UNIVERSIDAD DE OVIEDO
ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN

TRABAJO DE FIN DE GRADO

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

Autor:
Alejandro Rodríguez López

Tutor:
Juan José Palacios Alonso

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

5 de junio de 2024

UNIVERSIDAD DE OVIEDO

Resumen

Escuela Politécnica de Ingeniería de Gijón
Informática

Grado de Ingeniería Informática en Tecnologías de la Información

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

por Alejandro Rodríguez López

Índice general

Resumen I

1. Hipótesis de Partida y Alcance: *Estado del Arte* **1**
1.1. Objeto de la investigación . . . 1
1.2. Estado del arte . . . 1
1.3. Requisitos . . . 2
1.4. Alcance . . . 2

2. Problema a investigar: *Análisis, implementación y optimización* **3**
2.1. Job Shop Scheduling Problem . . . 3
2.2. Método a resolver . . . 4
2.2.1. Algoritmo . . . 4
2.2.1.1. Componentes . . . 4
2.2.1.2. Pseudocódigo . . . 6
2.2.2. Equipo de Estudio . . . 6
2.2.2.1. Arquitectura x86 . . . 7
2.2.2.2. FPGA . . . 7
2.3. Método de comparativas . . . 7
2.4. Implementación . . . 8
2.4.1. Task . . . 8
2.4.2. State . . . 8
2.5. Optimización . . . 9
2.5.1. State . . . 9
2.5.2. A* . . . 10
2.5.3. Paralelización . . . 10
2.5.3.1. Secciones críticas . . . 10

3. Experimentos: *Trabajo y Resultados* **14**
3.1. Conjuntos de datos . . . 14
3.1.1. *Jobshop Instances* . . . 14
3.1.2. Personalizados . . . 14
3.2. Método de medición . . . 14
3.2.1. Python . . . 15
3.2.2. C++ . . . 15
3.2.3. FPGA . . . 16
3.3. Resultados . . . 16
3.4. Análisis . . . 16

4. Conclusiones: *Observaciones y Trabajos futuros* **17**

Bibliografía **18**

Índice de figuras

2.1. Comparativa entre algoritmos Dijkstra y A* 4
2.2. Representación del algoritmo A* 11
2.3. Comparativa entre algoritmos monohilo y multihilo 12
2.4. Representación de la sección paralela del algoritmo 13

Índice de cuadros

Lista de Abreviaturas

Capítulo 1

Hipótesis de Partida y Alcance

Estado del Arte

1.1. Objeto de la investigación

El presente proyecto tiene como objetivo principal descubrir los beneficios de paralelizar una implementación del algoritmo A* diseñado para resolver el Job Shop Scheduling Problem. Adicionalmente, se observará el rendimiento de una implementación de la misma solución en una FPGA.

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

Respecto al Job Shop Scheduling Problem, presenta un problema establecido años atrás, profundamente estudiado y cuyas soluciones puden ser obtenidas aplicando diversos algoritmos.

Un algoritmo capaz de recibir como entrada las descripciones de una plantilla de trabajadores y un listado de trabajos y tareas a realizar puede ser aplicable en ámbitos industriales donde la automatización de la creación de planificaciones pueda ser de interés.

1.2. Estado del arte

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

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 capaces de resolverlo [Man67].

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

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

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

1.3. Requisitos

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

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

1.4. Alcance

El presente documento describe el problema a resolver en detalle, los métodos utilizados para resolverlo y los equipos en los que se ejecutan las diferentes versiones de las soluciones.

Entre los contenidos de este documento se encuentran detalles sobre la implementación de las soluciones, así como los cuellos de botella hallados, problemas y razonamientos a la hora de tomar ciertas decisiones en el desarrollo así como estudios teóricos realizados con el objetivo de hallar inconvenientes de forma preventiva.

