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

3 de julio 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 desarrolladores suelen aprovechar este principio para acelerar y presentar resultados a sus usuarios más rápido. Sin embargo, no siempre se obtiene una reducción en el tiempo de ejecución, generalmente porque 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.

Existen situaciones donde la capacidad de cómputo de todos los hilos de una CPU no es suficiente. En estos casos se suele utilizar hardware específico como las tarjetas gráficas de propósito general (GPGPU) para incrementar la velocidad. Mayor capacidad aún que una CPU y una GPGPU es la que ofrecería un circuito integrado (SoC) diseñado ad-hoc (ASIC) para resolver el problema en particular. El principal problema de este acercamiento es el alto coste y riesgo de la producción de este tipo de herramientas. Una FPGA es un dispositivo capaz de simular un SoC con un riesgo mucho menor ya que la FPGA es reprogramable.

Algunos algoritmos son más dados 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

Índice de figuras

Índice de cuadros

Lista de Abreviaturas

ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
CPU	**C**entral **P**rocessing **U**nit
CSV	**C**omma **S**eparated **V**alue
FCFS	**F**irst **C**ome **F**irst **S**erve
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
GPGPU	**G**eneral **P**urpose **G**raphics **P**rocessing **U**nit
HDA*	**H**ash **D**istributed **A***
HDL	**H**ardware **D**escription **L**anguage
HLS	**H**igh **L**evel **S**ynthesis
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
JSP	**J**ob **S**hop **P**roblem
JSSP	**J**ob **S**hop **S**cheduling **P**roblem
OS	**O**perative **S**ystem
SoC	**S**ystem on **C**hip

Capítulo 1

Hipótesis de Partida y Alcance

1.1. Hipótesis de partida

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 de un interés personal por el diseño, implementación y optimización de algoritmos aplicables a problemas reales. Adicionalmente, interesa analizar la utilidad y límites de un dispositivo como las FPGAs, frecuentemente utilizado en entornos industriales donde los problemas de optimización son comunes pero suelen ser resueltos utilizando CPUs tradicionales en servidores.

El Job Shop Scheduling Problem es un problema de optimización sobre la planificación de horarios. Este es un problema np-hard mundialmente conocido por la comunidad científica, ha sido resuelto utilizando un gran abanico de algoritmo diferentes y está 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, un listado de trabajos y tareas a realizar. Puede ser aplicable en ámbitos industriales donde la automatización de la creación de planificaciones sea de interés.

1.2. 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

Objetivos y estado actual

Objetivos y estado del arte

2.1. Objetivos

Este proyecto tiene como objetivos principales el estudio y análisis del algoritmo A* enfocado a solucionar el Job Shop Scheduling Problem, la implementación y optimización del mismo junto con el desarrollo y comparación de distintas soluciones paralelas. Adicionalmente, se analiza el funcionamiento de una FPGA y las posibilidades que tiene este dispositivo de incrementar el rendimiento del algoritmo.

En detalle, este proyecto contiene:

- El Algoritmo A*: Funcionamiento monohilo del algortimo A* utilizado para resolver el JSP.
- Implementación de A*: Implementación monohilo del algoritmo A*.
- Optimización monohilo: Análisis y optimización de cuellos de botella.
- Paralelización: Análisis de distintas alternativas para paralelizar el algoritmo.
- FPGA: Funcionamiento y oportunidades de una FPGA.
- Heurísticos: Comparativa empírica de diferentes funciones heurísticas.
- Dijkstra: Comparativa empírica con el algoritmo de Dijkstra.
- Comparativa de algoritmos paralelos: Observaciones empíricas del rendimiento de los distintos algoritmos.

2.2. Estado del arte

Este proyecto abarca diversos tópicos, cuyas bibliografías (incluso tomadas de una en una) 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], [Pin12].

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 dado 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. La principal conclusión de la literatura en este área es que el uso del hardware incrementa el rendimiento del algoritmo y reduce su coste energético al mismo tiempo. En función de las herramientas utilizadas para implementar el algoritmo las ganancias son distintas, estudios en los que se utilizaron sintetizadores (HLS) para generar el código de la FPGA obtuvieron resultados menos interesantes que aquellos que diseñaron en hardware directamente. [ZJW20], [NSG17].

2.3. Job Shop Scheduling Problem

Se estudia la implementación de una solución al problema *Job Shop Scheduling (JSP)* [Yan77] [1] utilizando el algoritmo A*. Como su nombre indica, se trata de un problema en el que se debe crear una planificación. Como se especifica en la literatura, el JSP es un problema de optimización np-hard.

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 restricciones que definen el problema JSP:

0. Existen un número natural conocido de trabajos.
1. A excepción de la primera tarea de cada trabajo, todas tienen una única tarea predecesora del mismo trabajo que debe ser completada antes de iniciar su ejecución.
2. Cada tarea puede tener una duración distinta.
3. La duración de cada tarea es un número natural conocido.
4. Existe un número natural conocido de trabajadores.
5. Cada tarea tiene un trabajador asignado, de forma que sólo ese trabajador puede ejecutar la tarea.
6. Una vez iniciada una tarea, no se puede interrumpir su ejecución.
7. Un mismo trabajador puede intercalar la ejecución de tareas de diferentes trabajos.
8. Un trabajador sólo puede realizar una tarea al mismo tiempo.
9. Los tiempos de preparación de un trabajador antes de realizar una tarea son nulos.
10. Los tiempos de espera entre la realización de una tarea y otra son nulos.

FUNDAMENTAL

El Job Shop Scheduling Problem (JSP) es un problema np-hard que consiste en crear una planificación para ejecutar una serie de tareas que tienen una duración y están asignadas a un trabajador en particular.

[1] El problema también es conocido por otros nombres similares como *Job Shop Scheduling Problem (JSSP)* o Job Scheduling Problem (JSP).

2.3.1. Notación

Con el objetivo de simplicar los ejemplos contenidos en este documento se ha diseñado una notación capaz de describir trabajos, tareas y estados. A continuación se encuentra una explicación de dicha notación necesaria para comprender correctamente los ejemplos.

2.3.1.1. Tarea

Una tarea está compuesta por dos datos: La duración y el trabajador que la debe ejecutar: (A, B) donde A es la duración y B el ID del trabajador. $(5, 0)$ es una tarea que dura 5 instantes de tiempo y debe ser ejecutada por el trabajador 0.

2.3.1.2. Trabajo

Un trabajo está compuesto por un listado de tareas. $(A, B), (C, D)$ es un trabajo compuesto por dos tareas: (A, B) es una tarea que dura A instantes de tiempo y debe ser ejecutada por B y (C, D) es una tarea que dura C instantes de tiempo y debe ser ejecutada por D. Así mismo, la duración de la tarea 2 del trabajo 0 sería $D_{0,2}$ A lo largo del documento se utilizará la letra k como el índice de la primera tarea sin planificar.

Finalmente, el número total de trabajos de un conjunto de datos se denomina J y el número total de tareas de un trabajo T.

2.3.1.3. Conjunto de datos

Un conjunto de datos está compuesto por un listado de trabajos:

0. (A, B), (C, D)
1. (E, F), (G, H)

El conjunto está compuesto por dos trabajos (IDs 0 y 1) cada uno con dos tareas. Para referirse a la tarea (A, B) se utilizaría T00 (Trabajo 0, tarea 0) mientras que para la tarea (G, H) se utilizaría T11 (Trabajo 1, tarea 1). Adicionalmente, se podrá añadir la duración de la tarea a esta notación: T00(A) corresponde al trabajo 0, tarea 0 con duración A instantes de tiempo.

2.3.1.4. Planificación

La planificación es un dato propio del estado. Indica el instante de tiempo en el que se inicia cada tarea: $(0, 3, -1)$ indica que la tarea 0 se inicia en el instante 0, la 1 en el 3 y la 2 aún está sin planificar.

Generalmente se muestra la planificación de todo el conjunto de datos utilizando un listado:

0. $(0, 3, -1)$
1. $(0, -1, -1)$

La T00 se inicia en el instante 0 mientras que la T11 está aún sin planificar.

2.3.1.5. Estado trabajadores

Propio también del estado es la información sobre el instante de tiempo en el que cada trabajador quedará libre. Estos datos se muestran en forma de lista donde el elemento *K* corresponde al instante de tiempo donde el trabajador con ID *K* queda libre. $(2, 8, 0)$ indica que el trabajador 0 esta libre a partir del instante 2, el 1 a partir del 8 y el 2 a partir del 0.

2.3.2. Ejemplo

A continuación se muestra un ejemplo de menor tamaño resuelto manualmente donde cada línea corresponde a un trabajo y cada tupla al par (duración, trabajador).

0. (2, 0), (5, 1), (1, 2)
1. (3, 1), (3, 2), (3, 0)

Se crea un diagrama Pert (figura 2.1) donde cada arco es una tarea, ignorando a los trabajadores, atendiendo únicamente a la duración de cada tarea. Los arcos están etiquetados con la tarea a ejecutar y su duración. Los nodos contienen dos números: el primero corresponde al primer instante de tiempo en el que la tarea se puede ejecutar y el segundo al último (tiempos *early* y *late*).

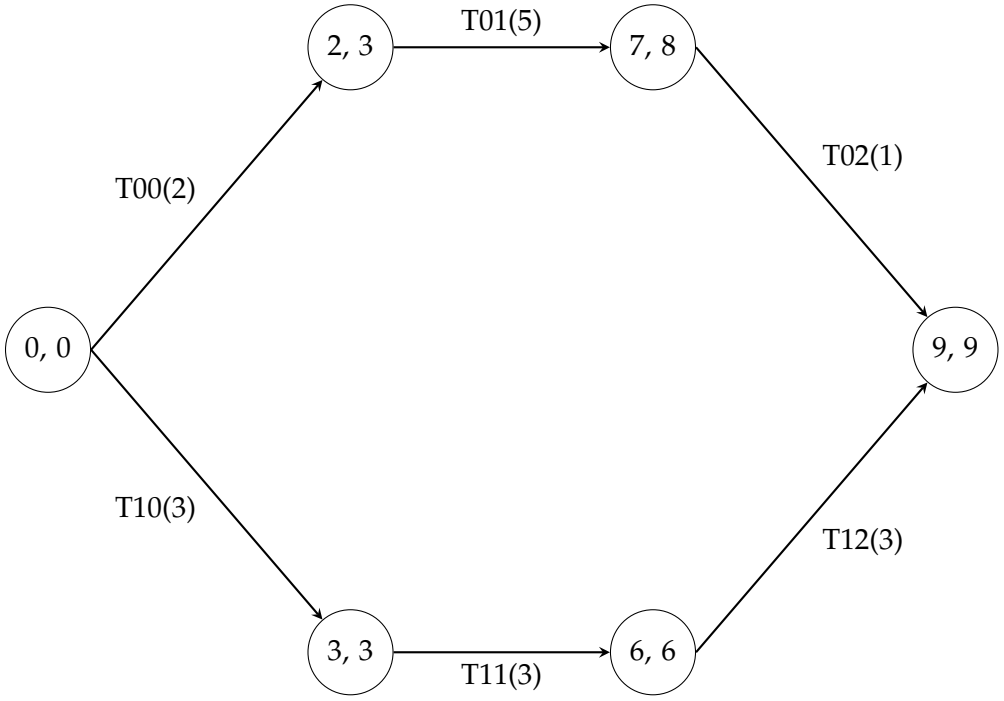


FIGURA 2.1: Diagrama Pert ejemplo.

