de ejecución necesario para completar el problema utilizando uno de los algoritmos.

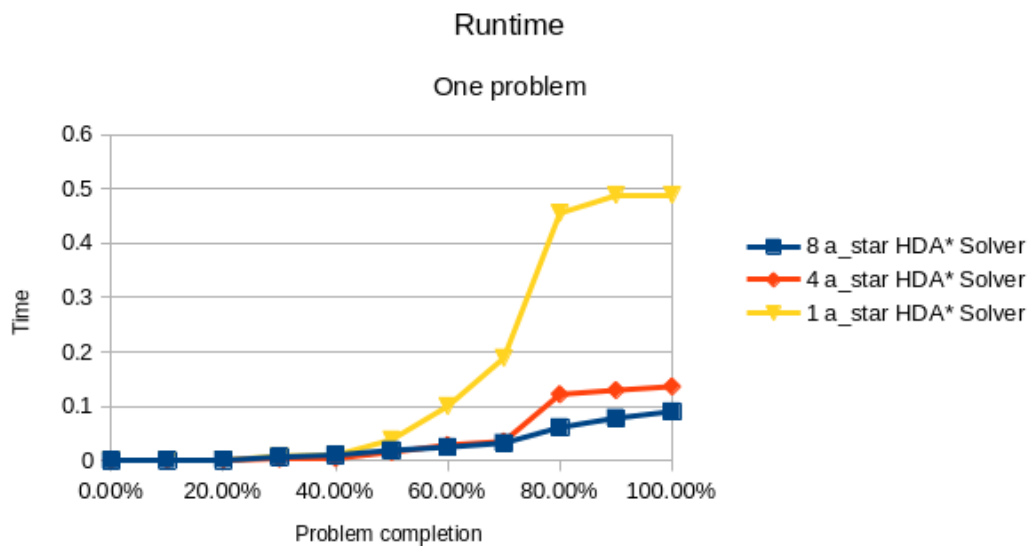


FIGURA 3.1: Tiempo de ejecución de un único problema.

Como se puede ver, el diagrama es de poca utilidad debido a la complejidad del problema. Para poder observar con mejor los resultados, se utiliza un eje vertical con una escala logarítmica (figura 3.2).

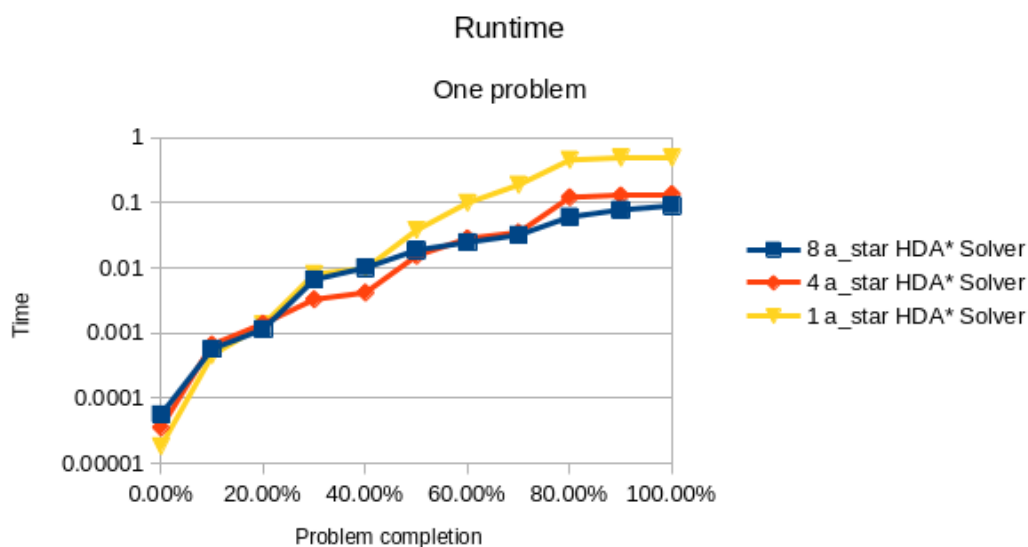


FIGURA 3.2: Tiempo de ejecución de un único problema (escala logarítmica).

Estas observaciones verifican que la complejidad del problema a resolver es cuadrática, como era de esperar.

