Universidad de Oviedo

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN

GRADO DE INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

ÁREA DE
CIENCIA DE LA COMPUTACIÓN E INTELIGENCIA ARTIFICIAL

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

Autor:
D. Rodríguez López, Alejandro

Tutor:
D. Palacios Alonso, Juan José

12 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 D. Rodríguez López, Alejandro

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

Existen una infinidad de algoritmos con distintos diseños, propósitos y utilidades. Como es de esperar, algunos requieren más tiempo para ser ejecutados que otros pero a diferencia de lo que muchos se podrían esperar, obtener un algoritmo que no se pueda resolver en un tiempo razonable es más simple de lo que parece.

La complejidad de un algoritmo se refiere a la tasa de variación del tiempo de ejecución en función del tamaño de la entrada. Entonces, la complejidad es la primera derivada del tiempo de ejecución.

La complejidad de un problema viene determininada por la complejidad del algoritmo más eficiente que lo resuelve. Los problemas son frecuentemente categorizados en P y NP. Aquellos pertenecientes a la categoría P son aquellos para los cuáles se conoce un algoritmo capaz de resolverlos en tiempo polinomial. Los problemas NP son aquellos para los que no se conoce ningún algoritmo de complejidad polinomial que los resuelva, pero que dada una posible solución, se puede validar en tiempo polinomial. Además de estas categorías, podemos destacar los problemas NP-hard. Un problema A es NP-hard si todo problema NP B se puede reducir a él polinómicamente. Es decir, son al menos tan difíciles como los problemas NP.

Frecuentemente los problemas NP-Hard son resueltos utilizando tamaños de entrada muy pequeños o algoritmos que den un resultado aproximado que es posteriormente contrastado con el tiempo necesario para obtenerlo. Este tipo de problemas aparecen frecuentemente en ámbitos industriales. En estos casos, su resolución permite optimizar sus procesos o crear productos más innovadores. En estos entornos industriales es común el uso de hardware específico como las FPGA u otros ASIC.

Este proyecto tiene como objetivo principal explorar los límites del A*, uno de los algoritmos más conocidos con aplicaciones en ámbitos industriales y descubrir los beneficios del paralelismo aplicables a este algoritmo. Adicionalmente, se observará el rendimiento de una implementación de la misma solución en una FPGA, atendiendo a los límites de este hardware en términos de qué tamaños de problema se pueden resolver en un tiempo razonable.

1.2. Alcance

En este proyecto, nos centramos en la resolución del Job Shop Scheduling Problem (JSP). El JSP 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.

Para resolver este problema, nos centramos en el algoritmo A*. Propondremos varios heurísticos y estrategias de paralelismo distintas. Finalmente, compararemos los resultados utilizando la arquitectura x86 tradicional y FPGAs.

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 mismas.

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. Se llevará a cabo 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:

1. Existen un número natural conocido de trabajos.
2. 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.
3. Cada tarea puede tener una duración distinta.
4. La duración de cada tarea es un número natural conocido.
5. Existe un número natural conocido de trabajadores.
6. Cada tarea tiene un trabajador asignado, de forma que sólo ese trabajador puede ejecutar la tarea.
7. Una vez iniciada una tarea, no se puede interrumpir su ejecución.
8. Un mismo trabajador puede intercalar la ejecución de tareas de diferentes trabajos.
9. Un trabajador sólo puede realizar una tarea al mismo tiempo.
10. Los tiempos de preparación de un trabajador antes de realizar una tarea son nulos.
11. 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

A continuación se describe una notación utilizada en los ejemplos de este documento y en diversas explicaciones.

El problema del JSP consiste en la planificación de un número natural (M) de trabajos $(J_0 \dots J_{M-1})$ donde cada trabajo J_i está formado por un número (N) de tareas $(T_{i,0} \dots T_{i,N-1})$.

A cada tarea $T_{i,j}$ le corresponden el par de atributos ($D_{i,j}$ y $W_{i,j}$) donde $D_{i,j}$ es la duración de la tarea y $W_{i,j}$ es el trabajador que la debe ejecutar.

2.3.1.1. Conjunto de datos

El conjunto de datos es el inicio del problema a resolver, contiene los datos de entrada necesarios para ejecutar cualquier algoritmo. En particular, está compuesto por un listado de trabajos formado por sus tareas, duraciones y trabajadores:

0. (2, A), (5, B), (1, C)
1. (3, B), (3, C), (3, A)

En este ejemplo, el conjunto de datos estaría formado por dos trabajos (J_0, J_1) con tres tareas cada uno ($T_{i,0} \dots T_{i,2}$). Las tareas serán ejecutadas por tres trabajadores diferentes (A, B y C).

La tarea $T_{0,0}$ tiene una duración de dos instantes y debe ser realizada por el trabajador A. La tarea $T_{1,2}$ tiene una duración de tres instantes y también debe ser realizada por el trabajador A. Por lo que estas dos tareas no podrán ser ejecutadas de forma simultánea ya que ambas requieren al mismo trabajador.

2.3.1.2. Planificación

La planificación es un dato propio del estado. Indica el instante de tiempo en el que se inicia cada tarea de forma que el elemento en la posición i del vector se corresponde con el instante de inicio de la tarea i: $(0, 3, -1)$ 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 $T_{0,0}$ se inicia en el instante 0 mientras que la $T_{1,1}$ está aún sin planificar.

2.3.1.3. Estado trabajadores

El estado de los trabajadores muestra en forma de lista el instante de tiempo donde cada trabajador queda libre. Simbólicamente, w_i corresponde al instante en el que el trabajador i queda libre y W es el número total de trabajadores.

En los ejemplos, la estructura está construida de forma que el elemento k corresponde al instante de tiempo donde el trabajador k queda libre. $(2, 8, 0)$ indica que el trabajador 0 esta libre a partir del instante 2 ($w_0 = 2$), el 1 a partir del 8 ($w_1 = 8$) y el 2 a partir del 0 ($w_2 = 0$).

Aunque esta información no es parte de la salida del programa, es necesaria para el cómputo de la solución.

2.3.1.4. Función objetivo, *Makespan*

Existen varios objetivos que se pueden buscar con este problema de optimización. En este caso se selecciona la minimización del *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. Algunas versiones del algoritmo desarrolladas en este proyecto no tienen la certeza de obtener el resultado óptimo. Por ello, si una implementación A retorna un *makespan* menor que otra B, A es mejor que B. [2]

Dado un estado final (con todas sus tareas planificadas) el *makespan* se define como el $cost_g$ de ese estado. O lo que es lo mismo, el instante en el que el último trabajador queda libre:

$$\max_{0<i<W}(w_i)$$

EJEMPLO

Sea la siguiente planificación:

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

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

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

2.3.2. Ejemplo

A continuación se muestra una posible solución para un ejemplo de pequeño tamaño donde cada línea corresponde a un trabajo y cada tupla al par (duración, trabajador).

0. (2, A), (5, B), (1, C)
1. (3, B), (3, C), (3, A)

Los siguientes diagramas (Figura 2.1 y 2.2) representan los trabajos y planificación respectivamente. En el primero cada nodo representa una tarea, los arcos continuos el orden y los discontinuos el plan de cada trabajador. El tiempo de inicio de cada tarea viene dado por la longitud del camino más largo desde i hasta el nodo de esa tarea, mientras que el makespan sería el camino más largo entre i y f. El segundo diagrama se conoce como diagrama de Gantt y muestra el instante de inicio, fin y duración de cada trabajo. [3]

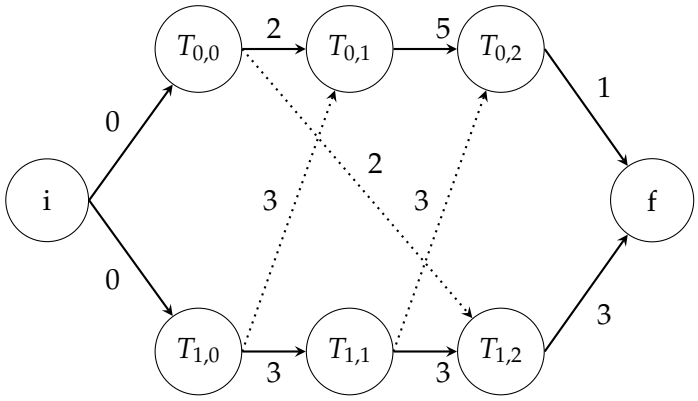


FIGURA 2.1: Diagrama ejemplo.

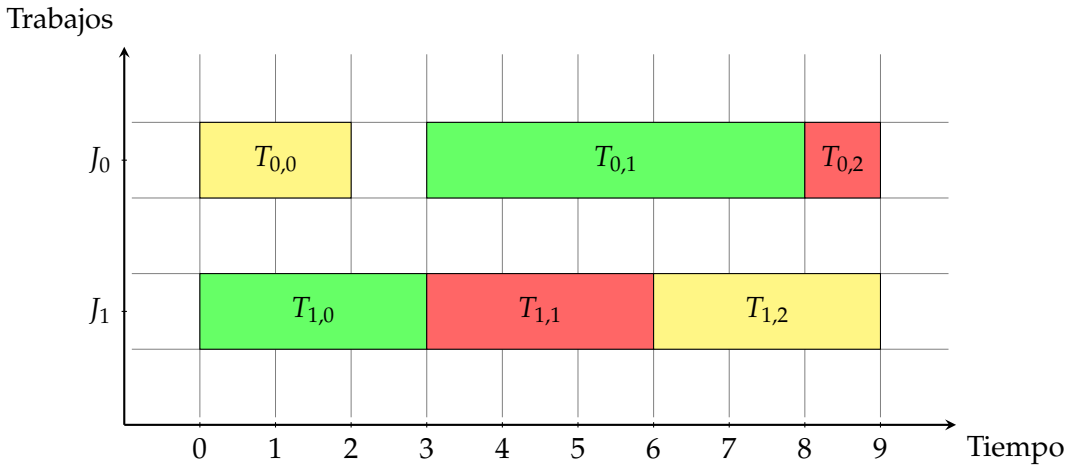


FIGURA 2.2: Diagrama planificación ejemplo.