El diagrama siguiente diagrama Gantt (2.2) tiene en cuenta la disponibilidad de cada trabajador, nótese que la tarea T01 comienza en el instante 3 en lugar del 2, esto se debe a que el trabajador no está disponible hasta ese instante. Los nodos del diagrama muestran el instante de inicio de la tarea, la tarea y su duración y el instante de fin. Cada trabajador tiene asignado un color distinto, por lo que no es posible que dos tareas del mismo color se encuentren planificadas en el mismo instante de tiempo.

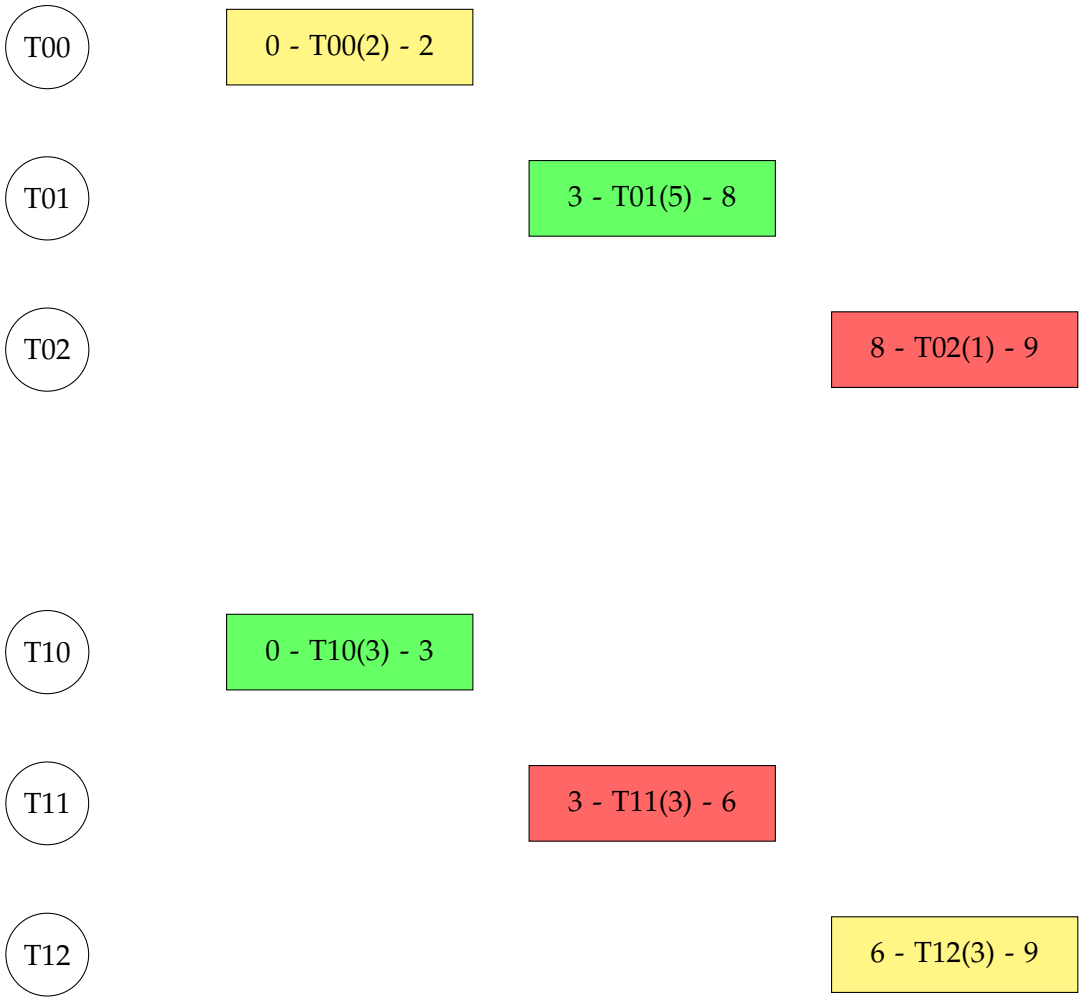


FIGURA 2.2: Diagrama Gantt ejemplo.

Capítulo 3

Metodología de trabajo

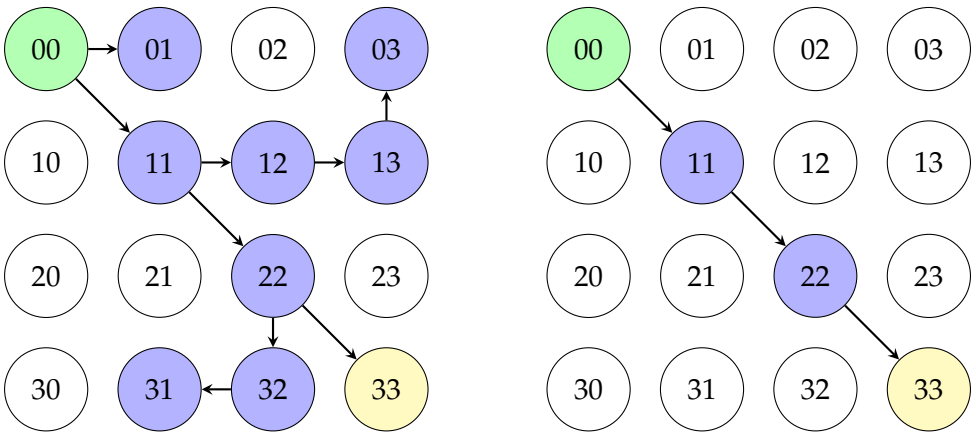
Método de resolución y comparativas

3.1. Método de resolución

3.1.1. El Algoritmo A*

Existen numerosos algoritmos capaces de resolver el problema del JSP. 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 3.1).



(A) Algoritmo Dijkstra (B) Algoritmo A*

FIGURA 3.1: Comparativa entre algoritmos Dijkstra y A*

FUNDAMENTAL

El algoritmo A* sirve para realizar búsquedas en grafos, el problema JSP consiste en una búsqueda en un grafo por lo que se puede utilizar el A* para resolverlo.

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

3.1.1.1. Pseudocódigo

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

LISTING 3.1: Pseudocódigo del algoritmo A*

El algoritmo (véase pseudocódigo 3.1) utiliza una lista de estados a explorar inicialmente compuesta por el estado inicial (ln1-2). Además, se crean estructuras de datos diseñadas para almacenar el mejor coste conocido para llegar a un estado cualquiera (ln4-5). De esta forma, `g_costes[estado_a]` es el mejor coste G conocido hasta el momento para el `estado_a`.

El resto del algoritmo se encarga de explorar los nodos de la lista hasta que esté vacía o el estado actual sea el objetivo. Si el estado actual es el objetivo, el algoritmo retorna (ln13). En otro caso, calcula los sucesores del estado actual y los procesa individualmente (ln16). Si el coste G del sucesor es mejor que el mejor coste G conocido para llegar al estado sucesor se graban sus costes y se añade a la lista si no estaba ya en ella (ln19-24).

3.1.1.2. Componentes A*

3.1.1.2.1. 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 representa una planificación parcial o completa en la cual una serie de tareas han sido planificadas:

- 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).

3.1.1.2.2. Costes

3.1.1.2.2.1. 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.

3.1.1.2.2.2. 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.

Idealmente esta función heurística será una cota inferior cuando el problema sea de minimización y una cota superior cuando sea de maximización. El heurístico deberá ser entonces diseñado atendiendo al tipo de problema.

FUNDAMENTAL

El coste H es una estimación de la respuesta a la pregunta: '¿Cuánto queda hasta el nodo objetivo?'

3.1.1.2.2.3. 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,

$$cost_f = cost_g + cost_h$$

3.1.1.2.3. Generación de sucesores

Un estado con N trabajos por completar ($J_0 \dots J_N$) tiene N tareas ($T_0 \dots T_N$) que están listas para ser ejecutadas (sus predecesoras han sido ya completadas). Este estado generará entonces N estados sucesores donde en cada uno se planificará una de las tareas para el primer instante de tiempo posible. Este primer instante se calcula utilizando el máximo entre el instante de fin de la tarea predecesora y el instante en el que el trabajador asignado para la tarea a planificar T_x queda libre.

EJEMPLO

En el ejemplo anterior (2.3.2) con trabajos:

0. (2, 0), (5, 1), (1, 2)
1. (3, 1), (3, 2), (3, 0)

El estado:

- Planificación:
  0. (0, 3, -1)
  1. (0, -1, -1)
- Estado trabajadores: (2, 8, 0)

Tiene dos trabajos sin completar, por lo que generará dos sucesores. Uno en el que se planifica la primera tarea sin planificar del trabajo 0.

- Instante en el que se puede iniciar la tarea: $3 + 5 = 8$
- Instante en el que el trabajador está libre: 0

Por lo tanto, la tarea comenzará en $max(0, 8) = 8$. Y el trabajador estará libre en $8 + 1 = 9$.

- Planificación:
  0. (0, 3, 8)
  1. (0, -1, -1)
- Estado trabajadores: (2, 8, 9)

Y otro en el que se planifica la primera tarea sin planificar del trabajo 1.

- Instante en el que se puede iniciar la tarea: $0 + 3 = 3$
- Instante en el que el trabajador está libre: 0

Por lo tanto, la tarea comenzará en $max(0, 3) = 3$. Y el trabajador estará libre en $3 + 3 = 6$.

- Planificación:
  0. (0, 3, -1)
  1. (0, 3, -1)
- Estado trabajadores: (2, 8, 6)

3.1.1.2.4. 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 explorar.

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 $cost_f$.

Nótese que el coste temporal de insertar un elemento en una posición intermedia de una lista suele tener un coste lineal en relación con el tamaño de la lista (es necesario desplazar el resto de elementos una posición hacia la derecha). Este coste puede ser mitigado utilizando estructuras de datos específicas como las listas doblemente enlazadas que faciliten este tipo de inserción. De cualquier forma, incluso con este tipo de listas es necesario iterar elemento a elemento utilizando un comparador para localizar la posición indicada para el elemento a insertar.