Capítulo 2

Problema a investigar

Análisis, implementación y optimización

2.1. Job Shop Scheduling Problem

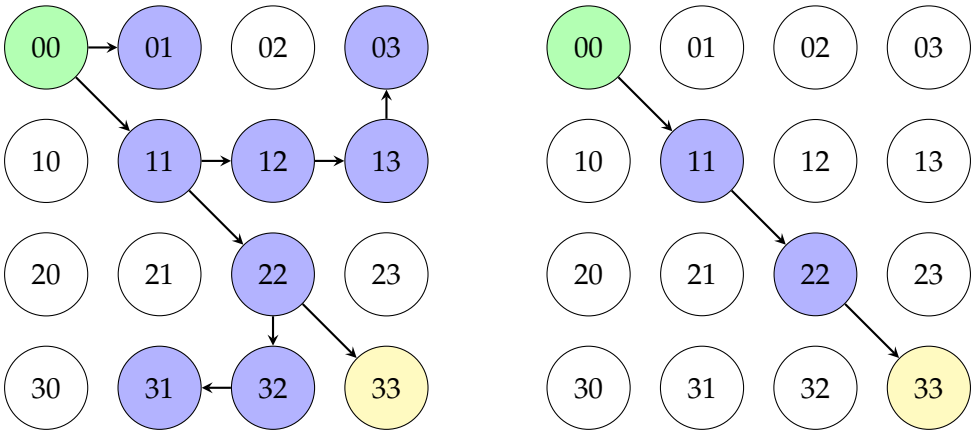
Se estudiará la implementación de una solución al problema *Job Shop Scheduling (JSP)* [Yan77] [1]. Como su nombre indica, se trata de un problema en el que se debe crear una planificación. Desde el punto de vista de la ingeniería informática, el JSP es un problema de optimización.

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

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

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



(A) Algoritmo Dijkstra (B) Algoritmo A*

FIGURA 2.1: Comparativa entre algoritmos Dijkstra y A*

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

2.2. Método a resolver

2.2.1. Algoritmo

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

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

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

2.2.1.1. Componentes

2.2.1.1.1. Estado

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

- Instante de tiempo en el que comienza cada tarea planificada.
- Instante de tiempo futuro en el que cada trabajador estará libre.

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

2.2.1.1.2. Costes

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

2.2.1.1.2.1. Coste G

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

2.2.1.1.2.2. Coste H

El coste H (de ahora en adelante $cost_h$) es el coste estimado desde el estado actual hasta el estado final. Este coste es calculado utilizando una función heurística, que estima el coste. Esta función debe no debe ser optimista, por lo que en este estudio se utilizará el sumatorio de duraciones de las tareas restantes por planificar.

2.2.1.1.2.3. Coste F

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

Por lo tanto,

$$cost_f = cost_g + cost_h$$

2.2.1.1.3. Generación de sucesores

El algoritmo A* debe generar un número de estados sucesores dado un estado actual. Por lo que será necesario una función que dado un estado retorne un listado de estados.

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

2.2.1.1.4. Listas de prioridad

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

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

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

2.2.1.2. Pseudocódigo

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

2.2.2. Equipo de Estudio

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

2.2.2.1. Arquitectura x86

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

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

Una vez desarrolladas, se realizan comparativas entre las distintas implementaciones de esta aquitectura, en particular:

1. Comparativa entre el rendimiento monohilo de Python y C++.
2. Comparativa entre el rendimiento monohilo de C++ y multihilo de C++.

2.2.2.2. FPGA

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

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

2.3. Método de comparativas

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

1. Tiempo de ejecución.
2. Consumo de memoria volátil.

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 dos veces ignorando las métricas de la primera ejecución.

2.4. Implementación

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

2.4.1. Task

La clase `Task` 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.

2.4.2. State

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

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

NOTA

Nótese que el atributo `std::vector<std::vector<Task>> jobs` será el mismo en todos los estados de un mismo problema. Por lo que no será necesario revisarlo en `State::operator==` ni `State::operator()`.

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 utilice para distinguir diferentes instancias de `State` (`FullHash::operator()`).

2.5. Optimización

2.5.1. 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 para evitar tener que recalcularlo.

La operación `operator()` contiene 2 bucles `for` anidados. Su principal objetivo es calcular una reducción de los atributos de la instancia `State`. Se utiliza `#pragma omp parallel for collapse(2) reduction(+: seed)` para paralelizar la reducción.

```
1  std::size_t FullHash::operator()(State key) const
2  {
3      if (key.get_full_hash() != UNINITIALIZED_HASH)
4          return key.get_full_hash();
5      std::vector<std::vector<int>>
6          schedule = key.get_schedule();
7      std::vector<int> workers_status = key.get_workers_status();
8      std::size_t seed = schedule.size() * schedule[0].size() *
       workers_status.size();
9
10 #pragma omp parallel for reduction(+ : seed)
11     for (size_t i = 0; i < workers_status.size(); i++)
12         seed += (workers_status[i] * 10 ^ i);
13
14     if (schedule.empty())
15         return seed;
16
17     const std::size_t nTasks = schedule[0].size();
18 #pragma omp parallel for collapse(2) reduction(+ : seed)
19     for (size_t i = 0; i < schedule.size(); i++)
20     {
21         for (size_t j = 0; j < nTasks; j++)
22             seed += (schedule[i][j] * 10 ^ ((1 + i + workers_status.
       size()) * j + j));
23     }
24     key.set_full_hash(seed);
25     return seed;
```

```
26 }
```