[3] Otros diagramas: Pert (B.1) y Gantt (B.2).

Capítulo 3

Metodología de trabajo

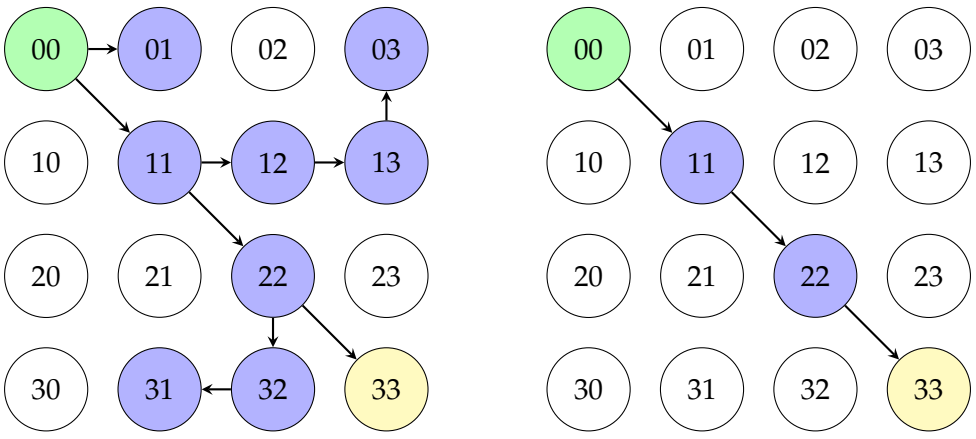
Método de resolución y comparativas

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.is_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 cumple las condiciones necesarias para considerarse como final, el algoritmo retorna (ln10-11). En otro caso, calcula los sucesores del estado actual y los procesa individualmente (ln13-21). 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 (ln17-21).

El método `estado.is_estado_final()` consiste en una función que dado un estado retorna `true` si es el estado final y `falso` si no lo es. Lógicamente, esta función tendrá una implementación diferente dependiendo del problema que se trate de resolver. En este caso se ha implementado de forma que retorne `true` cuando todas las tareas del estado hayan sido planificadas y `false` en cualquier otro caso. [1]

[1] Existen otras implementaciones equivalentes como que la longitud de `calcular_sucesores(estado)` sea 0.

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. Es decir, dados dos estados debe ser posible 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. En concreto, el estado está compuesto por:

- 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. Para obtenerlo, se utiliza una función heurística.

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)

Donde el estado:

- Planificación parcial:
  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 J_0, $T_{0,2}$.

- Instante en el que finaliza su tarea predecesora ($T_{0,1}$): $3 + 5 = 8$
- Instante en el que el trabajador 2 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 J_1, $T_{1,1}$.

- Instante en el que finaliza su tarea predecesora ($T_{1,0}$): $0 + 3 = 3$
- Instante en el que el trabajador 2 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 retorna una cota inferior del coste restante, por lo que dirigirá al algoritmo a resultados óptimos o que se encuentren relativamente cerca del óptimo.

Sea $J_{j,k}$ el índice de la primera tarea sin planificar del trabajo J_j:

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

EJEMPLO

El conjunto de datos:

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

La planificación y 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 retorna una cota superior del coste restante, por lo que dirigirá al algoritmo hacia cualquier nodo solución independientemente de si es óptimo o no.

Sea $J_{j,k}$ el índice de la primera tarea sin planificar del trabajo J_j:

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

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:

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

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.

4.2.1.2.3. Heurístico Dijkstra

El algoritmo de Dijkstra se puede entender como un caso particular del algoritmo A* donde el heurístico utilizado siempre retorna cero. Esta implementación del heurístico retorna lógicamente una cota inferior del coste restante por lo que siempre retornará el resultado óptimo.

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` .

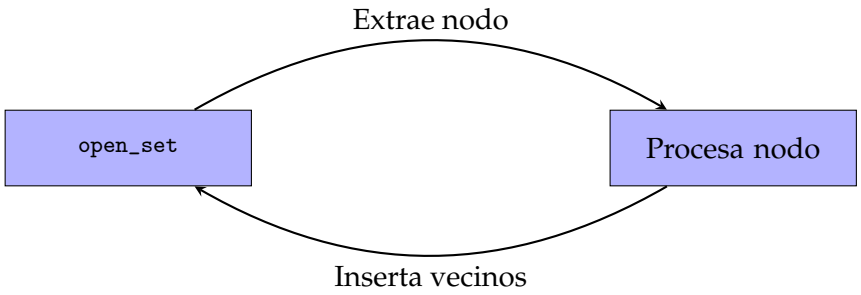
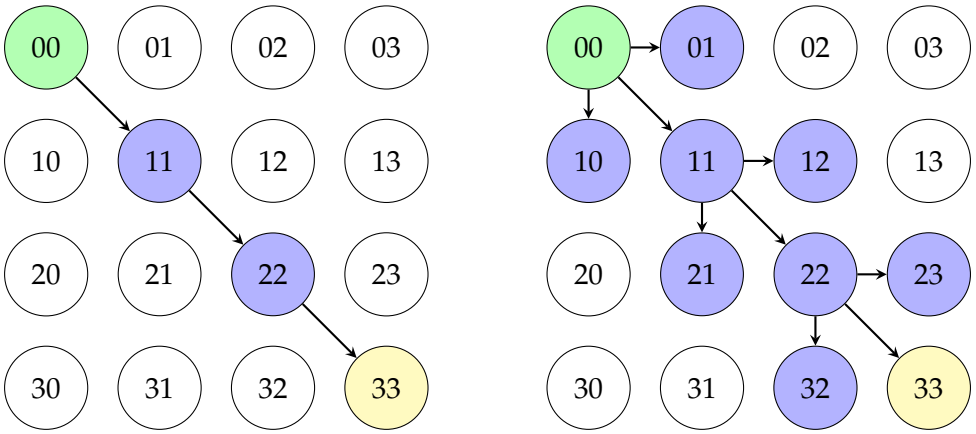


FIGURA 4.1: Representación de la estrategia FCFS

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.

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

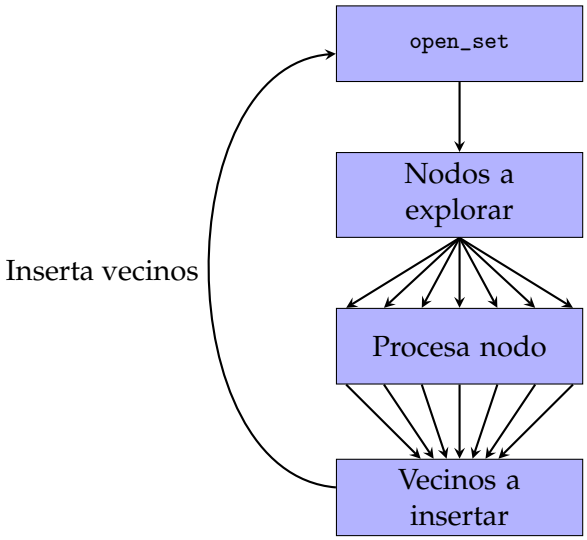


FIGURA 4.3: Representación de la estrategia Batch

4.2.2.2.1. 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`.

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.

1. Calcular sucesores del estado inicial.
2. Asignar un hilo a cada sucesor.
3. Para cada sucesor, 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.

FUNDAMENTAL

El Recursive Solver es equivalente a ejecutar el algoritmo A* N veces de de forma individual siendo N el número de sucesores 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 sucesores 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.

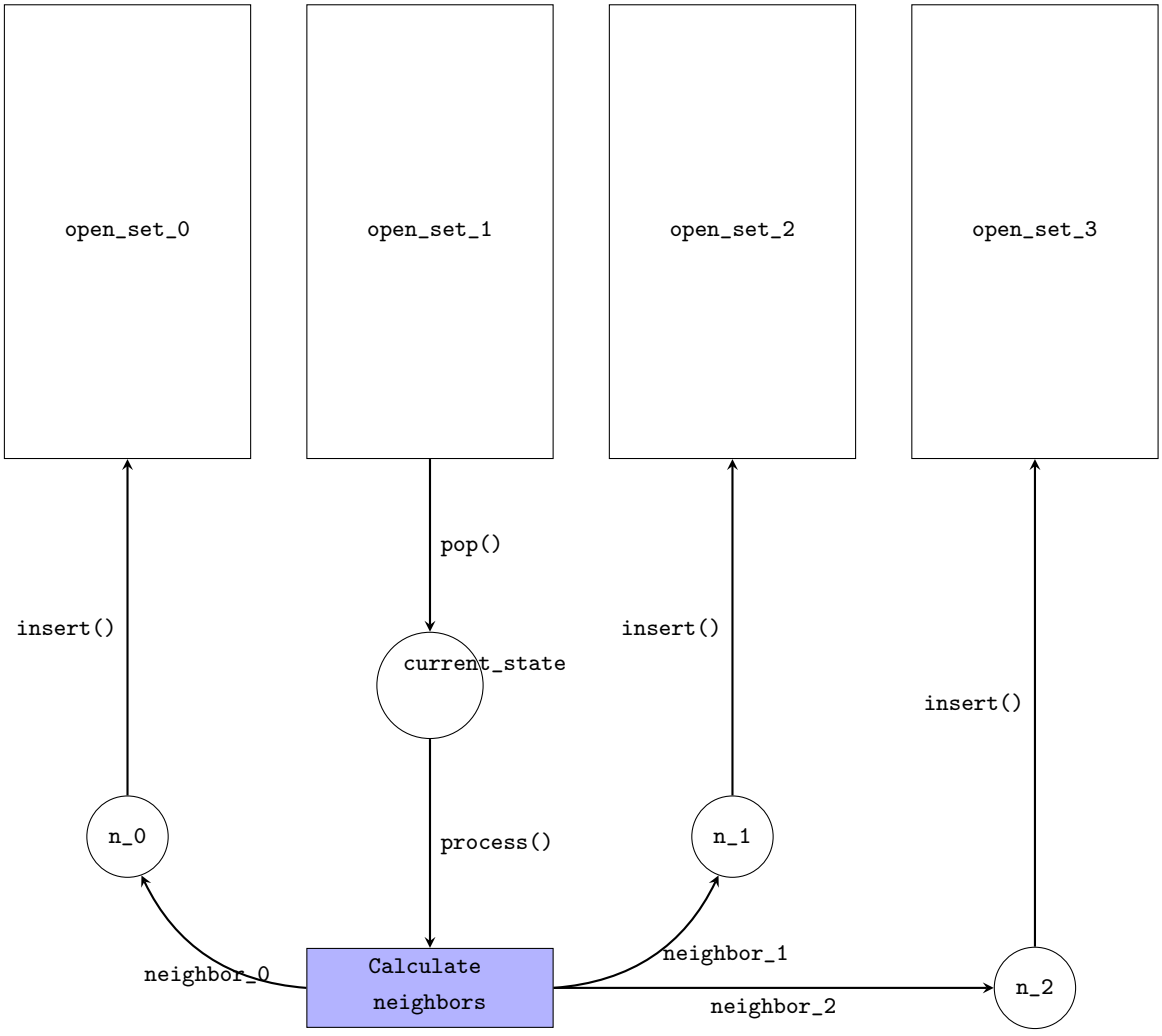


FIGURA 4.4: Representación de la estrategia HDA*

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

NOTA

Al igual que la implementación FCFS (4.2.2.1), el HDA* no tiene sincronización entre hilos por lo que dos ejecuciones distintas pueden retornar resultados diferentes y por tanto tampoco garantiza retornar el resultado óptimo.