FUNDAMENTAL

El coste temporal de utilizar la lista de prioridad incrementa con el número de elementos que haya en la misma. Por ello es de interés que se inserten el menor número de elementos posibles.
Estas operaciones en la lista de prioridad serán posteriormente el principal cuello de botella del algoritmo.

En todos los casos, el número de nodos explorados (insertados en la lista) es directamente proporcional con el tiempo de ejecución.

3.1.2. Equipo de Estudio

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

3.1.2.1. 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.

3.1.2.2. 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 HLS, Vitis (Vitis Unified) y Vivado. Este programa ofrece entre muchas otras herramientas un sintetizador capaz de transpilar código C++ a VHDL o Verilog que puede ser entonces compilado para ejecutarse en la FPGA.

3.2. Método de comparativas

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

0. Tiempo de ejecución.
1. 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.

Capítulo 4

Trabajo y Resultados

Implementación, Optimización, Experimentos y Resultados

4.1. 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.

NOTA

El principal contenido de esta investigación es el estudio de distintas implementaciones paralelas del algoritmo A*, por ello es necesario tener alguna noción sobre la Implementación del algoritmo.

4.1.1. Task

La clase `Task` correspone 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.

4.1.2. 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()`).

4.2. Optimización

4.2.1. Algoritmo A* monohilo

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.

4.2.1.1. 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==`.

4.2.1.2. 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.

FUNDAMENTAL

Las funciones heurísticas son una estimación del tiempo restante hasta completar todas las tareas. Esta estimación puede ser una cota inferior o superior. Las cotas inferiores retornarán soluciones óptimas a costa de explorar más nodos. Las cotas superiores retornarán soluciones de peor calidad a coste de explorar menos nodos.

4.2.1.2.1. 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.

$$cost_h = max_{0 \leq j \leq J} \left(\sum_{t=k}^{T} D_{j,t} \right)$$

EJEMPLO

El conjunto de datos:

1. (2, 0), (4, 1), (5, 2)
2. (5, 3), (2, 4), (1, 5)

La planificación:

1. (0, 2, -1)
2. (0, -1, -1)

Se calcula la suma de la duración de las tareas no planificadas de cada trabajo:

1. (0, 2, -1): 5
2. (0, -1, -1): 3

Se obtiene el máximo: $cost_h = max(5, 3) = 5$

4.2.1.2.2. Heurístico para tiempo

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

$$cost_h = \sum_{j=0}^{J} \sum_{t=k}^{T} D_{j,t}$$

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

EJEMPLO

El conjunto de datos:

1. (2, 0), (4, 1), (5, 2)
2. (5, 3), (2, 4), (1, 5)

La planificación:

1. (0, 2, -1)
2. (0, -1, -1)

Se suman todas las duraciones de las tareas no planificadas:

$$cost_h = \sum_{j=0}^{J} \sum_{t=k}^{T} D_{j,t} = 5 + 2 + 1 = 8$$

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.

4.2.2. 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.

El algoritmo A* monohilo retorna la solución óptima para el problema. Esta característica no se mantiene para todas las implementaciones paralelas. Esto sucederá cuando no exista sincronización entre los distintos hilos, o lo que es lo mismo, cuando existan condiciones de carrera que puedan alterar el curso del algoritmo.

EJEMPLO

Una estrategia en la que los hilos procesen nodos a medida que se insertan en el `open_set` tendrá condiciones de carrera. La solución dependerá de qué hilo procese qué estado primero.

4.2.2.1. 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 (Véase Figura 4.1).

FUNDAMENTAL

La estrategia FCFS permite que los hilos procesen los nodos a medida que entran en el `open_set` .

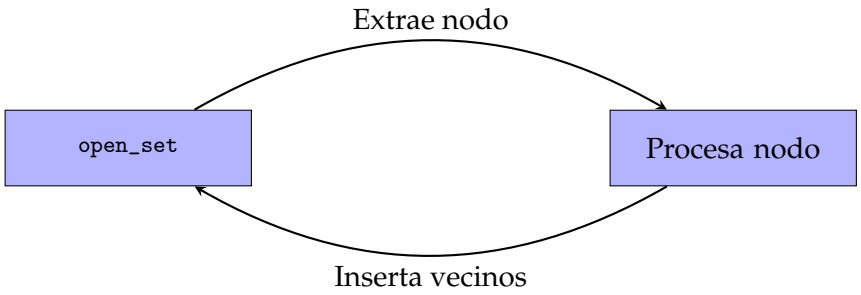
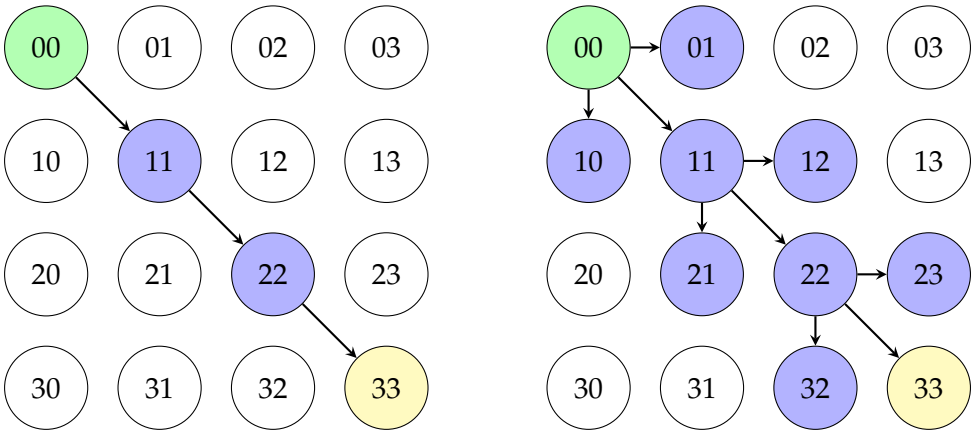


FIGURA 4.1: Representación de la estrategia FCFS

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 4.2)



(A) Algoritmo A* monohilo (B) Algoritmo A* multihilo (3 hilos)

FIGURA 4.2: 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.

4.2.2.1.1. 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.

4.2.2.1.1.1. 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 simultaneo 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.

4.2.2.1.1.2. `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 ser haría en $O(n)$ [1].

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.

4.2.2.2. 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 4.3.

FUNDAMENTAL

La estrategia Batch Solver explora el máximo número posible de nodos de forma simultánea, haciendo que los hilos esperen al resto cuando finalizan de explorar su nodo.
El máximo número de nodos está limitado por el mínimo entre el número de nodos del `open_set` y el número de hilos.

4.2.2.2.1. Secciones críticas

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

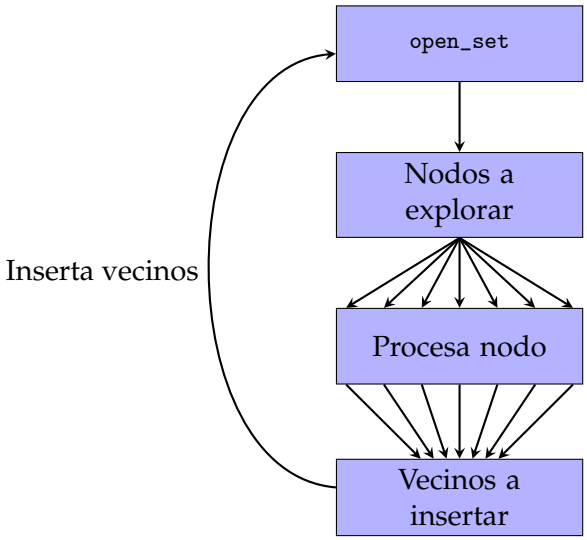


FIGURA 4.3: Representación de la estrategia Batch

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`.

4.2.2.3. Recursive Solver

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

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

FUNDAMENTAL

El Recursive Solver es equivalente a ejecutar el algoritmo A* N veces de de forma individual siendo N el número de vecinos del nodo inicial.

4.2.2.4. 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 consecuente, 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 [2] 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.

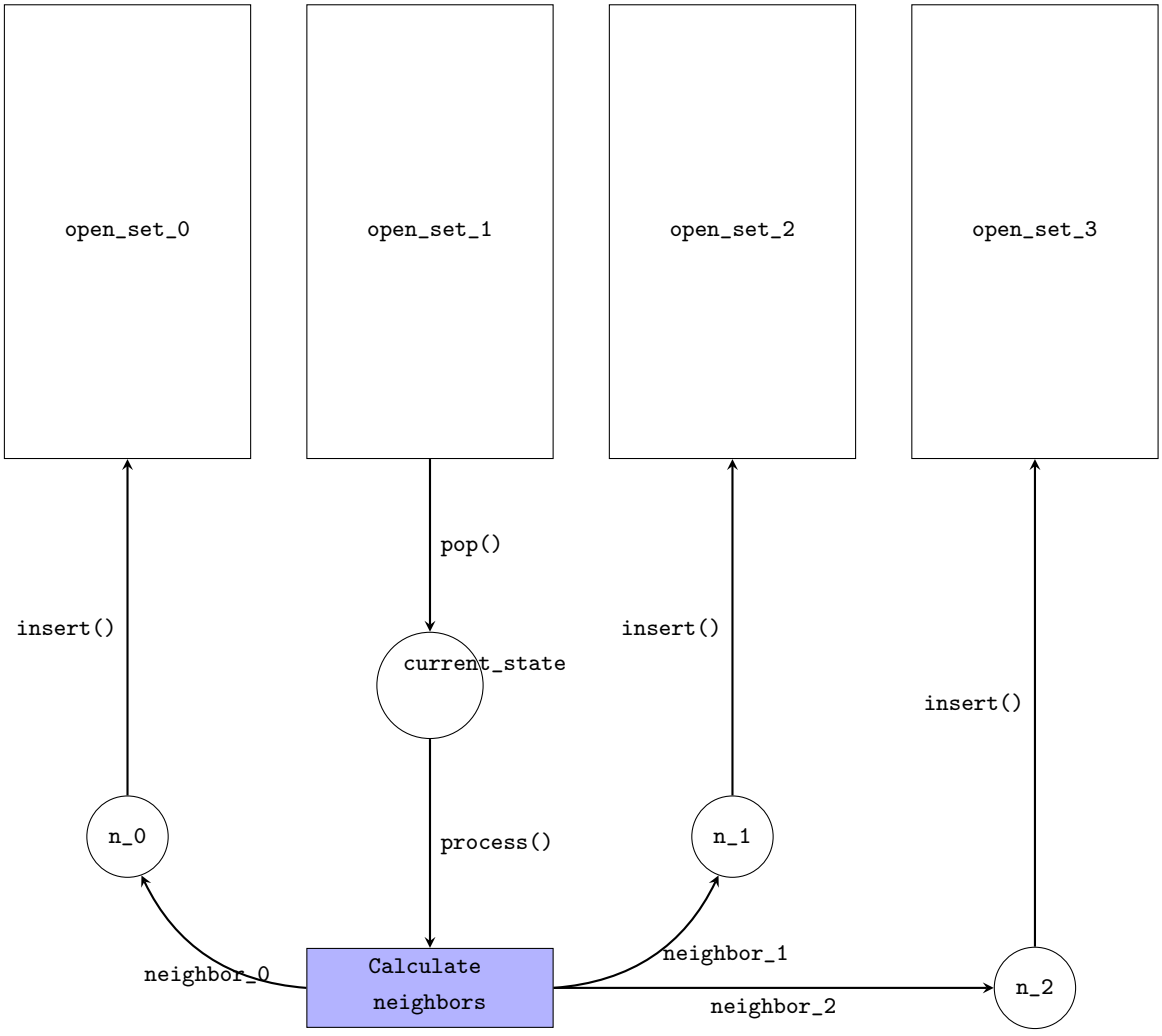


FIGURA 4.4: Representación de la estrategia HDA*

[2] Una función hash es uniforme si los valores que retorna tienen la misma probabilidad de ser retornados.