#### NOTA

Esta sección contiene diversas gráficas de las métricas obtenidas, obsérvese con detalle la escala utilizada en el eje de ordenadas de cada una ya que en algunos casos será logarítmica.

### 3.5.2. Comparativa de heurísticos

Como ya se discutió en la sección sobre optimización (2.5.1.1), existen varias implementaciones de la función heurística que da al algoritmo A\* su particular comportamiento de ir ‘dirigido’ hacia la solución. En esta investigación se han implementado dos heurísticos: uno de ellos busca una solución que se acerque a la óptima lo máximo posible mientras que el otro busca la solución priorizando la velocidad del algoritmo. A continuación se comparan los heurísticos utilizando la implementación paralela HDA\*.

En primer lugar se observa que existe un limitado número de soluciones propuestas, las más comunes siendo también las más bajas: 452 y 472. Respecto al tiempo de ejecución, las pruebas que utilizan el heurístico rápido reducen el tiempo de ejecución en varias magnitudes.

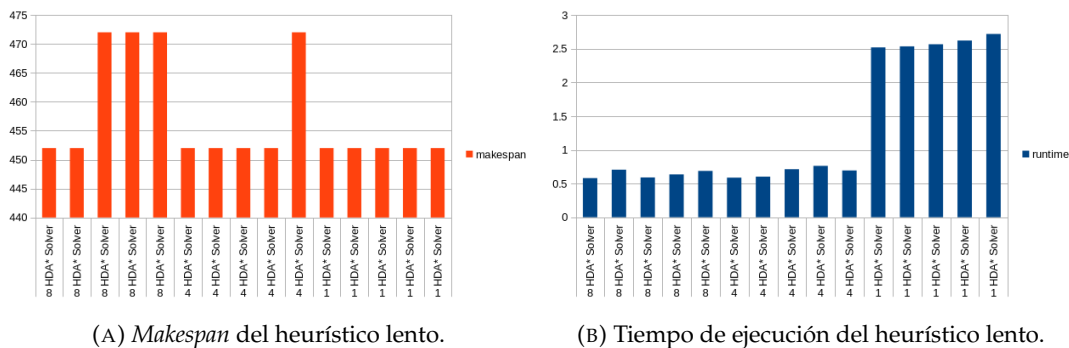


FIGURA 3.3: Métricas heurístico lento.

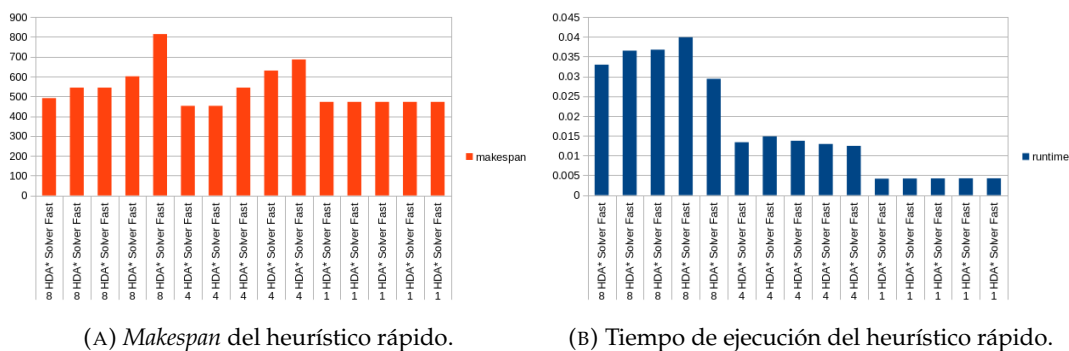


FIGURA 3.4: Métricas heurístico rápido.

Analizándolas por separado, las ejecuciones que utilizan el heurístico lento retornaron siempre 452 o 472, los dos mejores resultados y se vieron beneficiadas por

el uso de varios hilos. Por otro lado, las ejecuciones que utilizan el heurístico rápido sólo retornaron 452 o 472 en algunas instancias y no se vieron beneficiadas por el uso de varios hilos (Véase figuras 3.3 y 3.4).