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.

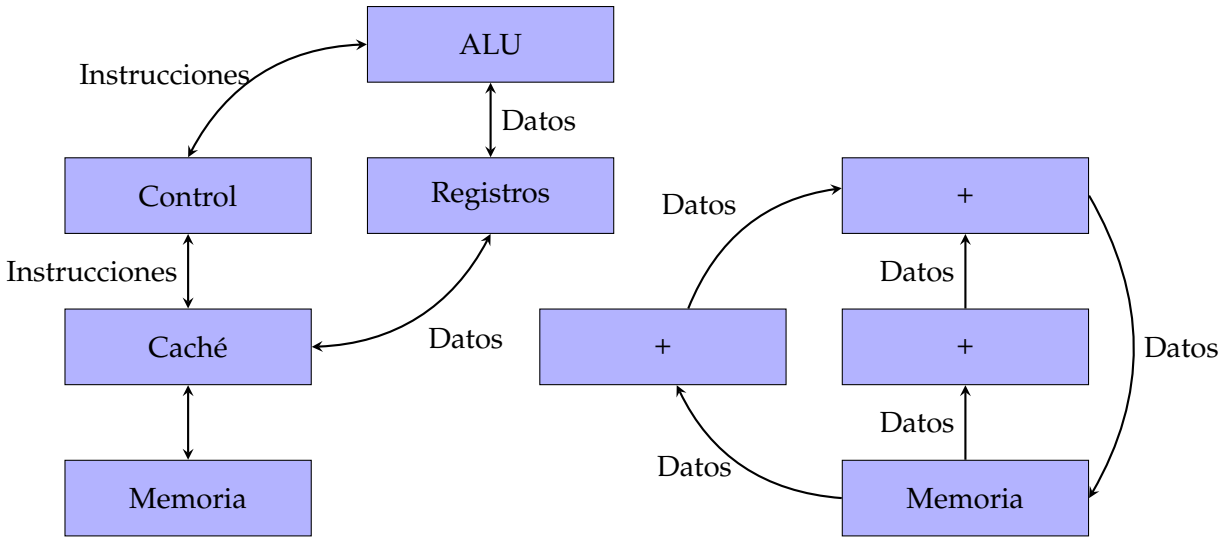
El funcionamiento de una FPGA no se debe confundir con el de un procesador convencional. Las CPUs comúnmente encontradas en ordenadores utilizan la arquitectura de Von Neumann (Véase Figura 4.5a) para realizar cálculos. Por otro lado, las FPGAs no utilizan esta arquitectura. La principal diferencia se 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.

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

[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.



(A) Arquitectura Von Neumann (B) Arquitectura FPGA

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.

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.

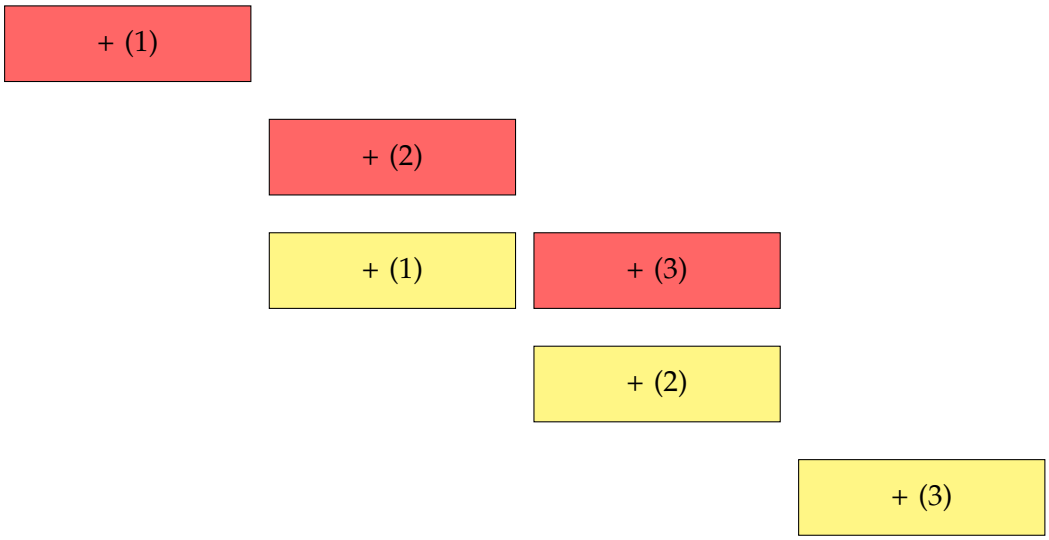


FIGURA 4.6: Dos operaciones ejecutadas en el mismo *pipeline*.

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.

4.3.1. Síntesis

Se ha llevado a cabo una síntesis de parte del código C++ a VHDL con el objetivo de observar principalmente los beneficios de utilizar este hardware. Adicionalmente este procedimiento ha resultado de utilidad para reconocer que a pesar de la existencia de herramientas como HLS, la migración de código existente en C++ al entorno de la FPGA no es un procedimiento sin inconvenientes.

La herramienta HLS nos permite seleccionar una única función a acelerar en la FPGA. Además, esta función no puede estar compuesta por estructuras de datos (ni punteros a ellas) que contengan punteros, limitando nuestro abanico de datos de entrada a prácticamente tipos de datos primitivos. Si se introduce código C++ (como en el caso de esta investigación), debe ser de la versión C++11 o inferior ya que muchas características de versiones posteriores no están soportadas. Finalmente, la función a acelerar no puede ser un método de una clase, descalificando la fácil migración de cualquier programa que se base en programación orientada a objetos.

Lógicamente estas restricciones dificultan la síntesis de una función completa encargada de ejecutar el algoritmo A* [5]. Alternativamente, se puede sintetizar una función utilizada durante la ejecución del algoritmo de forma que sólo esa función esté acelerada en la FPGA. De las posibles opciones, una mayoría de ellas quedan descartadas por no tener el coste computacional suficiente, mientras que otras quedan decartadas por cachearse tras ser procesadas por primera vez (i.e. funciones hash). Por último, queda la posibilidad de acelerar la función encargada con insertar elementos en el `open_set`, el principal cuello de botella del programa.

[5] Además, debido a la poca capacidad de memoria RAM de la FPGA, una implementación de esta función sería capaz de procesar problemas de tamaño 5x5x10 aproximadamente como máximo.

4.3.2. Kernel

Se ha decidido escribir una versión del algoritmo A* monohilo en una función tradicional de forma que sea sintetizable. Entre las principales diferencias se encuentran los parámetros ya que no se pueden utilizar las estructuras de datos vistas hasta ahora.

```
1     void kernel_astar_foo(
2         unsigned int* in_jobDurations,
3         unsigned int* in_jobQualifiedWorkers,
4         unsigned int* in_size,
5         unsigned int* in_nQualifiedWorkers,
6         unsigned int* out_makespan
7     );
```

LISTING 4.1: Prototipo del kernel a sintetizar.

Como se puede ver, la función a sintetizar (denominada kernel) recibe únicamente parámetros de tipo `unsigned int*`. La mayoría de ellos están precedidos de `in_` ya que son datos de entrada, pero también se pasa la dirección a un `out_makespan` que es el resultado del algoritmo. Al igual que una GPGPU, la FPGA no tiene acceso a la memoria RAM de la CPU directamente, por lo que hay que utilizar punteros [6].

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 *A*x*B*x*C* cuando está compuesto por *A* trabajos, *B* tareas en cada uno y *C* trabajadores.

[6] En una GPGPU el proceso es algo más tedioso, teniendo que copiar manualmente de la memoria de la CPU a la de la GPGPU.

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

4.5.2. FPGA

Se ha utilizado una FPGA Zynq-7000 (C7Z010-1CLG400C) en una placa Digilent Inc. ZYBO Z7 10 para realizar las mediciones en este proyecto. Este hardware en específico cuenta con un gigabyte the memoria RAM, y un procesador ARM Cortex A9 de dos núcleos que no se utiliza en esta investigación. Una descripción más detallada del dispositivo en cuestión puede encontrarse en la (HTTPS) referencia de Digilent Inc.

4.6. Método de medición

Todas las versiones del programa imprimen por salida estándar datos que posteriormente son procesados en formato CSV. Para evitar valores atípicos, cada algoritmo se ejecuta varias veces. En el caso de versiones lentas se han tomado cinco mediciones por cada versión mientras que en el caso de las más veloces se han tomado cien mediciones por cada versión. La recolección de todos estos datos puede llevar aproximadamente 72 horas (tres días).

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[7]. 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.

[7] Véase A.1

4.6.1. 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 [8].