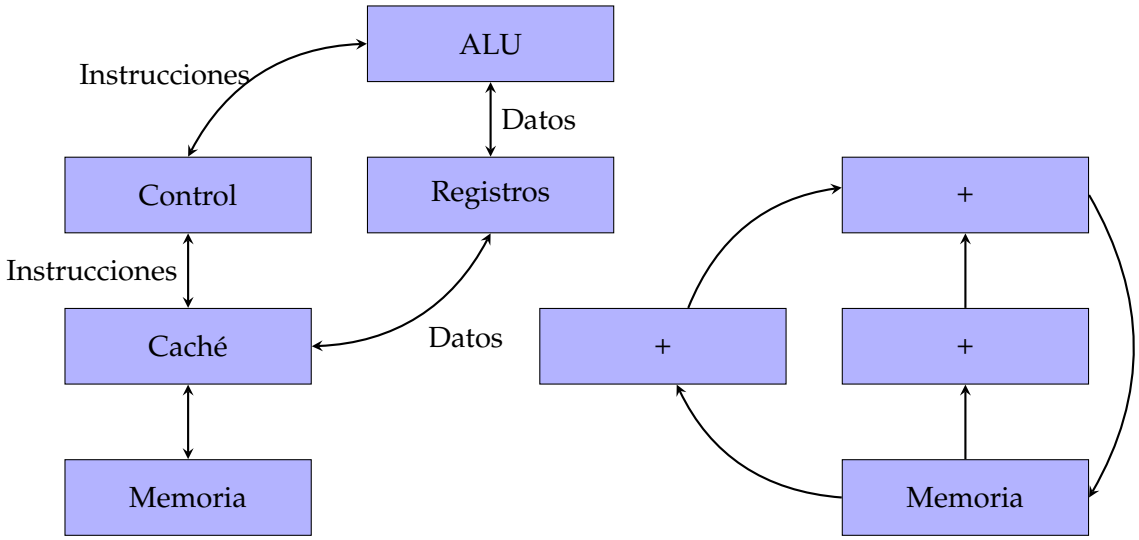
FUNDAMENTAL

El Hash Distributed A* Solver es la única estrategia que utiliza un `open_set` para cada hilo, reduciendo el tamaño de cada uno y aliviando el principal cuello de botella del algoritmo.

4.3. FPGA

Como se indicó en el resumen, una FPGA es un dispositivo capaz de emular el comportamiento de un circuito integrado diseñado para resolver un problema en particular (ASIC). El principal beneficio de una FPGA es que es reprogramable mientras que un circuito impreso no se puede modificar una vez producido.

El funcionamiento de una FPGA no se debe confundir con el de un procesador convencional. Las CPUs comúnmente encontradas en ordenadores utilizan la arquitectura de Von Neumann (Véase Figura 4.5a) para realizar calculos. Por otro lado, las FPGAs no utilizan esta arquitectura. La principal diferencia se encuentra en la carga de instrucciones: Una CPU tiene las instrucciones almacenadas en la memoria RAM y se van cargando en la caché para poder ser leídas por la ALU a medida se van necesitando. Una FPGA no tiene las instrucciones almacenadas en ningún sitio, las 'instrucciones' están formadas principalmente por puertas lógicas, registros y *Look-up Tables (LUTs)* que están 'impresas' en el circuito [3]. Por la FPGA sólo 'viajan' datos, nunca instrucciones.



(A) Arquitectura Von Neumann (B) Arquitectura FPGA

Entonces, los lenguajes en los que se suelen programar las FPGAs (VHDL, Verilog [4]) permiten el uso de puertas lógicas, operaciones matemáticas y movimiento

[3] A lo largo del documento, se hablará de la FPGA como si fuese un circuito impreso reprogramable. En realidad el dispositivo no tiene impresa ninguna puerta lógica, pero esta forma de verlo facilita mucho el entendimiento del hardware.

[4] De ahora en adelante, se mencionará sólo VHDL, nótese que VHDL y Verilog son lenguajes que sirven para lo mismo, son intercambiables.

de datos entre registros. Este acercamiento es de muy bajo nivel, no admite programación orientada a objetos ni facilita la abstracción. Un programa escrito en estos lenguajes genera un *bitstream* que es la información escrita posteriormente en la FPGA. Finalmente, las funciones del *bitstream* que están escritas en la FPGA pueden ser llamadas desde C/C++.

Alternativamente, se puede escribir código en C/C++ y utilizar una herramienta denominada *High Level Synthesizer (HLS)* (Sintetizador de Alto Nivel) que transpila código C/C++ a VHDL. Una vez generado el código VHDL se puede proceder por los cauces correspondientes como si se hubiese escrito el código a mano en VHDL.

Como es de esperar, una solución escrita directamente en VHDL tiene la oportunidad de estar mucho más optimizada que una solución transpilada utilizando HLS. No obstante, en la mayoría de casos el código generado por HLS suele ser lo suficientemente adecuado para cumplir las necesidades del programa. En algunos casos de la literatura de este proyecto se ha utilizado VHDL ([ZJW20]) y en otros se ha optado por C/C++ y HLS ([NSG17]).

Las operaciones en una FPGA siguen el mismo esquema: Un dato en un registro se usa como operando en una operación y el resultado se almacena en otro registro. El valor resultante puede ser leído como salida o puede utilizarse como factor de otra operación. Al conjunto de operaciones se le denomina *pipeline*, la duración del mismo se mide en ciclos (e.g. número de puertas lógicas) y la frecuencia con la que se pueden introducir nuevos datos de entrada se denomina intervalo de iniciación. Como los valores intermedios se almacenan en registros es posible ejecutar la misma operación para varios datos de forma casi paralela, no es necesario que el dato de entrada permanezca en la entrada hasta finalizar la operación.

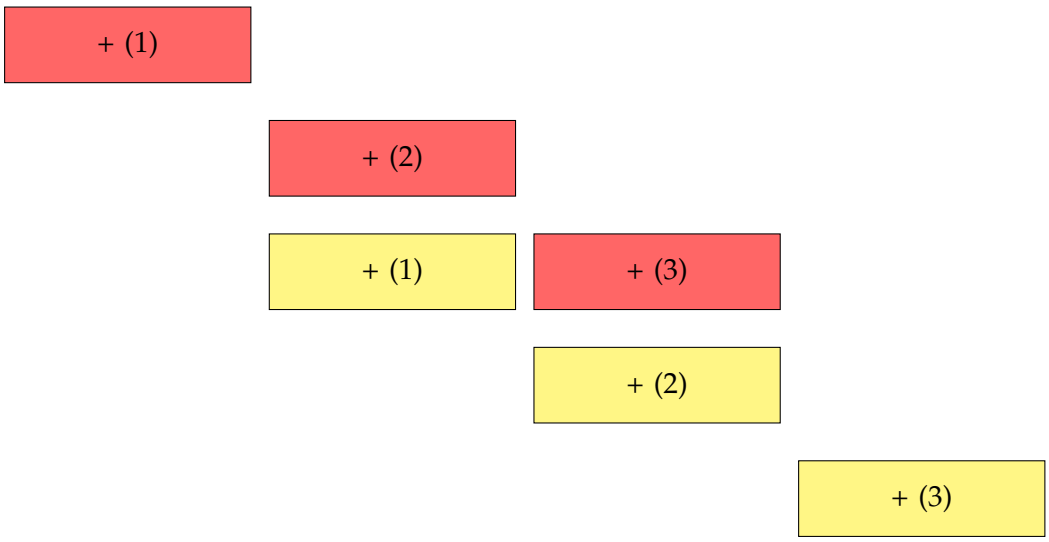


FIGURA 4.6: Dos operaciones ejecutadas en el mismo *pipeline*.

La figura 4.6 muestra una simulación de un *pipeline* que realiza tres sumas a los datos de entrada. La duración es de tres ciclos, pero es posible ejecutar dos veces la operación utilizando tan sólo cuatro ciclos porque el intervalo es sólo de uno. Adicionalmente, es posible replicar la impresión de las puertas lógicas en distintas áreas de la FPGA, paralelizando el cálculo e incrementando la productividad.

Desde un punto de vista de bajo nivel, el principal desafío de programar una FPGA se encuentra en minimizar el intervalo. HLS siempre retornará la implementación con el mínimo intervalo posible a menos que se le indique lo contrario. No obstante, HLS no modificará el código ni la secuencia de operaciones.

El hecho de que HLS no modifique el código ni la secuencia de operaciones es de vital importancia para el desarrollo. Esto implica que el programa se debe de escribir de forma que la implementación resultante de HLS minimice el intervalo.

EJEMPLO

El siguiente algoritmo en C:

```
1  for (unsigned int i = 1 ; i < N - 1 ; i++)
2  {
3      out[i] = in[i-1] + in[i] + in[i+1];
4  }
5
```

Sería un error transpilar este código en HLS porque en lugar de leer un valor y ejecutar una operación se estan leyendo tres valores y ejecutando una operación. Esto resultaría en un intervalo de tres en lugar de uno, dividiendo en tres la productividad del algoritmo en FPGA.
En su lugar:

```
1  int pprevious = in[0];
2  int previous = in[1];
3  for (unsigned int i = 1 ; i < N - 1 ; i++)
4  {
5      const int current = in[i + 1];
6      out[i] = pprevious + previous + current;
7      pprevious = previous;
8      previous = current;
9  }
10
```

Esta implementación sólo lee un valor en cada iteración y sustituye los que ya tiene en los registros convenientes. Al transpilar esta solución, se obtendría un algoritmo con un intervalo de un ciclo.