Sería razonable suponer que el heurístico rápido sí se vería beneficiado por el paralelismo si el tamaño del problema fuese lo suficientemente grande. Para comprobar esta hipótesis, se ha creado un conjunto con un tamaño mucho mayor (70x10) y se ha obtenido el *speedup*.

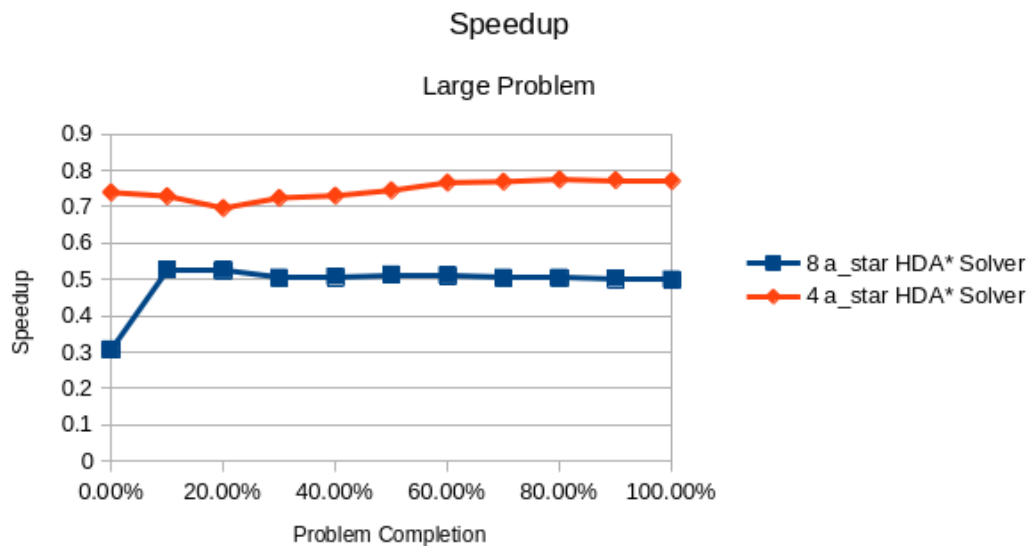


FIGURA 3.5: *Speedup* en un problema de gran tamaño.

La gráfica (3.5) muestra que incluso en un problema de mayor tamaño la versión monohilo es más rápida que las multihilo. Esta investigación no ha podido encontrar un conjunto de datos en el que utilizando el heurístico rápido valga la pena el paralelismo. No obstante, se ha observado que a medida que el tamaño del problema incrementa, el *speedup* también se ve incrementando por lo que si se supone que la tendencia del *speedup* se mantiene, sería razonable suponer que existe un tamaño de problema donde sí vale la pena utilizar varios hilos y el heurístico rápido.

### 3.5.3. Cuellos de botella en secciones críticas

Los algoritmos que utilizan estructuras de datos compartidas para almacenar los nodos y sus costes se ven gravemente afectadas cuando el tamaño de estas estructuras incrementa. Como esta información es compartida por todos los hilos, es necesario acceder a ella de forma serializada, reduciendo notablemente el *speedup*.

En algunos casos extremos es posible incluso que versiones monohilo del mismo algoritmo tengan mejor rendimiento que versiones paralelas. Nótese que el *speedup* del algoritmo FCFS cuando se utilizan varios hilos (respecto a un hilo solo) es inferior a 1. (Véase figura 3.6).