LISTING 2.1: Implementación de `FullHash::operator()`

Los operadores `operator==` necesarios se implementan utilizando los `operator()` correspondientes.

NOTA

Las funciones hash utilizadas en los `operator()` son resistentes a colisiones, esto es, $(\nexists(State_a, State_b)/h(State_a) = h(State_b)) \rightarrow (h(State_a) \neq h(State_b) \iff State_a \neq State_b)$ por lo que se pueden utilizar para comparar elementos en `operator==`.

2.5.2. A*

A la hora de paralelizar un algoritmo existen dos principales direcciones en las que basar el diseño:

- Paralelización de datos: N datos independientes sufren el mismo procesamiento de forma paralela.
- Paralelización de tareas: N tareas independientes son ejecutadas de forma paralela.

Visto el diagrama (2.2), está claro que las tareas del algoritmo A* no son paralelizables y el número de datos (nodos) varía según se ejecuta el algoritmo. Ignorando las posibles oportunidades de paralelismo que se puedan encontrar en los procesos individuales del algoritmo, la principal paralelización se encuentra en el procesamiento de los nodos.

2.5.3. Paralelización

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.

De cualquier forma, este diseño es particular no tiene por qué reducir el tiempo requerido para hallar un nodo solución, simplemente tiene la oportunidad de reducirlo en algunos casos específicos. Porque se explora un mayor número de nodos en el mismo tiempo. (Véase 2.3)

2.5.3.1. 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.