Como se puede ver, una FPGA puede ser de gran utilidad para implementar algunas funciones de un programa que posteriormente sean llamadas desde un programa principal. De esta forma, es posible centrar la atención de los desarrolladores en optimizar las funciones utilizadas en la FPGA para reducir sus intervalos. También es posible transpilar un programa completo a VHDL para que sea ejecutado en una FPGA, que sea más rentable una opción o la otra es una cuestión dependiente en gran medida del programa a implementar y del uso que se tenga pensado darle. Es importante tener en cuenta que generalmente una FPGA cuenta con una cantidad limitada de memoria, y que si bien existen FPGAs con mayor capacidad, éstas suelen tener un coste notablemente más elevado.

4.4. 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 40%, 50%, 60% del conjunto 'abz5' (4x4x10, 5x5x10, 6x6x10), o el 100% de 'ft06' (6x6x6). El algoritmo utilizado para importar y cortar los conjuntos de datos se puede encontar en el Anexo A.3.

NOTA

Un conjunto de datos tiene un tamaño de AxBxC cuando está compuesto por A trabajos, B tareas en cada uno y C trabajadores.

4.5. Equipos de Prueba

4.5.1. Arquitectura x86

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

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 4.1: Equipos de prueba, arquitectura x86: OS

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 4.2: Equipos de prueba, arquitectura x86: CPU

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

CUADRO 4.3: Equipos de prueba, arquitectura x86: RAM

4.6. 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[5]. 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.

4.7. 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 [6].

[5] Véase A.1

[6] 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.

4.7.1. *Speedup*

Sean T_1 y T_0 dos métricas del tiempo de ejecución de dos algoritmos distintos ($A0$ y $A1$), el *speedup* del algoritmo $A0$ respecto al $A1$ ($S_{A0,A1}$) es:

$$S_{A0,A1} = T_1/T_0$$

EJEMPLO

Si el algoritmo $A0$ tardó $5s$ en finalizar y $A1$ tardó $10s$, el *speedup* ($S_{A0,A1}$):

$$S_{A0,A1} = T_1/T_0 = 10/5 = 2$$

$A0$ es el doble de rápido que $A1$.

Por lo tanto:

$$S_{A0,A1} = 1 \rightarrow T_1 = T_0$$
$$S_{A0,A1} > 1 \rightarrow T_1 > T_0$$
$$S_{A0,A1} < 1 \rightarrow T_1 < T_0$$

FUNDAMENTAL

Un *speedup* inferior a 1 implica una reducción en el rendimiento.
Un *speedup* superior a 1 implica un incremento en el rendimiento.

4.7.2. *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, el *makespan* óptimo es el mínimo. El algoritmo desarrollado en este proyecto no tiene la certeza de obtener el resultado óptimo. Por ello, si una implementación A retorna resultados con un *makespan* que otra B, A es mejor que B [7].

EJEMPLO

Sea la siguiente planificación:

- Planificación:
  1. (0, 3, 5)
  2. (0, 3, 3)
- Estado trabajadores: (2, 8, 3)

El *makespan* es $max([2, 8, 3]) = 8$.

[7] Suponiendo que el resto de métricas entre A y B son lo suficientemente similares.

4.8. Resultados y Análisis

4.8.1. Complejidad del problema

NOTA

Como se ha visto, el algoritmo A* es bastante modular, existen diversas partes intercambiables que resulta en un gran número de posibilidades a probar. Las ejecuciones realizadas a continuación se basan en probar distintas combinaciones que resultan de interés. Para simplificar la lectura de los resultados mostrados en las gráficas, a continuación se muestra un resumen de las variables utilizadas en las mismas:

- Número de hilos: 1T, 4T, 8T...
- Solucionador: 'FCFS Solver', 'HDA* Solver', 'Recursive Solver'...
- Heurísticos: 'Fast', 'Slow'...[a]
- Problema: 60%abz5.csv (6x6x10), 40%abz5.csv (4x4x10)...

Si todas las mediciones de la gráfica coinciden en una de las variables, ésta se anotará en el subtítulo de la gráfica.

EJEMPLO

- '4 FCFS Solver Fast': FCFS con 4 hilos y el heurístico rápido.
- '2 HDA* Solver': FCFS con 2 hilos y el heurístico lento (óptimo).

[a] La falta de una etiqueta para el heurístico implica el óptimo (también llamado slow)

En la siguiente gráfica (figura 4.7), se puede ver el tiempo de ejecución necesario para completar el problema utilizando uno de los algoritmos.

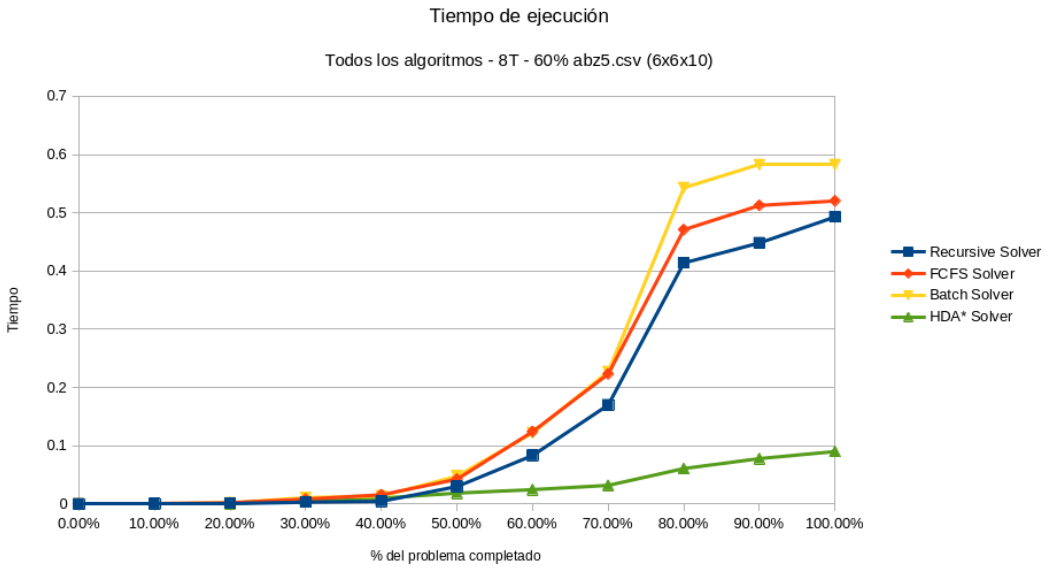


FIGURA 4.7: Tiempo de ejecución de un único problema.

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 4.8).

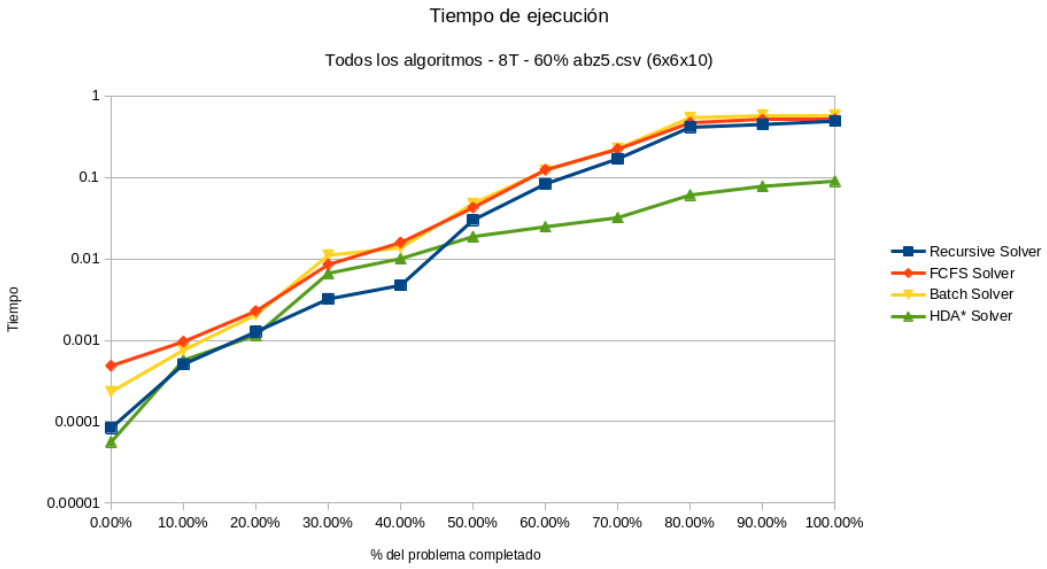


FIGURA 4.8: Tiempo de ejecución de un único problema (escala logarítmica).

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

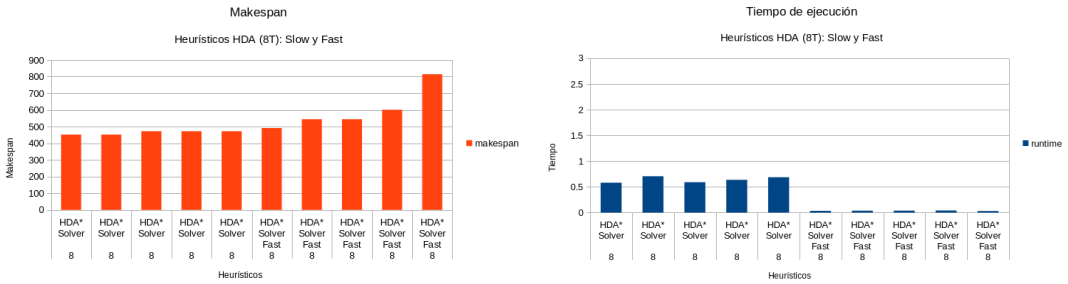
NOTA

Este capítulo 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.

4.8.2. Comparativa de heurísticos

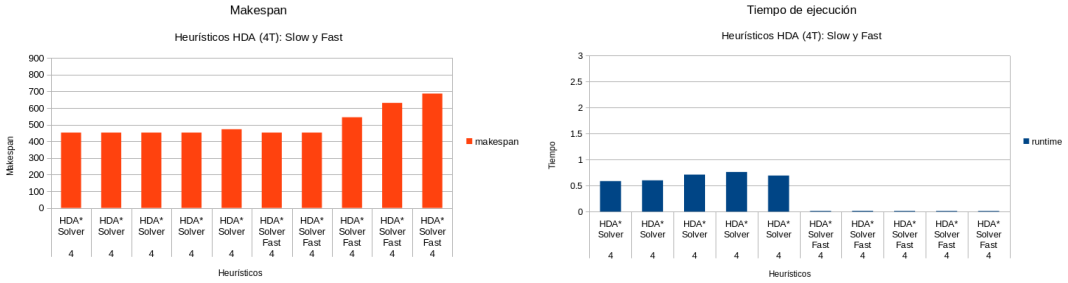
Como ya se discutió en la sección sobre optimización (4.2.1.2), existen varias implementaciones de la función heurística que da al algoritmo A* su particular comportamiento de ir 'dirigido' hacia la solución. En esta investigación se han implementado dos heurísticos: uno de ellos busca una solución que 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.