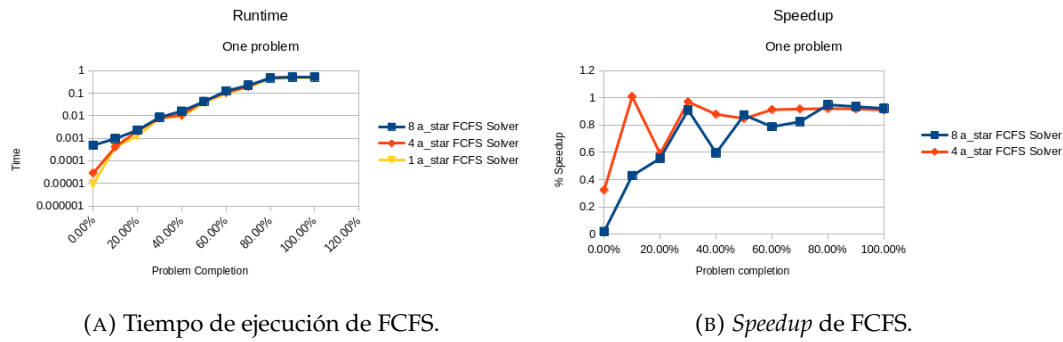
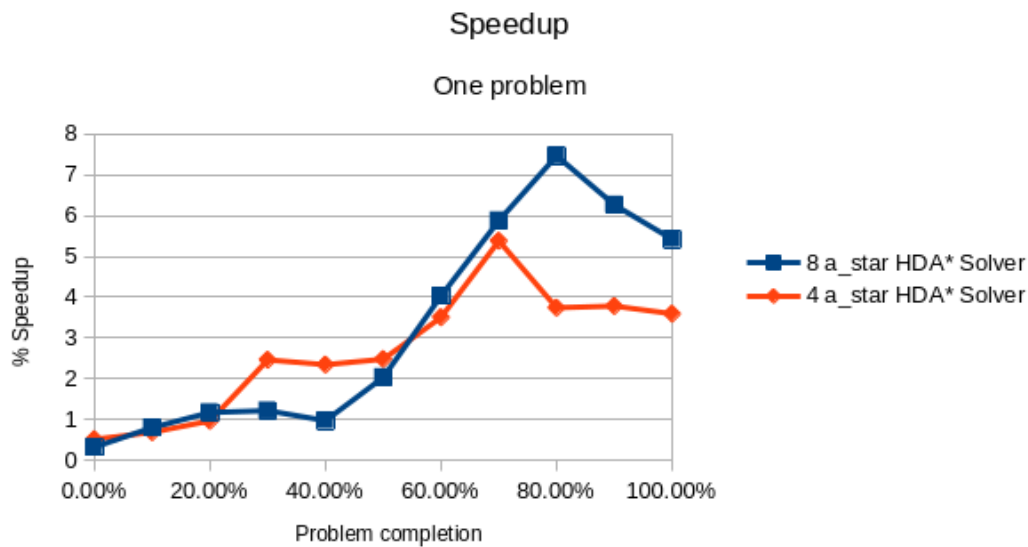


FIGURA 3.6: Métricas de paralelismo en FCFS.

Por otro lado, algoritmos como el HDA\* que utilizan una estructura de datos privada para cada hilo no ven sus tareas serializadas, incrementando notablemente el *speedup* (Véase figura 3.7).

FIGURA 3.7: *Speedup* de HDA\*.

Nótese que al inicio del problema (aproximadamente hasta completar el 30%) la versión monohilo de los algoritmos es más rápida que cualquier multihilo. Esto se debe a que para tamaños de problema muy pequeños el coste de crear  $N$  hilos es superior al de resolver el problema utilizando uno sólo.

Si se observa el *speedup* al final del problema, la versión con cuatro hilos tiene un *speedup* de 3,5 mientras que la de ocho tiene 5,5. Esto implica que al utilizar cuatro hilos, el algoritmo ha sido capaz de aprovecharlos casi al máximo ya que el tiempo de ejecución casi se reduce en 4 veces. Mientras tanto, al utilizar 8 hilos el algoritmo no ha sido capaz de rentabilizarlos en la misma proporción. Este déficit podría deberse a que el tamaño del problema es demasiado pequeño para aprovechar la cantidad de hilos o podría deberse al propio diseño del algoritmo.

### 3.5.4. Comparativa de algoritmos

Como es de esperar, el rendimiento monohilo de todas las implementaciones es el mismo. Todas las implementaciones están diseñadas de forma que al ser ejecutadas con un sólo hilo el algoritmo sea el A\* básico.