El *speedup* es una métrica utilizada para medir la aceleración de un algoritmo frente a otro. 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.

[8] 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. Resultados y Análisis

4.7.1. 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.

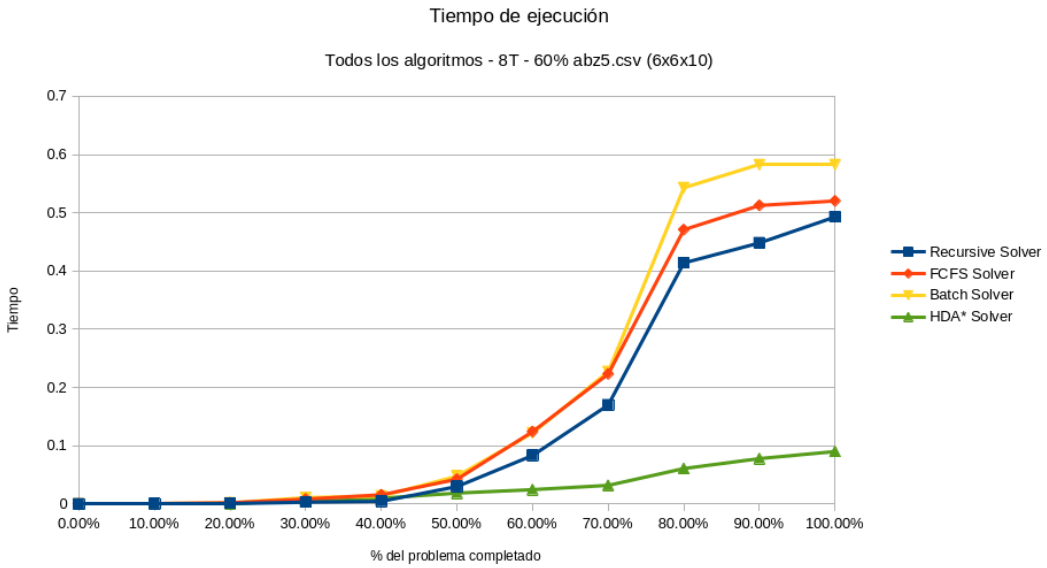


FIGURA 4.7: Tiempo de ejecución de un único problema.

Como se puede ver, el diagrama es de poca utilidad para comparar algoritmos debido a la complejidad del problema. Para poder observar con mejor los resultados, en las siguientes secciones se utilizan frencuentemente gráficas con ejes verticales en escala logarítica (figura 4.8).

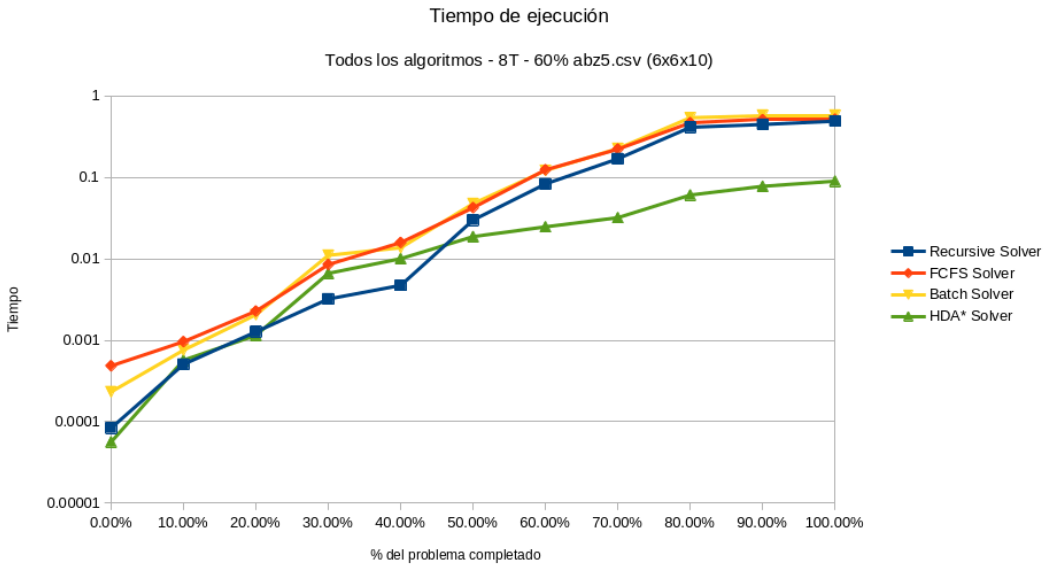


FIGURA 4.8: Tiempo de ejecución de un único problema (escala logarítmica).

Estas observaciones verifican que la complejidad del problema a resolver es factorial, 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.

4.7.2. Comparativa de heurísticos

4.7.2.1. Cotas Inferior v. Superior

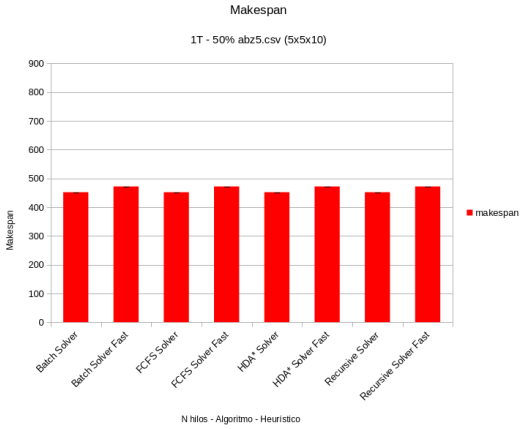
Como ya se discutió en la sección sobre optimización (4.2.1.2), existen varias implementaciones de la función heurística que da al algoritmo A* su particular comportamiento de ir 'dirigido' hacia la solución. En esta investigación se han implementado dos heurísticos: uno de ellos es una cota inferior del coste restante mientras que el otro es una cota superior. El heurístico de cota inferior busca una solución que se acerque a la óptima lo máximo posible mientras que el de cota superior busca la solución priorizando la velocidad del algoritmo. A continuación se comparan los heurísticos utilizando el 50% del conjunto `abz5.csv` (5x5x10).

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.

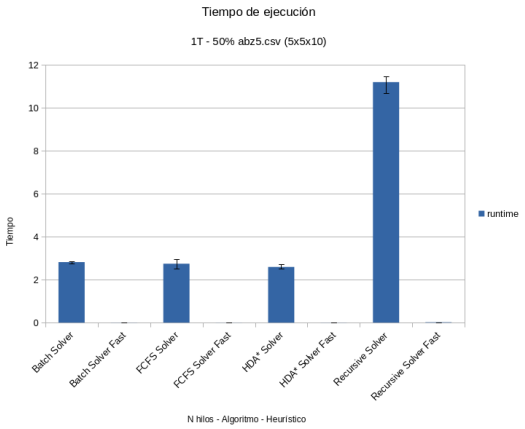
Aquellas implementaciones con sincronización (que retornan siempre el mismo resultado) muestran una mayor precisión que aquellas que no (FCFS y HDA*). Los algoritmos sin sincronización no se deberían utilizar junto al heurístico rápido ya que la pérdida en precisión no compensa el *speedup*. En los algoritmos con sincronización sin embargo sí existen argumentos para sopesar el uso del heurístico rápido ya que la pérdida en precisión podría compensar con creces el *speedup*.

FUNDAMENTAL

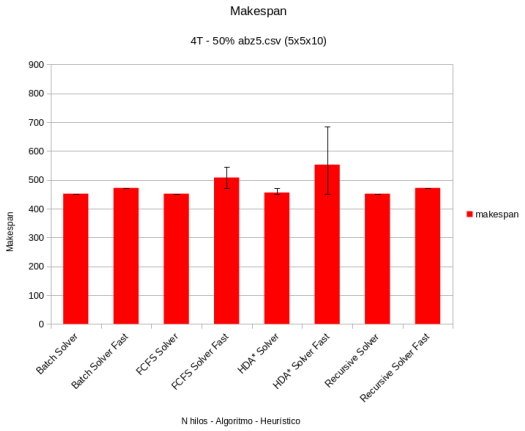
La principal conclusión de las siguientes observaciones es que el heurístico rápido puede ser considerado sólo cuando se utilizan algoritmos monohilo o con sincronización. Y siempre haciendo observaciones previas para comprobar que el tamaño del problema no es demasiado grande ya que a mayor el tamaño del problema, peores los resultados.



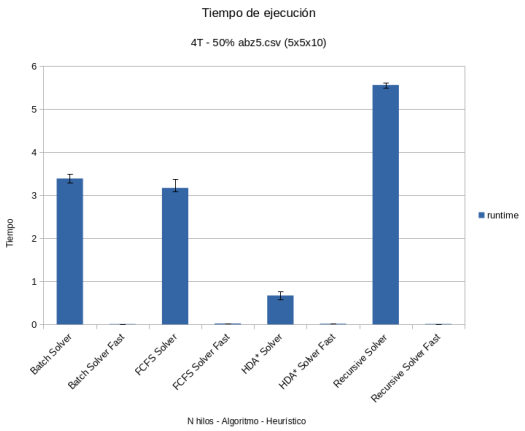
(A) *Makespan* heurísticos 1 hilo.



(B) Tiempo de ejecución heurísticos 1 hilo.

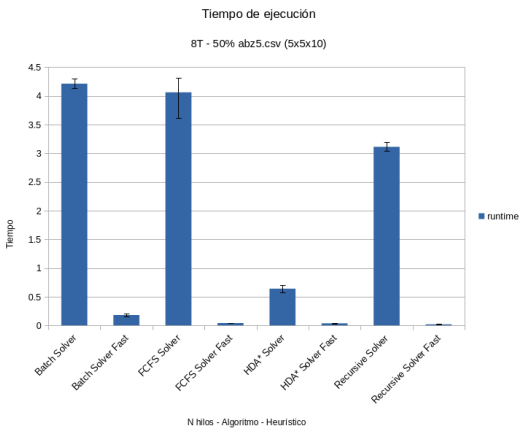


(C) *Makespan* heurísticos 4 hilos.



(D) Tiempo de ejecución heurísticos 4 hilos.