(A) *Makespan* con 8 hilos. (B) Tiempo de ejecución con 8 hilos.

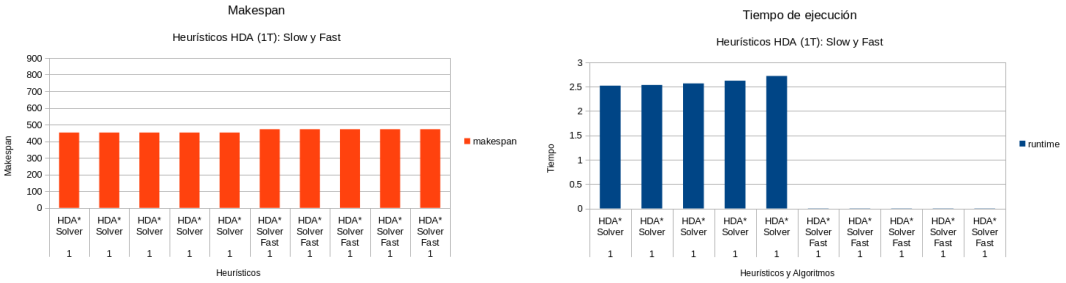
FIGURA 4.9: Métricas de heurísticos con 8 hilos.



(A) *Makespan* con 4 hilos. (B) Tiempo de ejecución con 4 hilos.

FIGURA 4.10: Métricas de heurísticos con 4 hilos.

Analizandolas 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 4.9, 4.10 y 4.11).



(A) *Makespan* con 1 hilo. (B) Tiempo de ejecución con 1 hilo.

FIGURA 4.11: Métricas de heurísticos con 1 hilo.

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 (70x10x10) y se ha obtenido el *speedup*.

Además, se puede observar una clara tendencia en los resultados que utilizan el heurístico rápido. A menor número de hilos, mejor *makespan* retornan.

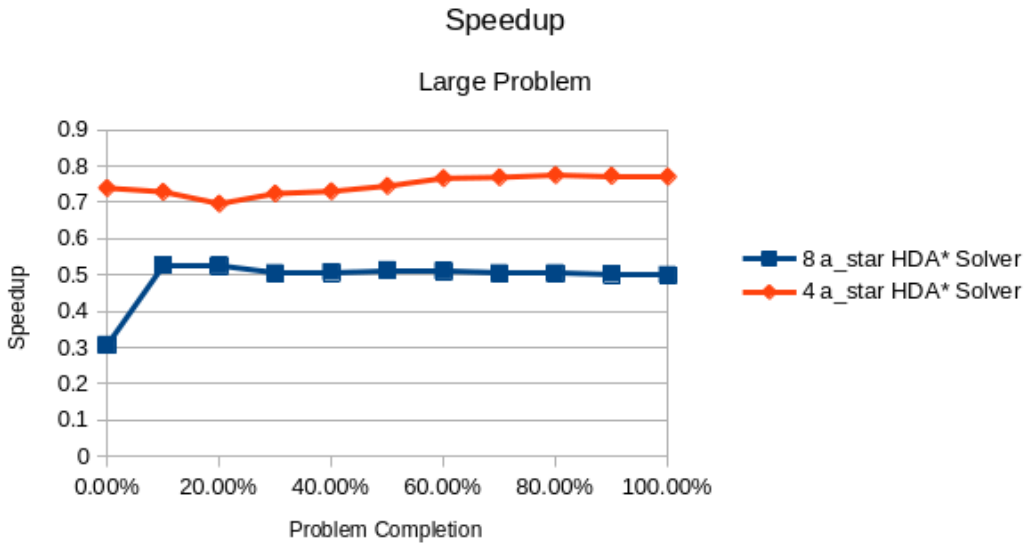


FIGURA 4.12: *Speedup* en un problema de gran tamaño.

La gráfica (4.12) 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 incrementado 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.

Se deja también constancia de que las pruebas ejecutadas indican que el error cometido por el heurístico rápido incrementa proporcionalmente con el tamaño del problema. En las pruebas que utilizaron conjuntos de datos de 6x6x10, el error se

encontraba entre 20 y 50, mientras que en conjuntos de datos de 70x10x10, el error se podría encontrar en los miles. Por ello, el heurístico rápido se podría considerar únicamente para resolver problemas de menor tamaño e incluso en esos casos se debería tener en cuenta su falta de precisión.

4.8.2.1. Comparativa con Random Solver

El heurístico rápido obtiene resultados cuya calidad es razonable únicamente cuando se tiene en cuenta la velocidad con la que los obtiene. Se ha implementado un algoritmo (no A*) que resuelve el problema del JSP de forma aleatoria (Anexo A.2), esto es, selecciona una tarea cualquiera y la introduce en el plan. Como el nombre del *solver* indica, la planificación que retorna este algoritmo es aleatoria. Utilizamos estos resultados para criticar el *makespan* del heurístico rápido.

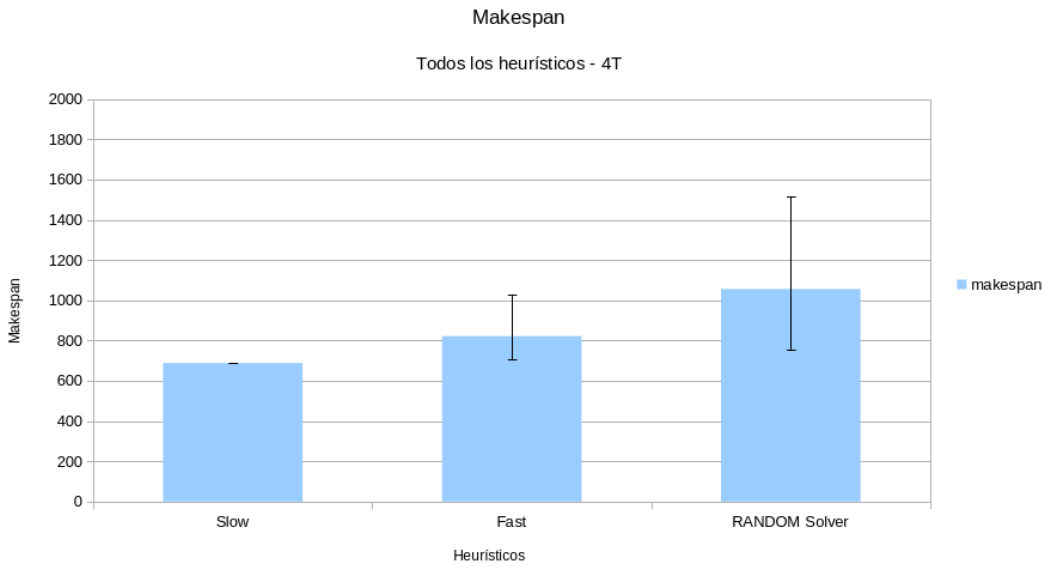


FIGURA 4.13: *Makespan* de diferentes heurísticos

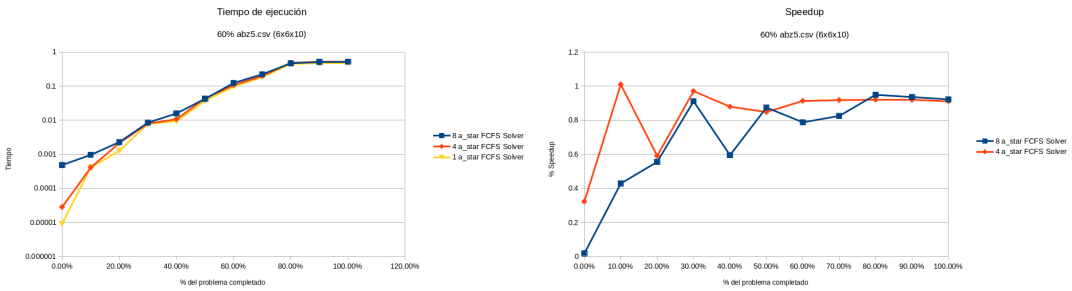
La figura 4.13 muestra el *makespan* del heurístico óptimo (el lento), el rápido y el obtenido por el Random Solver. Adicionalmente, se muestran también los *makespan* máximo y mínimo obtenidos por cada uno de ellos.

Como se puede ver en el diagrama, el heurístico rápido obtiene mejores resultados que el Random Solver y más importantemente tiene un margen de error mucho menor. Entonces, aunque los resultados del heurístico rápido no sean óptimos, son mejores que crear una planificación aleatoria.

4.8.3. 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 4.14b).



(A) Tiempo de ejecución de FCFS. (B) *Speedup* de FCFS.

FIGURA 4.14: 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 4.15).

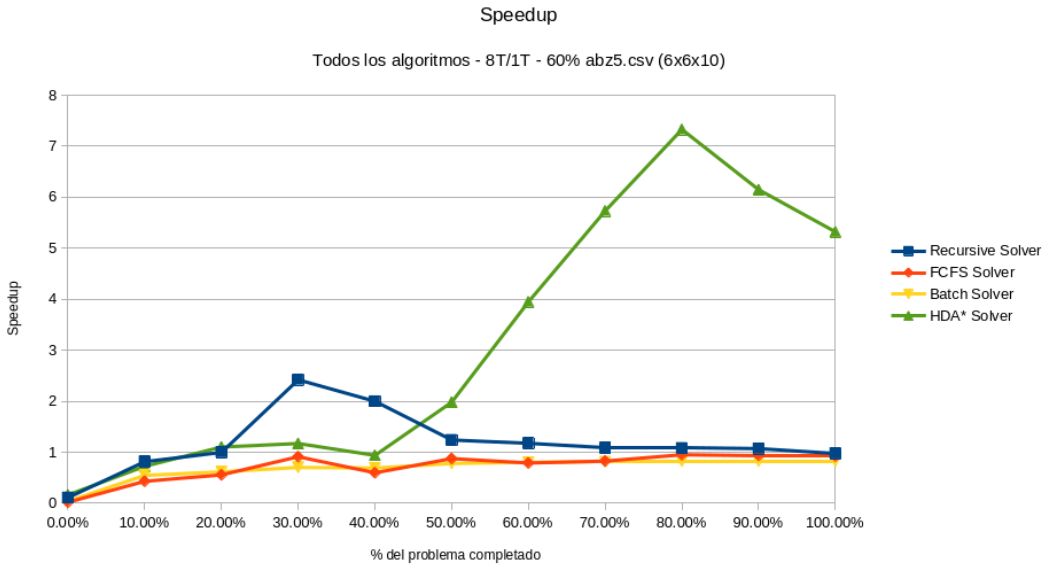


FIGURA 4.15: *Speedup* de HDA*.