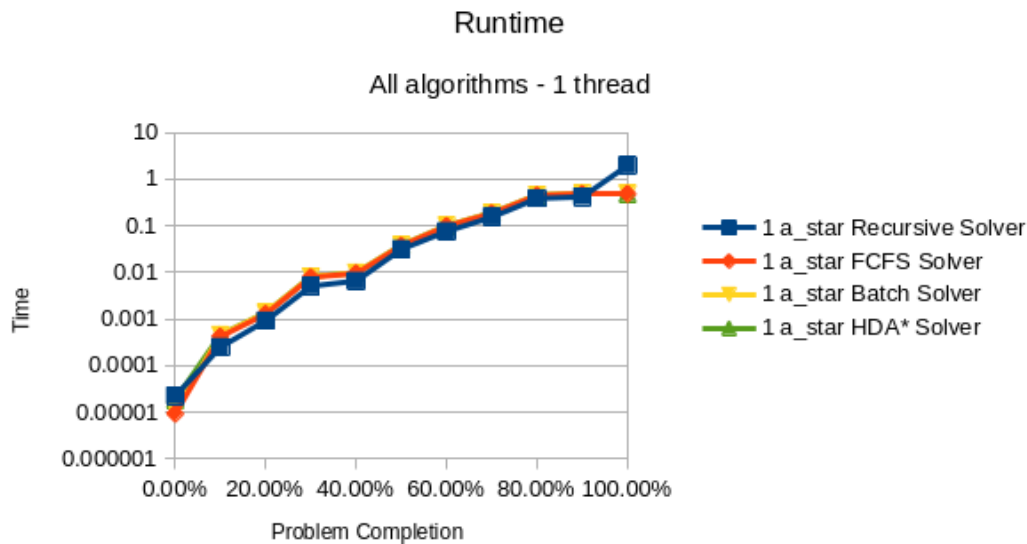
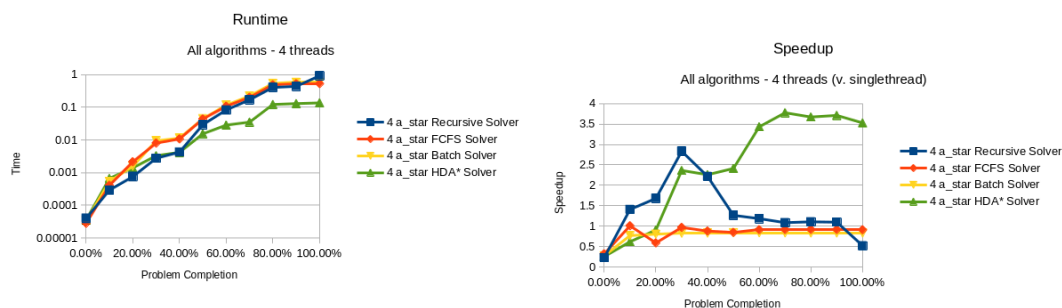


FIGURA 3.8: Tiempo de ejecución de todos los algoritmos (1 hilo).



(A) Tiempo de ejecución de todos los algoritmos (4 hilos).

(B) *Speedup* de todos los algoritmos (4 hilos).

FIGURA 3.9: Métricas con 4 hilos.

Al utilizar cuatro hilos (figura 3.9), se puede comenzar a ver una diferencia clara en el rendimiento del algoritmo HDA\*, que obtiene un *speedup* de casi 4.

Al utilizar ocho hilos (figura 3.10), el algoritmo HDA\* incrementa aún más su diferencia en el tiempo de ejecución con respecto al resto de algoritmos, aunque esta vez el *speedup* fluctúa más. De cualquier forma, parece ser capaz de alcanzar casi 8.

La principal conclusión de estas observaciones es que (al menos en los conjuntos de datos observados) el paralelismo es sólo rentable si se implementa el HDA\*. En el resto de casos, el paralelismo sólo sirve para gastar núcleos y ciclos de CPU a cambio de nada.

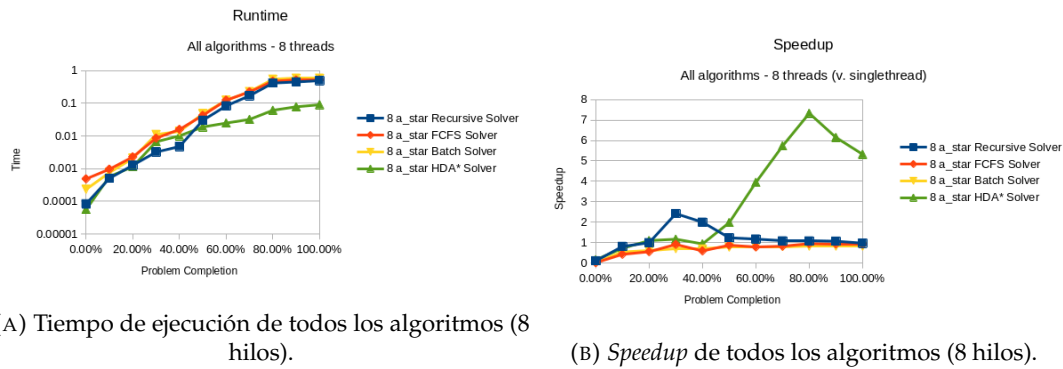


FIGURA 3.10: Métricas con 8 hilos.