(E) *Makespan* heurísticos 8 hilos.



(F) Tiempo de ejecución heurísticos 8 hilos.

FIGURA 4.9: Métricas de heurísticos.

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 4.9e, 4.9c y 4.9a).

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.

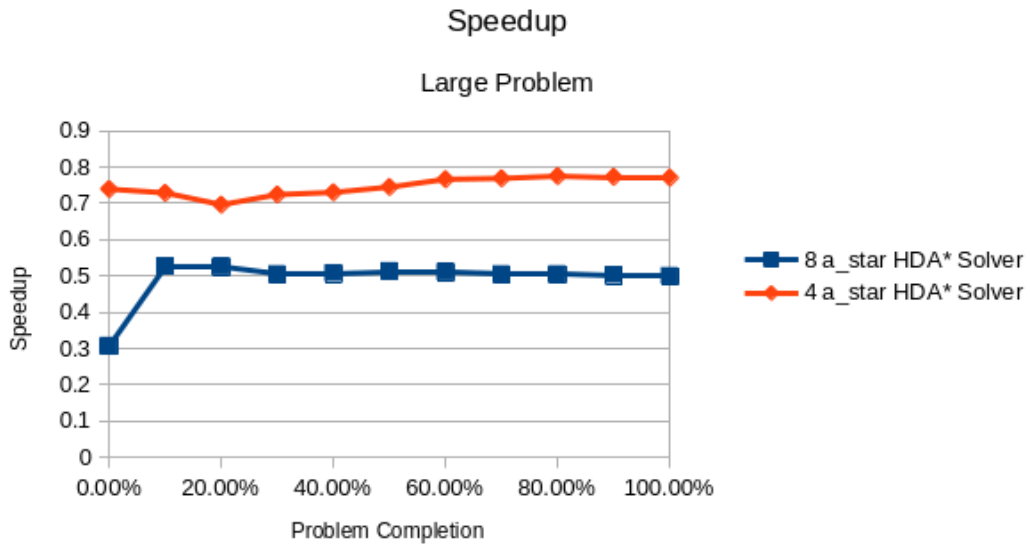


FIGURA 4.10: *Speedup* en un problema de gran tamaño.

La gráfica (4.10) 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.

NOTA

A menos que se indique lo contrario, las pruebas ejecutadas en esta investigación utilizan el heurístico de cota inferior ya que es el que retorna resultados óptimos.

4.7.2.2. 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* de una de las peores lecutras, la del heurístico rápido en el algoritmo HDA* con cuatro hilos.

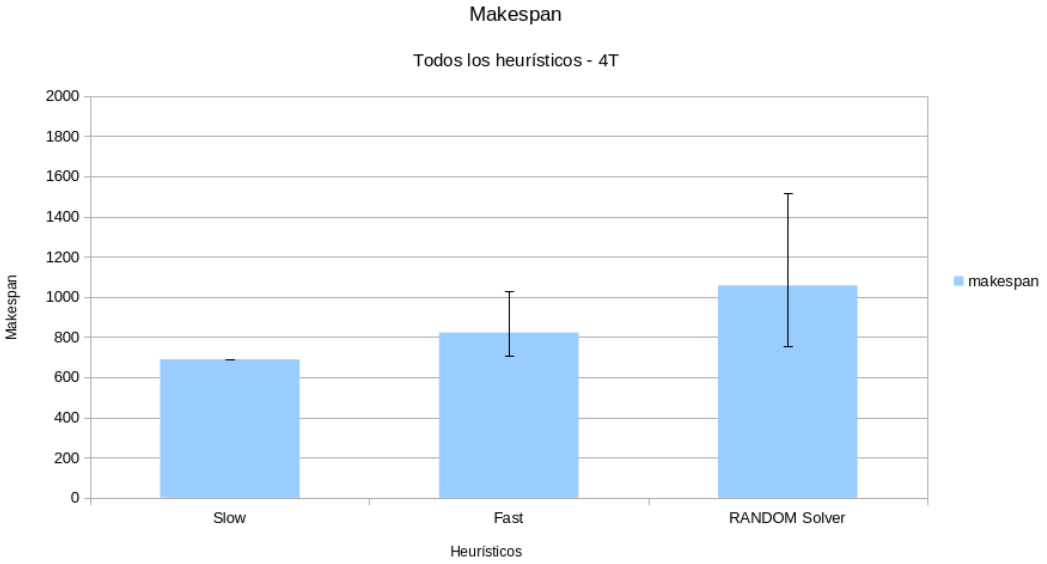


FIGURA 4.11: *Makespan* de diferentes heurísticos

La figura 4.11 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.

4.7.2.3. 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.

En esta subsección se ha estudiado el rendimiento del algoritmo de Dijkstra y los diferentes A* utilizando el heurístico de cota inferior, que retorna resultados óptimos a cambio de un mayor tiempo de ejecución.

La principal conclusión obtenida de las siguientes observaciones (Figura 4.12) 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 [9].

NOTA

Nótese que el *speedup* entre A* (utilizando el heurístico lento) y Dijkstra supera 8000 en todos los casos. Sabiendo que el heurístico lento es más lento que el heurístico rápido, se conoce por la propiedad transitiva que Dijkstra es más lento que el heurístico rápido.
Tratar de hacer pruebas con el objetivo de calcular Dijkstra con el heurístico rápido retornaría una diferencia de tiempo de tal magnitud que sería imposible comprender la diferencia entre ambos, más aún tratar de representarlo en una gráfica.

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.

Las ejecuciones de Dijkstra han sido capaces de obtener un *makespan* mejor que las implementaciones sin sincronización 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 [10].

El algoritmo A* es completo, retornará una solución al problema siempre y cuando exista al menos una. Esto no implica que sus resultados sean siempre óptimos (A diferencia de Dijkstra, cuyos resultados sí lo son). Que A* retorne soluciones óptimas o no 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.

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

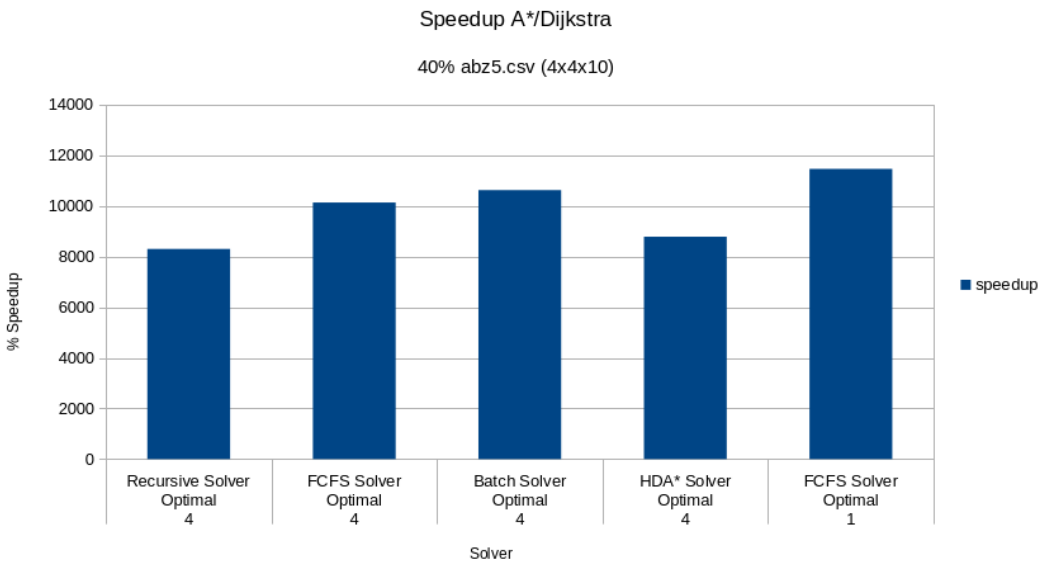


FIGURA 4.12: *Speedup* de A* frente a Dijkstra

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 (4.13) 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.

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

FIGURA 4.13: 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.

4.7.3. Comparativa de algoritmos

4.7.3.1. Monohilo

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.

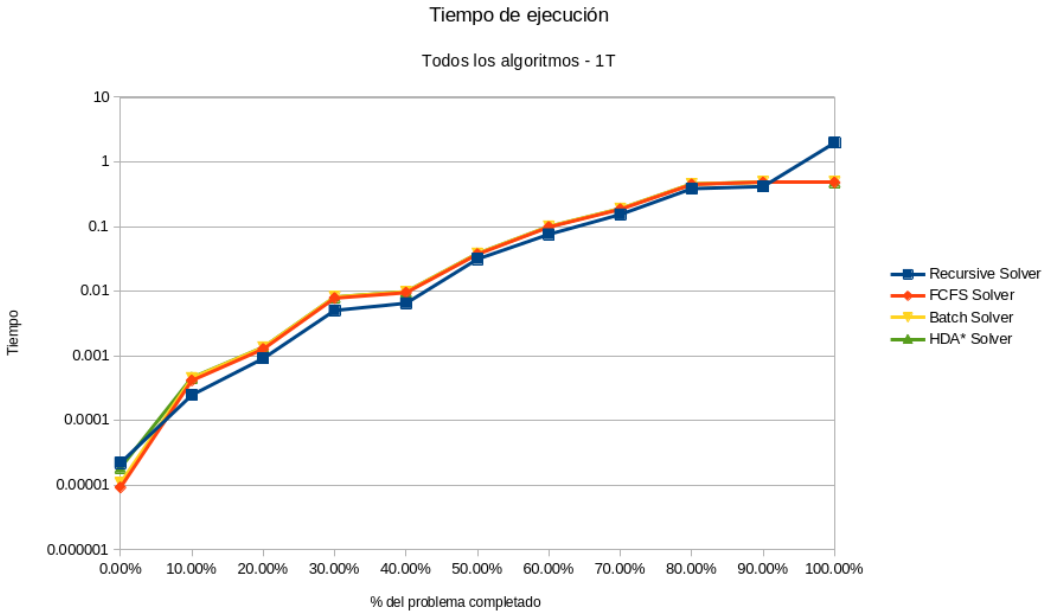
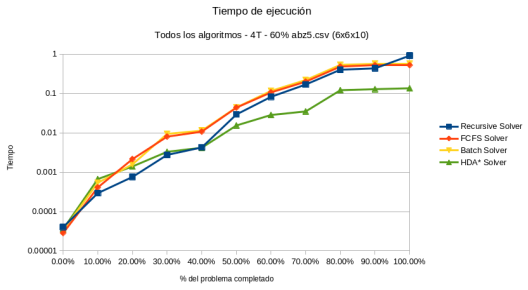
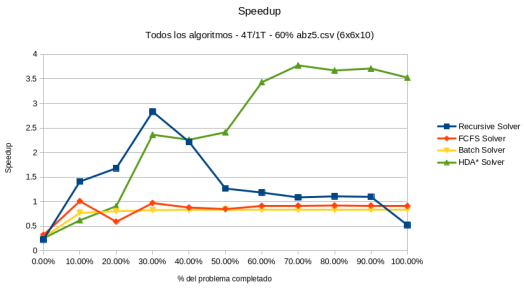


FIGURA 4.14: Tiempo de ejecución de todos los algoritmos (1 hilo).