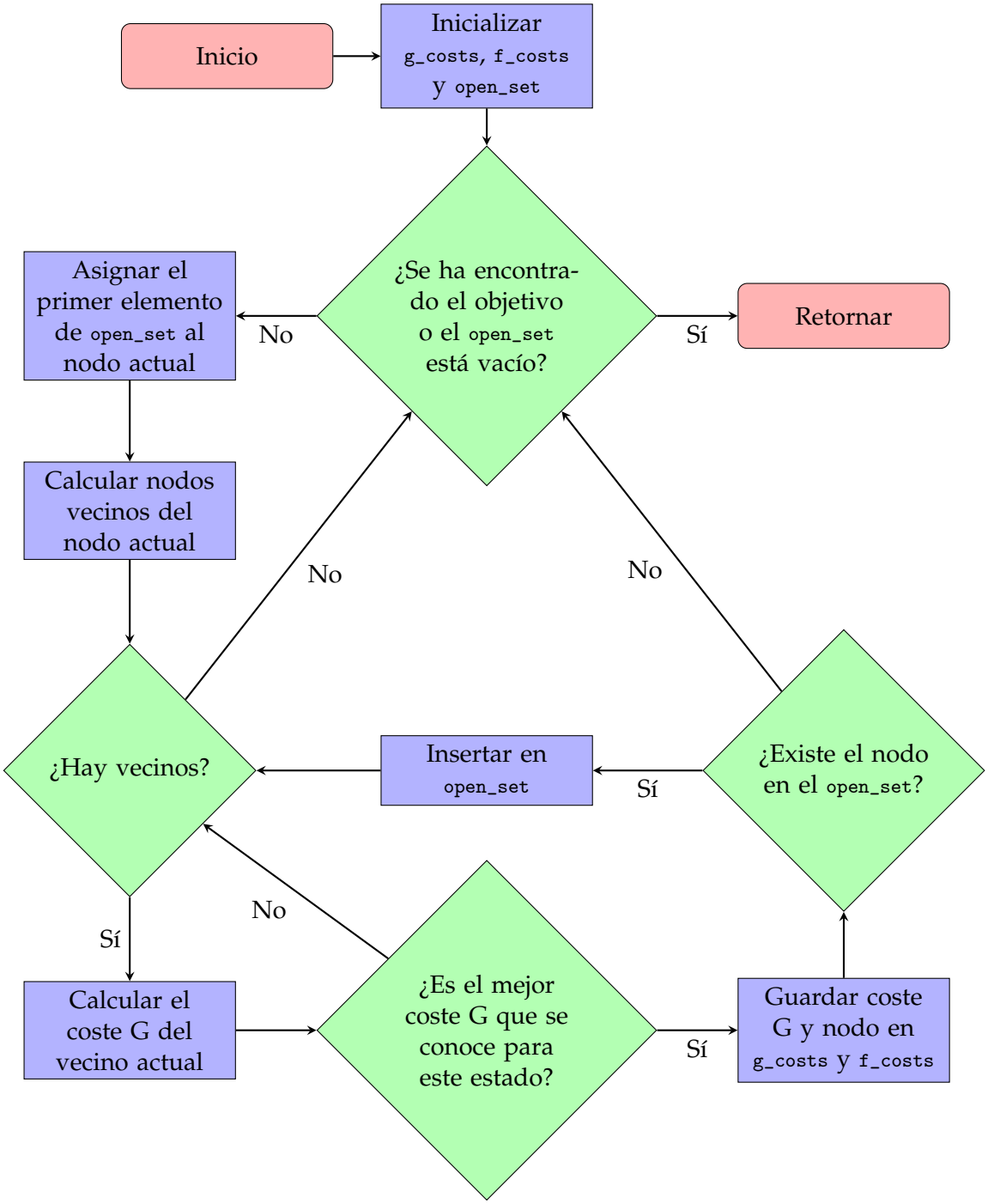
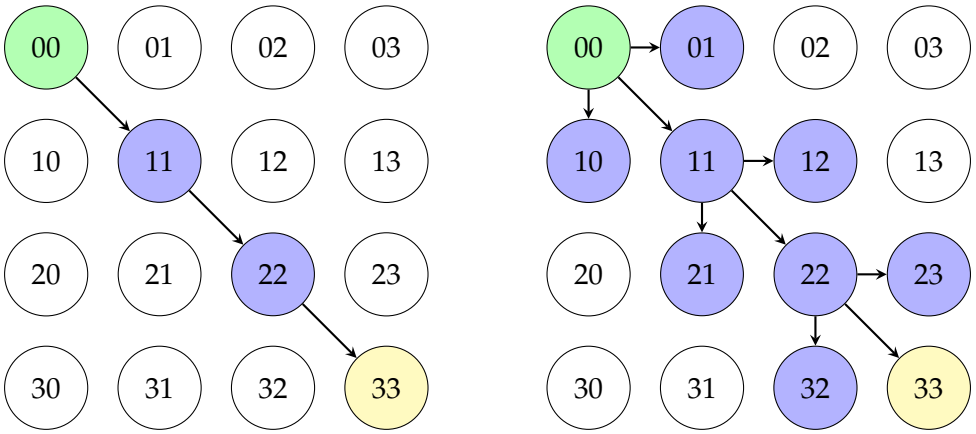


FIGURA 2.2: Representación del algoritmo A*



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

FIGURA 2.3: Comparativa entre algoritmos monohilo y multihilo

2.5.3.1.1. 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.

2.5.3.1.2. `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 $O(n)$ [2].

NOTA

La sección crítica correspondiente al `open_set` tiene una complejidad proporcional al tamaño del mismo. 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.