### 3.5.5. Casos particulares

Vistos los resultados anteriores se podrían dar como obsoletas algunas de las versiones paralelas por ofrecer muy pocas mejoras frente a otras versiones monohilo. No obstante, existen casos particulares del problema donde estas versiones fácilmente descartables podrían presentar una solución mucho más interesante.

Estos casos particulares generalmente involucran la posibilidad de que la población de nodos sea repartida entre los diferentes hilos de forma que cada uno tenga una sección parcial o totalmente independiente del resto. A continuación se presentan algunos de estos casos.

#### 3.5.5.1. Varios estados iniciales

Si el problema a resolver tiene varios estados iniciales sería posible asignar cada estado a un hilo (o grupo de hilos) de forma que cada uno busque una solución desde su estado inicial.

#### 3.5.5.2. Estados solución intermedios

Si se conoce algún nodo intermedio de la solución sería posible dividir el problema en dos, de forma que un hilo resuelva una de las partes <sup>4</sup>. Por ejemplo, si del problema se conocen el nodo inicial  $A$ , el final  $E$  y los intermedios  $B$ ,  $C$  y  $D$ , la solución se podría obtener repartiendo el trabajo entre 4 hilos diferentes:

1. Hilo 0: Resolver camino desde nodo  $A$  hasta  $B$ .
2. Hilo 1: Resolver camino desde nodo  $B$  hasta  $C$ .
3. Hilo 2: Resolver camino desde nodo  $C$  hasta  $D$ .
4. Hilo 3: Resolver camino desde nodo  $D$  hasta  $E$ .

<sup>4</sup>Si existiese más de un nodo intermedio conocido, el problema se podría seguir subdividiendo entre más hilos.

## Capítulo 4

# Conclusiones

*Observaciones y Trabajos futuros*

## **Apéndice A**

# **Código Fuente**



## **Apéndice B**

# **Resultados y Métricas**

# Bibliografía

- [Man67] G. K. Manacher. «Production and Stabilization of Real-Time Task Schedules». En: *J. ACM* 14.3 (1967), 439–465. ISSN: 0004-5411. DOI: [10.1145/321406.321408](https://doi.org/10.1145/321406.321408). URL: <https://doi.org/10.1145/321406.321408>.
- [HNR68] Peter E. Hart, Nils J. Nilsson y Bertram Raphael. «A Formal Basis for the Heuristic Determination of Minimum Cost Paths». En: *IEEE Transactions on Systems Science and Cybernetics* (jul. de 1968), págs. 100 -107. URL: <https://ieeexplore.ieee.org/abstract/document/4082128/authors#authors>.
- [Nil69] Nils J. Nilsson. «Problem solving methods in Artificial Intelligence». En: *Stanford Research Institute* (1969). URL: <https://stacks.stanford.edu/file/druid:xw061vq8842/xw061vq8842.pdf>.
- [Yan77] Yoo Baik Yang. «Methods and Techniques used for Job Shop Scheduling». Methods and Techniques, Analytical Techniques. Masters Thesis. College of Engineering of Florida Technological University, 1977. URL: <https://stars.library.ucf.edu/cgi/viewcontent.cgi?article=1389&context=rtd>.
- [Nil84] Nils J. Nilsson. «Shakey the Robot». En: *SRI International* (1984). URL: <http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/shakey-the-robot.pdf>.
- [KTM99] Joachim Käschel, Tobias Teich y B. Meier. «Algorithms for the Job Shop Scheduling Problem - a comparison of different methods». En: (ene. de 1999). URL: [https://www.researchgate.net/publication/240744093\\_Algorithms\\_for\\_the\\_Job\\_Shop\\_Scheduling\\_Problem\\_-\\_a\\_comparison\\_of\\_different\\_methods](https://www.researchgate.net/publication/240744093_Algorithms_for_the_Job_Shop_Scheduling_Problem_-_a_comparison_of_different_methods).
- [KFB09] Akihiro Kishimoto, Alex Fukunaga y Adi Botea. «Scalable, Parallel Best-First Search for Optimal Sequential Planning». En: *Proceedings of the International Conference on Automated Planning and Scheduling* 19.1 (2009), págs. 201-208. DOI: [10.1609/icaps.v19i1.13350](https://doi.org/10.1609/icaps.v19i1.13350). URL: <https://ojs.aaai.org/index.php/ICAPS/article/view/13350>.
- [MSV13] Carlos Mencia, Maria R. Sierra y Ramiro Varela. «Depth-first heuristic search for the job shop scheduling problem». En: *Annals of Operations Research* (jul. de 2013). URL: <https://link.springer.com/article/10.1007/s10479-012-1296-x#citeas>.
- [Kon14] Sai Varsha Konakalla. «A Star Algorithm». En: *Indiana State University* (dic. de 2014), pág. 4. URL: <http://cs.indstate.edu/~skonakalla/paper.pdf>.
- [WH16] Ariana Weinstock y Rachel Holladay. «Parallel A\* Graph Search». En: 2016. URL: <https://api.semanticscholar.org/CorpusID:218446573>.