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



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

FIGURA 4.15: Métricas con 4 hilos.

4.7.3.2. Con sincronización: Batch y Recursive

Se observa el rendimiento de los algoritmos con sincronización entre hilos: Batch y Recursive solver. Al tener sincronización estos algoritmos retornan siempre la misma solución para el mismo problema.

A modo de recordatorio:

- Batch solver: En cada iteración se exploran los mejores nodos conocidos.
- Recursive solver: Se resuelve el algoritmo A* para cada sucesor del nodo inicial.

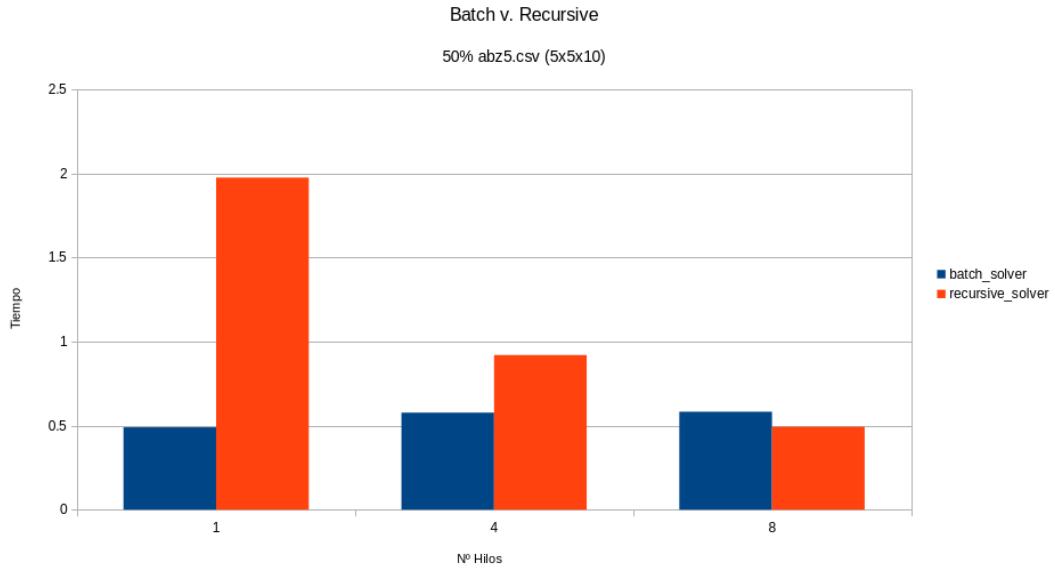


FIGURA 4.16: Tiempo de ejecución del algoritmos Batch y Recursive solvers.

En el caso general, el algoritmo Batch Solver sigue el mismo camino que el A* monohilo. La principal diferencia es que al evaluar varios nodos en cada iteración, puede encontrar un camino diferente al A* monohilo que le lleve al resultado óptimo en menos tiempo. No obstante, se ve obligado a destruir y crear al equipo de hilos en cada iteración. Además, al evaluar más nodos, el `open_set` crece más rápidamente, empeorando el principal cuello de botella del algoritmo. En las pruebas ejecutadas, estos sobrecostes no compensan el posible ahorro de tiempo.

El algoritmo Recursive Solver resolverá el algoritmo A* monohilo V veces siendo V el número de vecinos del nodo inicial. Esto implica que en monohilo, el tiempo se ve multiplicado por V veces. Si se utilizan N hilos pero hay $1{,}5 \times N$ vecinos del nodo inicial, N hilos ejecutarán el algoritmo A* y cuando hayan terminado, $\frac{N}{2}$ volverán a ejecutar otro A* para los nodos vecinos restantes.

Salvo en casos particulares, ninguna de estas estrategias supera el rendimiento del algoritmo A* normal monohilo.

4.7.3.3. Sin sincronización: FCFS y HDA*

Se observa el rendimiento de los algoritmos sin sincronización entre hilos: FCFS y HDA* solver. Al no tener sincronización estos algoritmos pueden retornar diferentes soluciones para el mismo problema.

A modo de recordatorio:

- FCFS: Los hilos exploran nodos tan pronto como estén disponibles en el `open_set` compartido.
- HDA*: Igual que FCFS pero los nodos son asignados al `open_set` privado de un hilo.

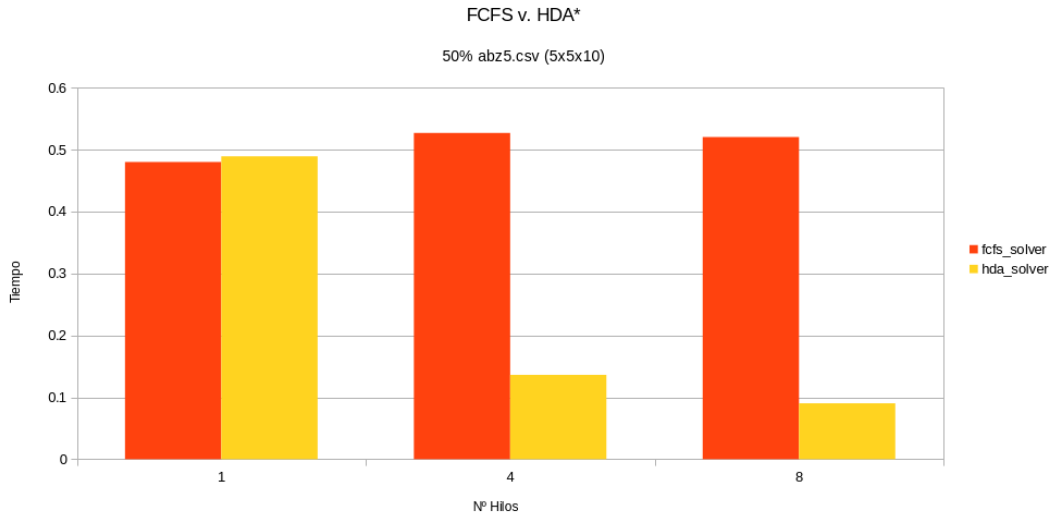


FIGURA 4.17: Tiempo de ejecución del algoritmos FCFS y HDA* solvers.

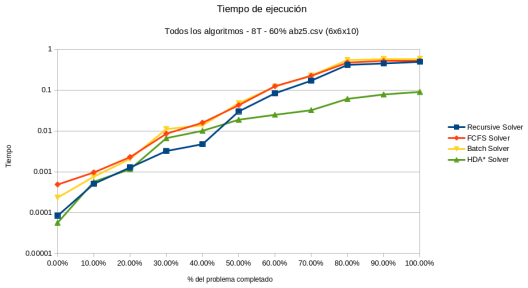
El algoritmo FCFS tiene prácticamente las mismas virtudes y defectos que el Batch solver. Tiene una mayor posibilidad de encontrar el camino óptimo en menos tiempo pero explora más nodos e incrementa el tamaño del `open_set` reduciendo su rendimiento.

El algoritmo HDA* sin embargo parece tener lo mejor de ambos mundos. Por un lado explora una mayor cantidad de nodos por lo que puede encontrar el camino óptimo en menos tiempo pero distribuye los nodos descubiertos en diferentes `open_set`, reduciendo los efectos del cuello de botella. Esto es, la reducción del cuello de botella es directamente proporcional al número de hilos que estén trabajando en el problema [11].

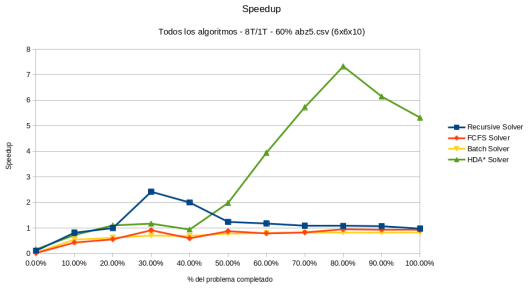
[11] Si el tamaño del problema es lo suficientemente grande.

4.7.3.4. Todos los algoritmos

Al utilizar cuatro hilos (figura 4.15), se puede comenzar a ver una diferencia clara en el rendimiento del algoritmo HDA*, que obtiene un *speedup* de casi 4.



(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.

Al utilizar ocho hilos (figura 4.18), el algoritmo HDA* incrementa 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 el máximo *speedup* posible para este número de hilos (8).

La principal conclusión de estas observaciones es que (al menos en los conjuntos de datos observados), en casos generales, el paralelismo sólo es 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.

4.7.4. FPGA

Se han tomado varias mediciones del algoritmo monohilo corriendo en la FPGA. Los resultados muestran que la implementación de la FPGA es más lenta en todos los casos salvo el Recursive Solver. Esto es comprensible ya que el Recursive Solver debe ejecutar varias veces el algoritmo cuando el número de hilos es inferior al número de nodos vecinos del nodo inicial.