Nótese que al inicio del problema (aproximadamente hasta completar el 20%) 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.

4.8.4. 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.

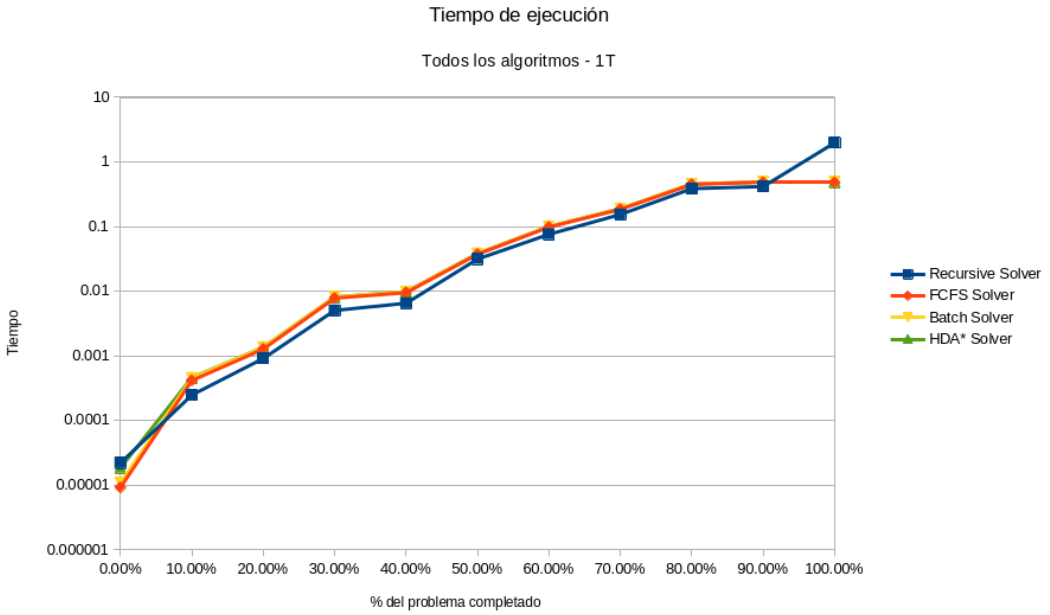
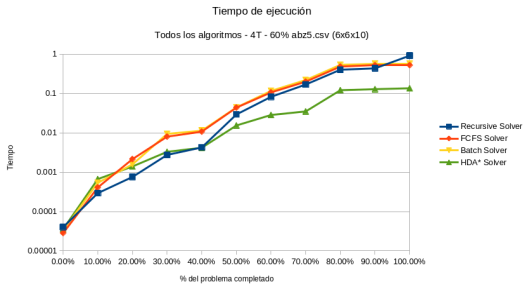
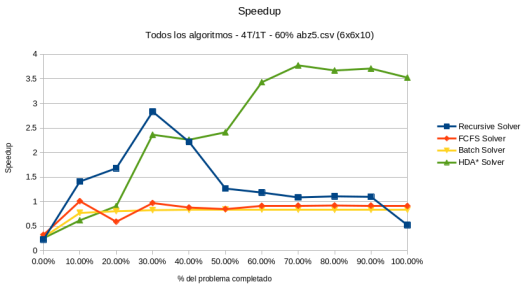


FIGURA 4.16: Tiempo de ejecución de todos los algoritmos (1 hilo).



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

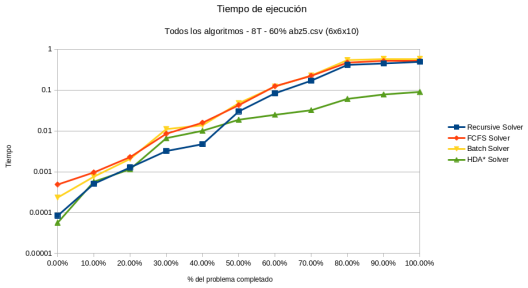


(B) *Speedup* de todos los algoritmos (4 hilos).

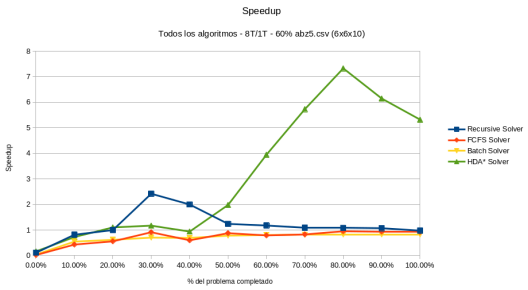
FIGURA 4.17: Métricas con 4 hilos.

Al utilizar cuatro hilos (figura 4.17), 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 4.18), el algoritmo HDA* incremente aún más su diferencia en el tiempo de ejecución con respecto al resto de algoritmos, aunque esta vez el *speedup* fluctua más. De cualquier forma, parece ser capaz de alcanzar casi 8.



(A) Tiempo de ejecución de todos los algoritmos (8 hilos).



(B) *Speedup* de todos los algoritmos (8 hilos).

FIGURA 4.18: Métricas con 8 hilos.

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.

NOTA

En la inmensa mayoría de observaciones realizadas, el algoritmo Batch Solver ha sido capaz de resolver el 30% del problema en menos tiempo que cualquier otro algoritmo. Esta reducción de tiempo se debe a que este algoritmo ejecuta el A* tantas veces como hilos haya disponibles de forma simultánea, facilitando la obtención de planificaciones profundas más rápido. En otras palabras, el algoritmo es más veloz al iniciar el problema pero es más lento al final.

4.8.5. Comparativa con Dijkstra

Como se indicó brevemente al inicio de este documento (Subsección 3.1.1), el algoritmo A* es una evolución del de Dijkstra donde la principal diferencia es la inclusión de una función heurística utilizada para seleccionar el siguiente nodo a explorar. Entonces, se podría decir que el algoritmo de Dijkstra es una versión particular del algoritmo A* donde la función heurística retorna siempre 0.

La principal conclusión obtenida de las siguientes observaciones (Figura 4.19) es que el *speedup* del algoritmo A* frente a Dijkstra es tan grande que incluso resulta complicado realizar pruebas. Esto se debe a que los conjuntos de datos que se pueden resolver utilizando Dijkstra en un tiempo razonable son resueltos en microsegundos por A* y los que son resueltos en un tiempo razonable por A* requerirían meses de ejecución para resolverlos utilizando Dijkstra [8].

Como era de esperar, el rendimiento de las implementaciones A* es notablemente superior al de Dijkstra en lo que corresponde a tiempo de ejecución. El gráfico anterior muestra que como poco el algoritmo A* tiene un *speedup* de 8000 frente a Dijkstra en el conjunto de datos estudiado.

[8] Es recomendable evitar extraer conclusiones de observaciones cuyo tiempo de ejecución es muy reducido. Existe un gran número de variables (cambios de contexto, fallos de caché, interrupciones del OS) que pueden afectar al tiempo de ejecución de un algoritmo, desvirtuando la medición.

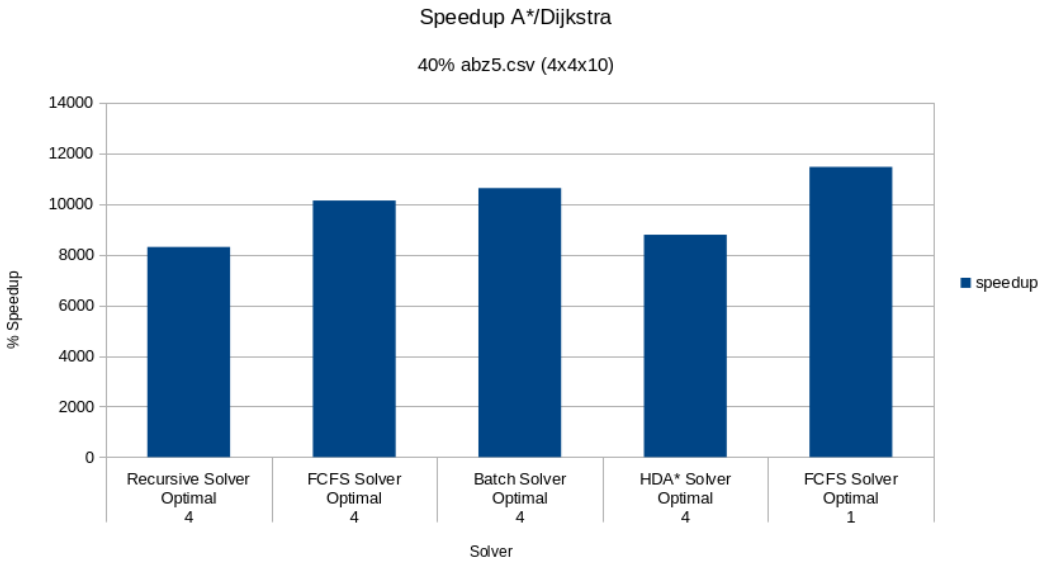


FIGURA 4.19: *Speedup* de A* frente a Dijkstra

No obstante, las ejecuciones de Dijkstra han sido capaces de obtener un *makespan* mejor que las de A*. El algoritmo de Dijkstra explorará todos los nodos posibles hasta obtener una solución óptima mientras que el A* simplemente explorará nodos hasta que se cumpla la condición de salida [9].

El algoritmo A* es completo, retornará una solución al problema siempre y cuando exista al menos una. Por otro lado, el algoritmo A* no tiene por qué retornar la solución óptima al problema. Esta característica depende de la función heurística encargada de calcular el coste H. Si la función heurística subestima el coste H real (es cota inferior del coste H real) entonces A* retornará la solución óptima. En otro caso, A* retornará una solución que puede ser la óptima.

De cualquier forma, las funciones heurísticas no son categorizables en una u otra clase, más bien se trata de un espectro. Las funciones heurísticas que subestimen el coste H real se denominan 'optimistas' mientras que las que lo igualan o sobreestiman se denominan 'pesimistas'.

FUNDAMENTAL

Las funciones optimistas retornarán los resultados óptimos pero requerirán un mayor tiempo de ejecución ya que se verán obligadas a explorar un mayor número de nodos mientras que las pesimistas no tienen por qué retornar un resultado óptimo y tendrán una mayor velocidad porque no explorarán tantas posibilidades.

Al extremo de las funciones optimistas se encuentra el algoritmo de Dijkstra cuyo heurístico retorna siempre 0 (no hay coste más bajo) y al extremo de las funciones pesimistas se podría encontrar la siguiente función (4.20) donde J es el número total de trabajos, T el de tareas y $D_{i,j}$ la duración de la tarea j del trabajo i.