- [Zag+17] Soha S. Zaghloul et al. «Parallelizing A\* Path Finding Algorithm». En: *International Journal Of Engineering And Computer Science* (sep. de 2017), 22469–22476. ISSN: 2319-7242. DOI: [0.18535/ijecs/v6i9.13](https://doi.org/10.18535/ijecs/v6i9.13). URL: <https://www.ijecs.in/index.php/ijecs/article/download/2774/2563/>.
- [ZJW20] Yuzhi Zhou, Xi Jin y Tianqi Wang. «FPGA Implementation of A\* Algorithm for Real-Time Path Planning». En: *International Journal of Reconfigurable Computing* (2020). DOI: [10.1155/2020/8896386](https://doi.org/10.1155/2020/8896386). URL: <https://doi.org/10.1155/2020/8896386>.
- [BC22] Cristina Ruiz de Bucesta Crespo. «Resolución del Job Shop Scheduling Problema mediante reglas de prioridad». En: *Universidad de Oviedo* (2022). URL: [https://digibuo.uniovi.es/dspace/bitstream/handle/10651/62015/TFG\\_CristinaRuizdeBucestaCrespo.pdf?sequence=10](https://digibuo.uniovi.es/dspace/bitstream/handle/10651/62015/TFG_CristinaRuizdeBucestaCrespo.pdf?sequence=10).

# Índice alfabético

	<b>A</b>	
Algoritmo A*		5
Costes		6
Coste F		6
Coste G		6
Coste H		6
Estado		6
Generación de sucesores		7
Listas de prioridad		7
Pseudocódigo		8
	<b>C</b>	
Conjuntos de datos		21
	<b>E</b>	
Equipo de Estudio		8
Arquitectura x86		8
FPGA		9
Equipos de Prueba		22
Arquitectura x86		22
CPU		22
OS		22
RAM		22
	<b>H</b>	
Hipótesis de Partida y Alcance		1
Alcance		3
Estado del arte		2
Objeto de la investigación		1
Requisitos		3
	<b>I</b>	
Implementación A*		11
State		11
Task		11
	<b>J</b>	
Job Shop Scheduling Problem		4
	<b>M</b>	
Makespan		25
	<b>O</b>	

Optimización A*	13
Monohilo	13
Heurísticos	14
State	13
Multihilo	16
Batch Solver	18
First Come First Serve (FCFS) Solver	16
Hash Distributed A* (HDA*) Solver	20
Recursive Solver	19

## P

Problema a investigar	4
-----------------------	---

## R

Resultados	26
Casos particulares	31
Estados solución intermedios	31
Varios estados iniciales	31
Comparativa de algoritmos	30
Complejidad del problema	26
Cuellos de botella	28
Heurísticos	27

## S

Speedup	25
---------	----

## T

Tamaño del problema	21
---------------------	----