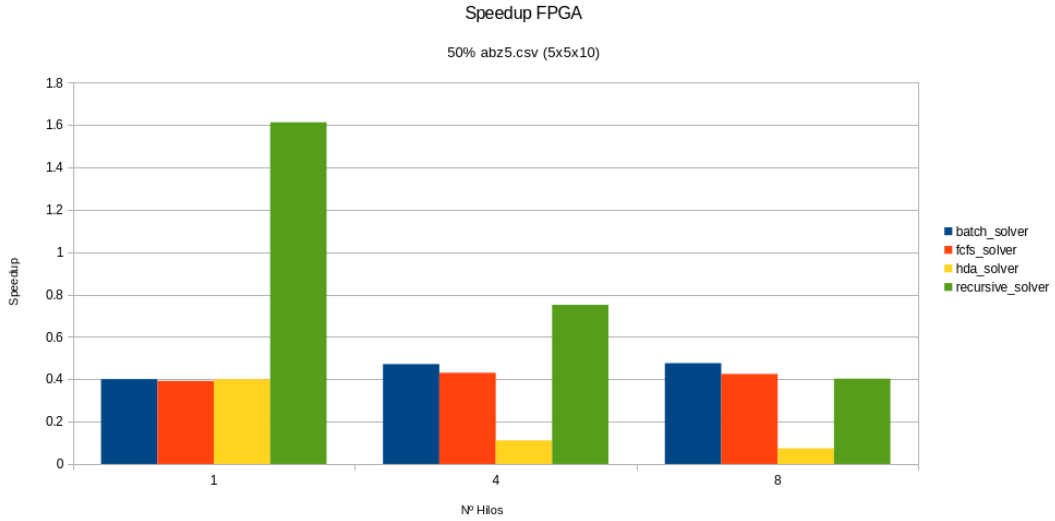


FIGURA 4.19: *Speedup* de la FPGA frente al resto de algoritmos.

La gráfica 4.19 muestra el *speedup* del algoritmo en la FPGA respecto al resto de implementaciones monohilo y paralelas. Se ha observado también el comportamiento de este algoritmo a lo largo de diferentes tamaños de problema con el objetivo de hallar el máximo tamaño que se puede resolver en un tiempo razonable.

Los resultados de *speedup* de todas estas mediciones son similares a la gráfica anterior por lo que se han omitido. Las ejecuciones en CPU han sido capaces de resolver en un tiempo razonable problemas de tamaño 6x6x10 generalmente en menos de una hora, los problemas de 7x7x10 tardan varias horas o días en función de cómo de fácil sea hallar el camino solución. La FPGA por otro lado ha sido capaz de resolver problemas de tamaño 4x4x10 con relativa facilidad en menos de una hora pero se estanca en algunos problemas de tamaño 5x5x10.

FUNDAMENTAL

Con la síntesis hecha en esta investigación, el algoritmo en FPGA tarda en resolver un problema de tamaño 4x4x10 lo mismo que una CPU tarda en resolver un problema 6x6x10.

Lo más seguro es que la FPGA sea capaz de superar en algún caso el rendimiento en CPU dados el algoritmo, implementación y problema adecuados. Esto no ha sucedido muy probablemente por falta de conocimiento y práctica a la hora de escribir código transpilable con HLS.

4.7.5. 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.

4.7.5.1. 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.

4.7.5.2. 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 [12]. 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.

[12] 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. Conclusiones

5.1.1. 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.

5.1.2. 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 reducir el tiempo consumido por el `open_set` sea determinar cómo almacenar menos nodos en ella.

5.1.3. Heurísticos

Como ya se discutió en la sección comparativa con Dijkstra (4.7.2.3), 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.

5.1.4. FPGA

Es indudable que la particular arquitectura de una FPGA da la oportunidad de acelerar la ejecución de algunos algoritmos. Como se ha observado en el caso de algoritmo A*, la implementación producida por HLS no ha sido capaz de superar el rendimiento de las implementaciones paralelas en CPU cuidadosamente diseñadas a mano.

No obstante, lo más probable es que un diseño de un programa VHDL implementado con el mismo cuidado con el que se han diseñado los algoritmos paralelos de este documento sea capaz de superar el rendimiento del algoritmo en CPU.

5.1.5. Paralelismo

Puede parecer que el problema JSP requiere una gran capacidad de cálculo para ser resuelto. En realidad, la mayoría de datos relevantes para resolver el problema se pueden precalcular antes de la ejecución del problema (funciones hash, costes H, costes G), reduciendo notablemente la complejidad de obtener los vecinos de un nodo. El principal efecto de este 'atajo' es que la complejidad del algoritmo deja de encontrarse en el cálculo y se relocaliza en simples movimientos de estados entre el `open_set` y estructuras de costes. Esto no sucedería si se utilizase un acercamiento funcional donde todos los calculos sería de tipo *lazy* y no sería posible cachear ningún valor.

El paralelismo, tanto en CPU, como GPGPU y FPGA es de particular utilidad cuando es necesario resolver problemas cuyos datos son procesados por algún tipo de operación matemática Como ya se indicó en el apartado 4.3, el uso de la FPGA requiere tres pasos:

1. Leer un valor de un registro.
2. Ejecutar una operación sobre el valor.
3. Almacenar el resultado en otro registro.

Al aplicar los 'atajos' descritos en el párrafo anterior, la mayoría de operaciones matemáticas son irrelevantes porque sus resultados ya han sido obtenidos. Esto elimina el segundo paso, reduciendo notablemente los beneficios de cualquier herramienta que paralelice el algoritmo.

5.2. Trabajos futuros

Este proyecto ha presentado el sistema modular que forma el algoritmo A* y para cada módulo se han implementado diferentes alternativas. Comprobar empíricamente el funcionamiento de todas las combinaciones posibles es una tarea tediosa, sólo la ejecución de las pruebas puede llevar fácilmente una semana de espera. Describir todas las combinaciones en un documento con una longitud razonable es simplemente imposible.

Una de las conclusiones de esta investigación es la importancia del diseño de la función heurística, que supera la del paralelismo. Primero, sería adecuado realizar un estudio sobre una cantidad de funciones heurísticas distintas para distinguir una implementación óptima.

Segundo, sería interesante desarrollar una solución del algoritmo A* para un problema que tenga un mayor peso en el cálculo, que requiera un mayor número de operaciones para explorar un nodo. Esto se puede conseguir de diferentes formas:

- Eliminando el precálculo de datos.
- Desarrollando una implementación con un acercamiento funcional.
- Cambiando el problema JSP por otro.

Tercero, el Recursive Solver no obtiene resultados muy interesantes en esta investigación, donde el algoritmo ejecuta varias veces el A* monohilo. Es posible que si en lugar de ejecutar el A* monohilo se utiliza uno de los algoritmos más rápidos sin sincronización (como el HDA*), el Recursive Solver sea capaz de obtener resultados en menos tiempo y con mejor calidad que si se ejecutase el HDA* sólo una vez. De cualquier forma, este estudio requeriría un sistema con un mayor número de núcleos, como mínimo 16 para ejecutar el HDA* con 4 hilos de forma simultánea 4 veces [2].

Finalmente, esta investigación ha dado prioridad al rendimiento del programa medido en función de su tiempo de ejecución, ignorando por completo los efectos en consumo de memoria RAM. Sería de interés desarrollar un estudio sobre diferentes implementaciones del algoritmo A* atendiendo al consumo de memoria. Este estudio estaría relacionado con el segundo ya que para reducir el consumo de memoria será necesario eliminar el precálculo o implementar una versión funcional.

[2] Preferiblemente 24 para ejecutar HDA* con 4 hilos 6 veces o 32 para ejecutarlo 8 veces

5.3. Crítica retrospectiva

A lo largo de esta investigación se han tomado decisiones que han facilitado profundamente el desarrollo del trabajo. Asimismo, otras decisiones han implicado barreras que dificultaron la investigación.

Ha sido un acierto comenzar un prototipo en **Python** , ya que aunque esta versión no ha sido observada en los resultados debido a su bajo rendimiento, ha supuesto una base sobre la que implementar rápidamente el algoritmo el A*.

La mayoría del tiempo de desarrollo de este proyecto se ha invertido en C++, un lenguaje orientado a objetos similar a Java pero con mucho mayor rendimiento y la posibilidad de micro-optimización que ofrecería C. Esta decisión ha facilitado notablemente el desarrollo del algoritmo en un entorno que permita el paralelismo con OpenMP.

La principal dificultad propuesta por C++ ha sido la síntesis del código a VHDL, que no admitía varias clases básicas para la ejecución del algoritmo. Tal vez hubiese sido adecuado continuar desarrollando una versión del A* en C que se abandonó poco después de iniciar el proyecto debido a la falta de una librería estandar que facilite estructuras de datos básicas. Incluso usando C++, se podría haber dependido en mayor medida de punteros y estructuras que de objetos.

Finalmente, las métricas han tenido en cuenta principalmente el *makespan* y el tiempo de ejecución. Una vez conocido el principal cuello de botella que supone el `open_set`, habría sido de utilidad medir su tamaño ya que esta métrica está estrechamente relacionada con el tiempo de ejecución. La bibliografía de este documento tiene en cuenta otros datos como el consumo energético, que hubiese sido de interés para esta investigación.

5.4. Tecnologías utilizadas

1. Visual Studio Code: Editor principal: Programas, memoria y presentación.
2. Vim: Editor principal: Programas, memoria y presentación.
3. Xilinx Vitis HLS (Unified): Transpilador para FPGA.
4. Xilinx Vivado: Editor para FPGA.
5. TypeScript: Lenguaje de presentación.
6. ReactJS: Componentes de presentación.
7. Motion Canvas: Librería de presentación.
8. Python: Lenguaje de prototipado.
9. C++: Lenguaje del programa principal y FPGA.
10. C: Lenguaje del programa FPGA (Versión descartada).
11. OpenMP: Librería paralelismo CPU.
12. LaTeX: Lenguaje de marcas de la memoria[3].
13. BASH: Lenguaje scripting para ejecución de pruebas.
14. SSH: Acceso remoto a servidor para ejecución de pruebas.
15. GIT: Control de versiones.
16. GDB: Depurador C/C++.
17. GPROF: Profiler C/C++.
18. Address Sanitizer (ASAN): Depurador de fugas de memoria y violaciones de segmento.
19. Valgrind: Depurador de fugas de memoria y violaciones de segmento.
20. Libreoffice Calc: Hoja de cálculo y gráficas.