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

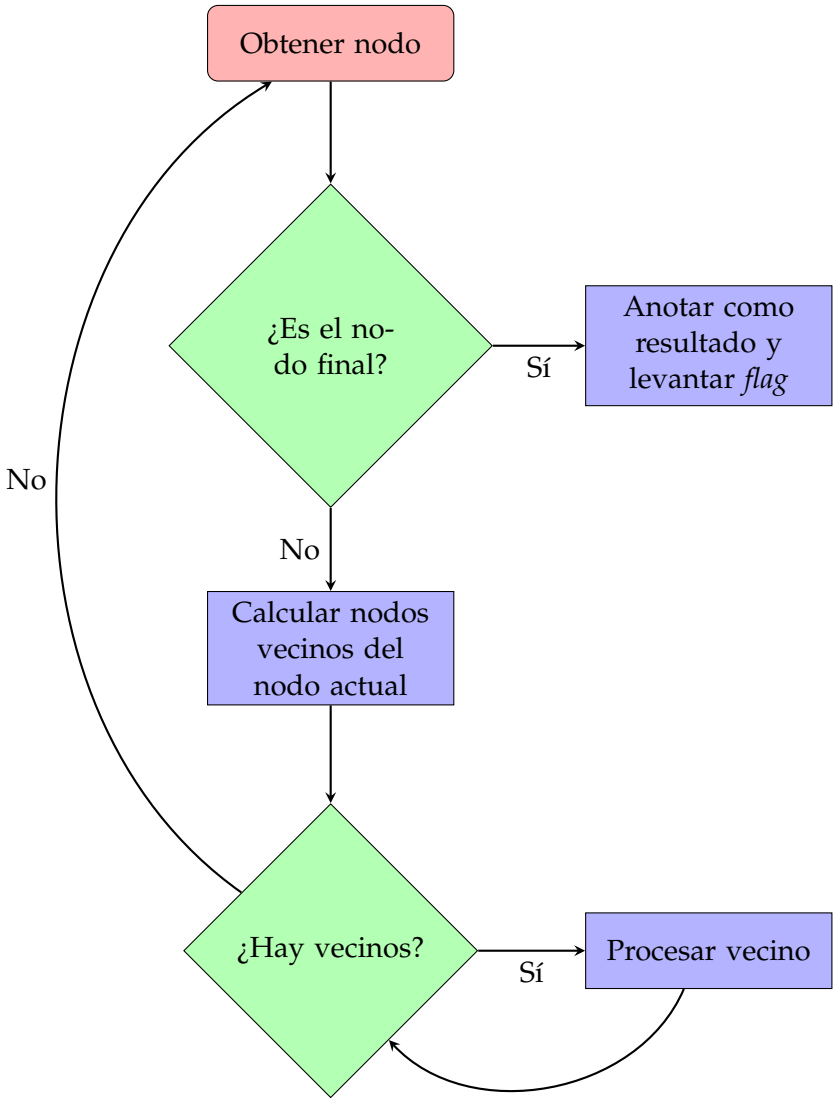


FIGURA 2.4: Representación de la sección paralela del algoritmo

Capítulo 3

Experimentos

Trabajo y Resultados

3.1. Conjuntos de datos

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

3.1.1. *Jobshop Instances*

3.1.2. Personalizados

3.2. Método de medición

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

- Lenguaje
- Número de hilos
- Trabajos
- Tareas
- Trabajadores
- Tiempo de ejecución
- Planificación
- *Makespan*

3.2.1. Python

Se utiliza un *Wrapper* de python que se encarga de tomar una medición de tiempo antes y después de iniciar el algoritmo.

```python
def timeit(func):
    @wraps(func)
    def timeit_wrapper(*args, **kwargs):
        n_jobs: int = len(args[0])
        n_tasks: int = len(args[0][0])
        n_workers: int = args[1]
        func(*args, **kwargs)  # Cache warm up
        start_time = time.perf_counter()
        result: State = func(*args, **kwargs)
        end_time = time.perf_counter()
        total_time = end_time - start_time
        print(
            (f"py;1;{func.__name__};{args};{n_jobs};{n_tasks};"
             f"{n_workers};{total_time:1.5E};{result.schedule};"
             f"{max(result.workers_status)}")
        )
        return result
    return timeit_wrapper
```

3.2.2. C++

Se utiliza una función que se encarga de tomar una medición de tiempo antes y después de iniciar el algoritmo.

```cpp
State timeit(
    const std::function<State(std::vector<std::vector<Task>>, int)> &
    foo,
    const std::vector<std::vector<Task>> &jobs, int n_workers)
{
    foo(jobs, n_workers); // Cache warm up
    auto start_time = std::chrono::high_resolution_clock::now();
    const State result = foo(jobs, n_workers);
    auto end_time = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> total_time = end_time - start_time;
    std::cout << "c++;" << omp_get_max_threads() << ";a_star" << ";("
    << jobs << ", "
              << n_workers << ");" << jobs.size() << ";" << jobs[0].
    size() << ";"
              << n_workers << ";" << std::setprecision(5) << std::
    scientific
              << total_time.count() << ";" << result << ";" << result.
    get_max_worker_status() << std::endl;
    return result;
}
```

3.2.3. FPGA

3.3. Resultados

3.4. Análisis

Capítulo 4

Conclusiones

Observaciones y Trabajos futuros

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.

[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.

[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.