[9] En este caso que todas las tareas estén planificadas

$$cost_h = \sum_{j=0}^{J} \sum_{t=0}^{T} D_{j,t}$$

FIGURA 4.20: Función heurística pesimista

Una función que calcula el sumatorio de todas las duraciones de las tareas restantes por planificar. Este heurístico sería lo equivalente a planificar que cada tarea se ejecute una a una de forma que sólo un trabajador se encuentra activo mientras el resto esperan (aunque puedan hacer tareas de forma paralela).

En algún punto intermedio entre estos dos extremos se encontraría el heurístico óptimo utilizado en la mayoría de las pruebas de esta investigación, pero es difícil de saber si se acerca más a los algoritmos optimistas o pesimistas. La principal dificultad de implementar el algoritmo A* es entonces decidir una función heurística que sea lo suficientemente pesimista para retornar resultados óptimos pero no tan pesimista que explore demasiadas alternativas.

4.8.6. 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.

4.8.6.1. 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.

4.8.6.2. 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 [10]. 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:

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

[10] Si existiese más de un nodo intermedio conocido, el problema se podría seguir subdividiendo entre más hilos.

Capítulo 5

Conclusiones

Observaciones y Trabajos futuros

5.1. El algoritmo A*

El algoritmo A* es de gran utilidad en un gran abanico de ámbitos. Realmente, se trata de un algoritmo apto para cualquier problema que esté formado por un grupo de nodos relacionables entre ellos dodne se quiera buscar una ruta entre ellos desde un inicio hasta un fin.

Desde el punto de vista más abstracto sólo existen tres funciones dependientes del problema en particular mientras que el resto del algoritmo sirve para cualquier problema:

- Dado un nodo conocer si es el objetivo o no.
- Dado un nodo conocer sus vecinos.
- Dado un nodo conocer sus costes G y H.

5.2. Principal cuello de botella

Existen tres principales operaciones necesarias para ejecutar el algoritmo A*:

- Obtención del siguiente nodo a explorar del `open_set`.
- Obtención de vecinos de un nodo.
- Inserción de vecinos en el `open_set`.

La obtención de vecinos de un nodo es un algoritmo dependiente del problema a resolver. Con el diseño de estructuras de datos e implementación correctos es fácil minimizar el coste de esta operación.

La obtención del siguiente nodo a explorar del `open_set` puede ser resuelta en O(1) por cualquier desarrollador con un conocimiento básico sobre estructuras de datos como listas enlazadas.

La inserción de vecinos en el `open_set` es más complicada ya que está estrechamente relacionada con la obtención del siguiente nodo a explorar. Generalmente el `open_set` se implementará como una lista ordenada de forma que el primer elemento sea el siguiente nodo a explorar. Esto implica que los vecinos se deben introducir en orden, insertando elementos en medio de la lista. La operación de insertar elementos en medio de una lista suele ser costosa. En esta investigación se ha conseguido hacer en O(n) utilizando una lista doblemente enlazada y un iterador que compara e inserta en el mismo barrido.

Es posible reducir el coste computacional de insertar vecinos en el `open_set`, pero esto implica un aumento en el coste de obtener el siguiente nodo a explorar. Sería necesario diseñar una estructura de datos distinta para reducir el coste computacional de una operación sin incrementar el de la otra. Seguramente esta estructura de datos sea también más dependiente de la memoria y como ya se ha visto en HDA*, el diseño de esta estructura debería facilitar el uso de varios hilos.

El principal problema del `open_set` es que la complejidad de sus operaciones incrementa a medida que se añaden más nodos. Es posible que otra forma de optimizar esta estructura se encuentre en hallar cómo almacenar menos nodos en ella.

5.3. Heurísticos

Como ya se discutió en la sección comparativa con Dijkstra (4.8.5), el diseño de la función heurística es crucial para determinar si la implementación del A* será admisible[1] y el tiempo que requerirá para resolver el problema.

Ignorando el paralelismo, la decisión del diseño de la función heurística es la principal clave que definirá el rendimiento y calidad de las soluciones propuestas por el algoritmo. Comprender las diferencias entre funciones optimistas y pesimistas y ser capaz de hallar la función que retorne resultados óptimos en el menor tiempo posible es el principal desafío de la implementación de este algoritmo.

[1] Si retornará siempre el resultado óptimo.

Apéndice A

Código Fuente

A.1. Chronometer

```
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 };
```

LISTING A.1: Clase Chronometer utilizada para tomar mediciones

A.2. Random Solver

```
1  class RandomSolver : public AStarSolver
2  {
3  private:
4      State get_random_next_state(const State &, std::mt19937 &) const;
5
6  public:
7      RandomSolver() = default;
8      ~RandomSolver() override = default;
9
10     State solve(
11         std::vector<std::vector<Task>>,
12         std::size_t,
13         Chronometer &) const override;
14
15     std::string get_name() const override { return "RANDOM Solver"; };
16 };
17
18 State RandomSolver::solve(
19     std::vector<std::vector<Task>> jobs,
20     std::size_t n_workers,
21     Chronometer &c) const
22 {
23     const std::size_t nJobs = jobs.size();
24     if (nJobs == 0)
25         return State();
26     const std::size_t nTasks = jobs[0].size();
27     if (nTasks == 0)
28         return State();
29     if (n_workers == 0)
30         n_workers = nTasks;
31
32     std::random_device rd;
33     std::mt19937 gen(rd());
34
35     const auto startingSchedule = std::vector<std::vector<int>>(nJobs,
       std::vector<int>(nTasks, -1));
36     const auto startingWorkersStatus = std::vector<int>(n_workers, 0);
37     auto currentState = State(jobs, startingSchedule,
       startingWorkersStatus);
38
39     while (!currentState.is_goal_state())
40     {
41         currentState = this->get_random_next_state(currentState, gen);
42         c.process_iteration(currentState);
43     }
44
45     return currentState;
46 }
47
48 State RandomSolver::get_random_next_state(const State &currentState,
       std::mt19937 &gen) const
49 {
50     std::vector<State> neighbors = currentState.get_neighbors_of();
51     std::uniform_int_distribution<> dis(0, (int)neighbors.size() - 1);
52     return neighbors[dis(gen)];
53 }
```

LISTING A.2: Clase RandomSolver utilizada para resolver un problema aleatoriamente

A.3. Read and Cut

```
struct ReadTaskStruct
{
    unsigned int duration;
    std::vector<unsigned int> qualified_workers;
};

std::vector<std::vector<Task>> get_jobs_from_file(
    const std::string &filename
) {
    std::ifstream file(filename);
    std::string my_string;
    std::string my_number;
    std::vector<std::vector<ReadTaskStruct>> data;
    std::vector<std::vector<Task>> out;
    if (!file.is_open())
    {
        std::cerr << "Could not read file" << std::endl;
        file.close();
        return out;
    }
    while (std::getline(file, my_string))
    {
        std::stringstream line(my_string);
        data.emplace_back();
        bool is_worker = true;
        unsigned int qualified_worker = 0;
        while (std::getline(line, my_number, ';'))
        {
            if (is_worker)
            {
                qualified_worker = stoi(my_number);
                is_worker = false;
            }
            else
            {
                data[data.size() - 1].emplace_back();
                data[data.size() - 1][
                    data[data.size() - 1].size() - 1
                ].duration = stoi(my_number);
                data[data.size() - 1][
                    data[data.size() - 1].size() - 1
                ].qualified_workers = std::vector<unsigned int>(
                    1,
                    qualified_worker
                );
                is_worker = true;
            }
        }
    }
    for (std::size_t job_idx = 0; job_idx < data.size(); job_idx++)
    {
        out.emplace_back();
        const std::vector<struct ReadTaskStruct> currentJob = data[
            job_idx
        ];
        for (
            std::size_t task_idx = 0;
            task_idx < currentJob.size();
            task_idx++
        ) {
```

```
61              const struct ReadTaskStruct currentTask = currentJob[
62                  task_idx
63              ];
64              unsigned int h_cost = 0;
65              for (
66                  std::size_t unscheduled_task_idx = task_idx;
67                  unscheduled_task_idx < currentJob.size();
68                  unscheduled_task_idx++
69              )
70                  h_cost += currentJob[unscheduled_task_idx].duration;
71              out[job_idx].emplace_back(
72                  currentTask.duration,
73                  h_cost,
74                  currentTask.qualified_workers
75              );
76          }
77      }
78      file.close();
79      return out;
80  }
81
82  std::vector<std::vector<Task>> cut(
83      std::vector<std::vector<Task>> jobs,
84      float percentage
85  ) {
86      percentage = percentage < 0 ? 0 : percentage;
87      percentage = percentage > 1 ? 1 : percentage;
88      const unsigned int jobs_to_keep = int(
89          float(jobs.size()) * percentage
90      );
91      const unsigned int tasks_to_keep = int(
92          float(jobs[0].size()) * percentage
93      );
94      std::vector<std::vector<Task>> new_jobs;
95
96      for (unsigned int job_idx = 0; job_idx < jobs_to_keep; job_idx++)
97      {
98          new_jobs.emplace_back();
99          for (
100             unsigned int task_idx = 0;
101             task_idx < tasks_to_keep;
102             task_idx++
103         )
104             new_jobs[job_idx].push_back(jobs[job_idx][task_idx]);
105     }
106
107     return new_jobs;
108 }
```

LISTING A.3: Funciones utilizadas para leer y cortar conjuntos de datos

A.4. Heurísticos

A.4.1. Slow - 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     return max_element == h_costs.end() ? 0 : *max_element;
14 }
```

A.4.2. Fast

```
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();
       job_idx++)
5          for (std::size_t task_idx = 0; task_idx < this->m_jobs[job_idx
       ].size(); task_idx++)
6              if (this->m_schedule[job_idx][task_idx] == -1)
7                  unscheduled_tasks_count += this->m_jobs[job_idx][
       task_idx].get_duration();
8      return unscheduled_tasks_count;
9  }
```

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. 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.

[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. URL: https://ojs.aaai.org/index.php/ICAPS/article/view/13350.

[Pin12] Michael Pinedo. «Scheduling: Theory, Algorithms, And Systems». En: (ene. de 2012). URL: https://www.academia.edu/45241856/Scheduling_Theory_Algorithms_and_Systems_M_Pinedo.

[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.

[NSG17] Alexandre S. Nery, Alexandre C. Sena y Leandro S. Guedes. «Efficient Pathfinding Co-Processors for FPGAs». En: *2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*. 2017, págs. 97-102. DOI: 10.1109/SBAC-PADW.2017.25.

[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. 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. 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.

Índice alfabético

A

C

E

H

I

M

O

R

S

T