[3]No se listan el gran número de librerías TeX utilizadas en este proyecto.

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

Diagramas

B.1. Diagrama Pert

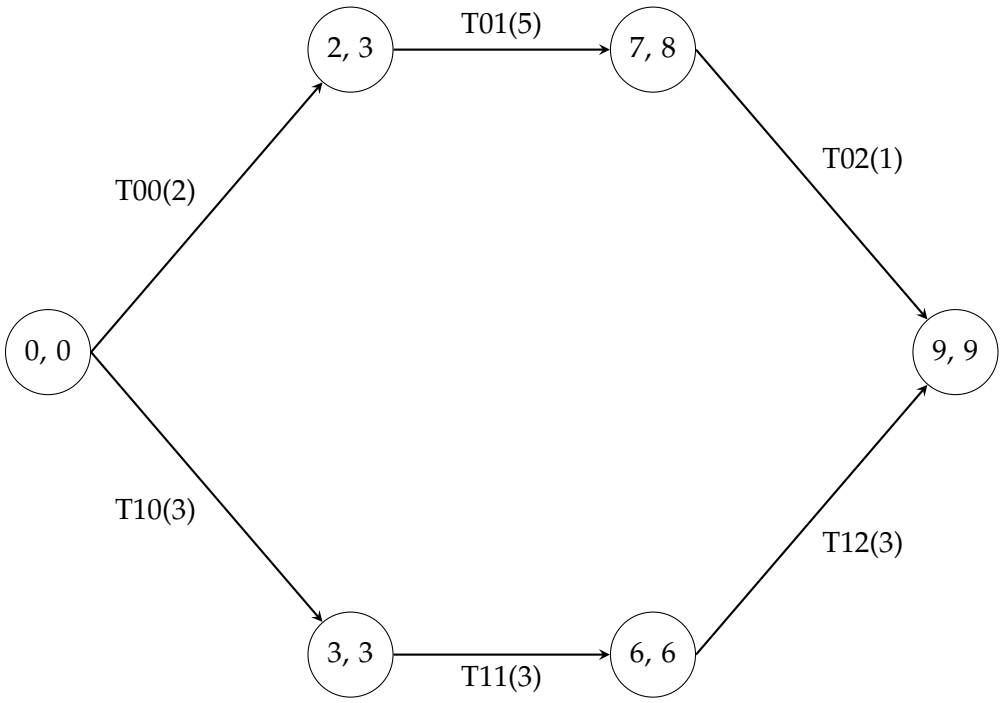


FIGURA B.1: Diagrama Pert ejemplo.

B.2. Diagrama Gantt

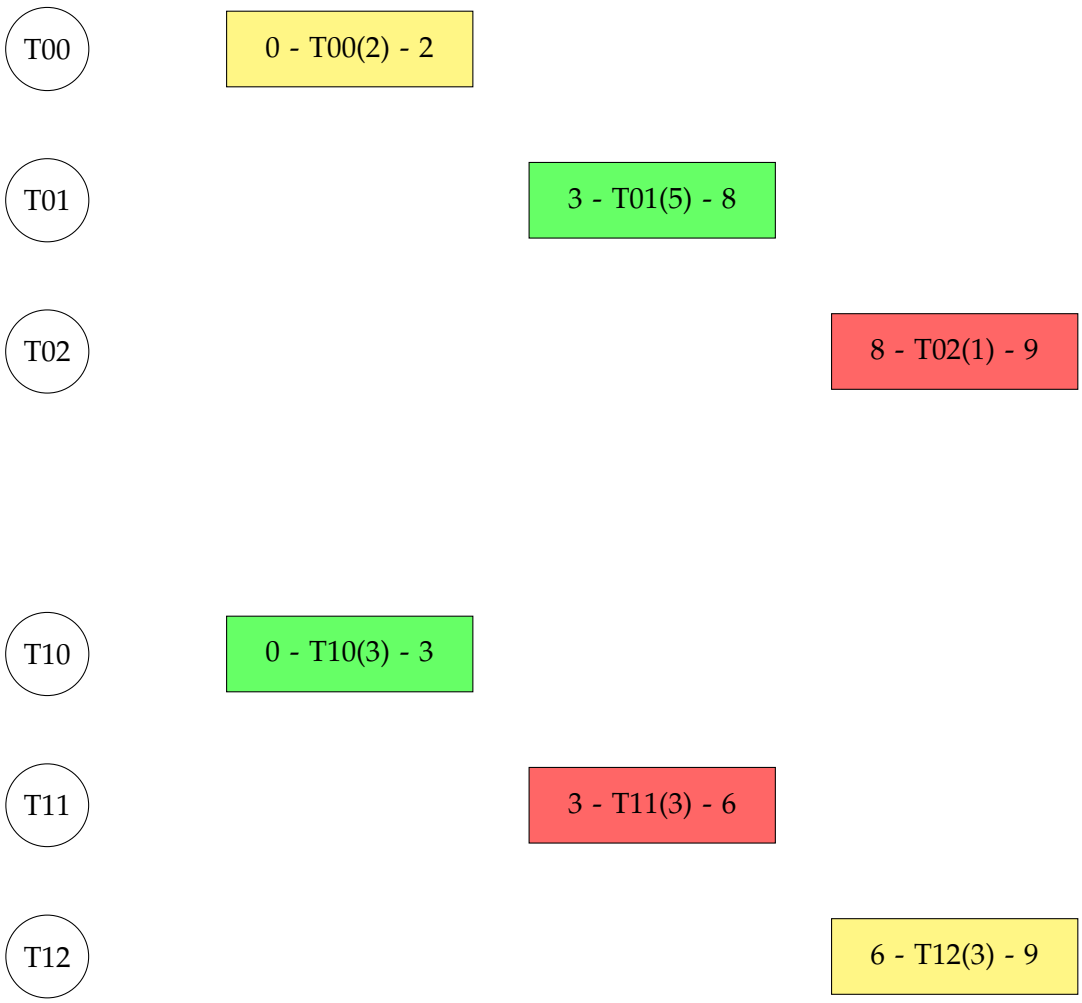


FIGURA B.2: Diagrama Gantt ejemplo.

B.3. Algoritmo A*

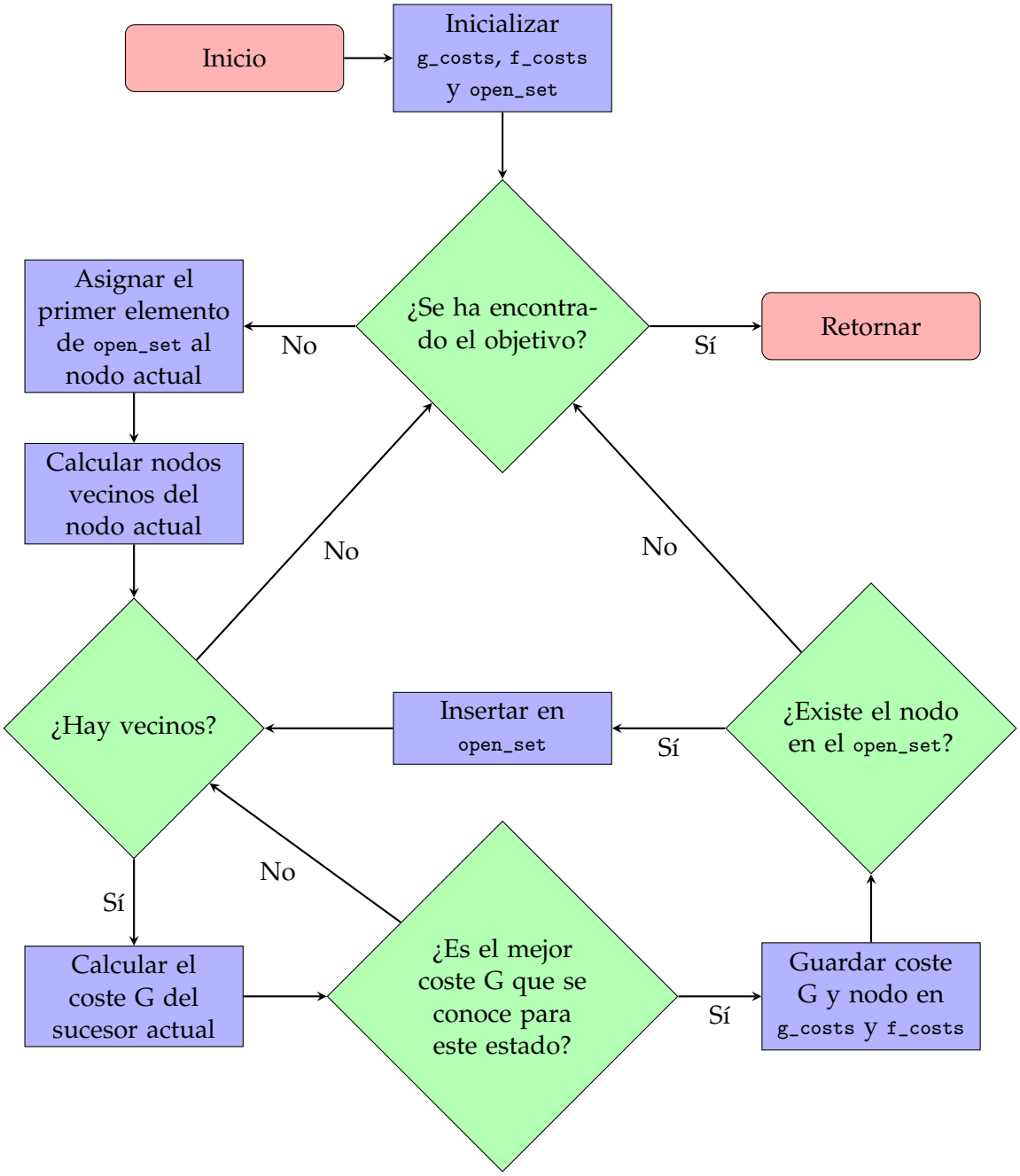


FIGURA B.3: Representación del algoritmo A*

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