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

4 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

Índice de figuras

Índice de cuadros

Lista de Abreviaturas

Capítulo 1

Hipótesis de Partida y Alcance

Estado del Arte

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).

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 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 problema 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 con mayor rendimiento 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.

2.2.3. 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 varios datos de entrada distintos múltiples veces. Es de crucial importancia tener en cuenta la caché de los dispositivos, esto implica que la primera ejecución de cada conjunto de datos de entrada tiene un peor rendimiento que las siguientes. Para aislar este hecho, las métricas de la primera ejecución de cada conjunto de datos son ignoradas.

2.4. Implementación

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 necesitarán que se proporcionen implementaciones para los operadores `State::operator==` y `State::operator()` de la clase `State`.

- El atributo `std::vector<std::vector<Task>> jobs` es igual en todas las instancias de `State`, por lo que será ignorado.
- El atributo `std::vector<std::vector<int>> schedule` denota la distancia restante hasta el fin del problema.
- El atributo `std::vector<int> workers_status` no tiene valor alguno a la hora de denotar la distancia restante hasta resolver el problema, pero es necesario para distinguir dos estados diferentes.

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 metodo tiene una complejidad de $O(n^2)$, por lo que su valor se almacena en un atributo una vez calculado por primera vez para evitar tener que recalcularlo.

La operación `operator()` consta de 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  }
```

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

NOTA

Las funciones hash utilizadas en los `operator()` son resistentes a colisiones, esto es, $\nexists(a,b)/h(a) = h(b)$ 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 anterior, 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 puedan tener los procesos individuales del algoritmo, la principal paralelización se encuentra en el procesamiento de cada nodo.

Un hilo principal se encarga de comprobar en cada iteración si se ha encontrado el nodo objetivo o si el `open_set` está vacío, mientras que el resto de hilos son asignados un nodo para procesar.

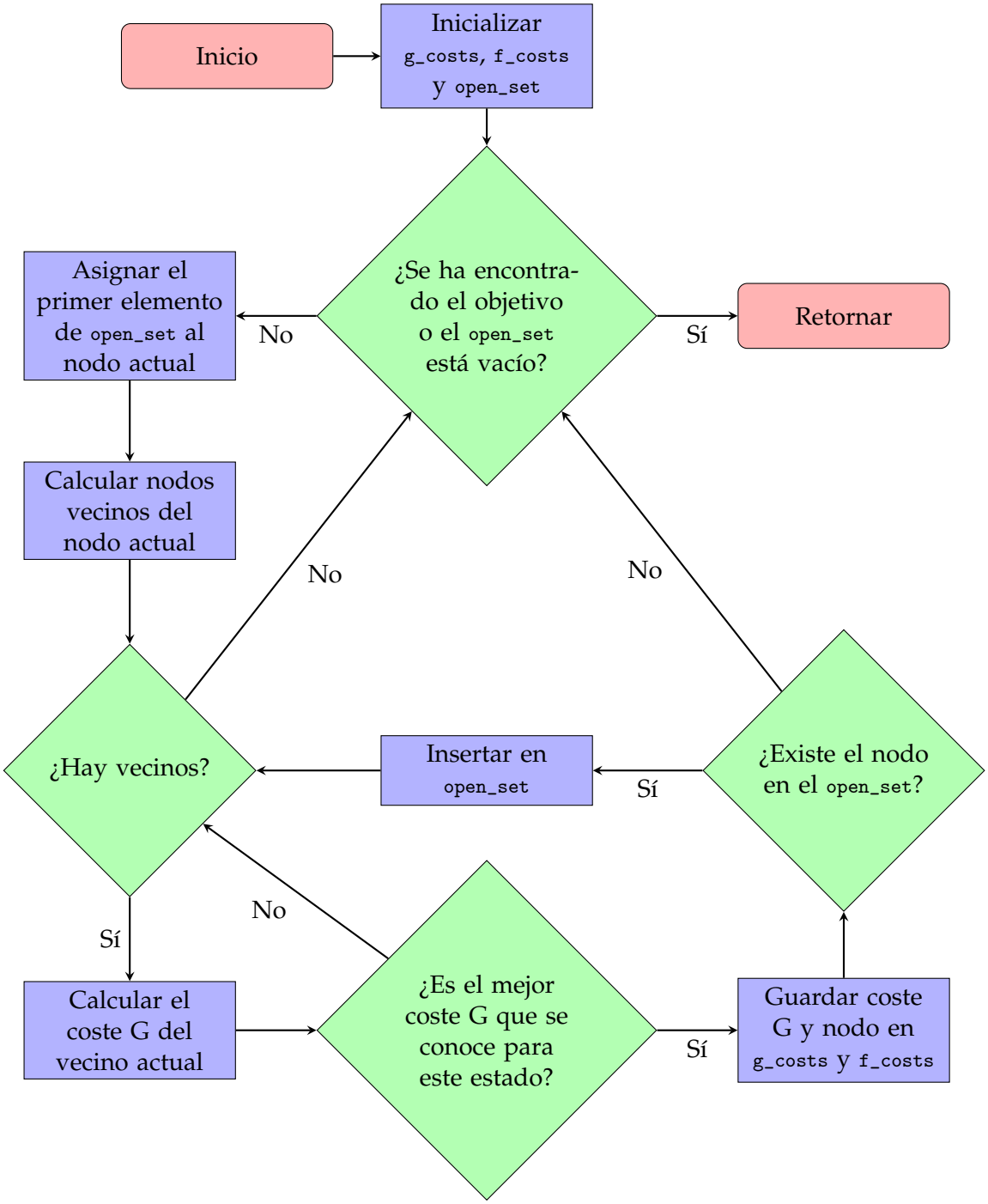
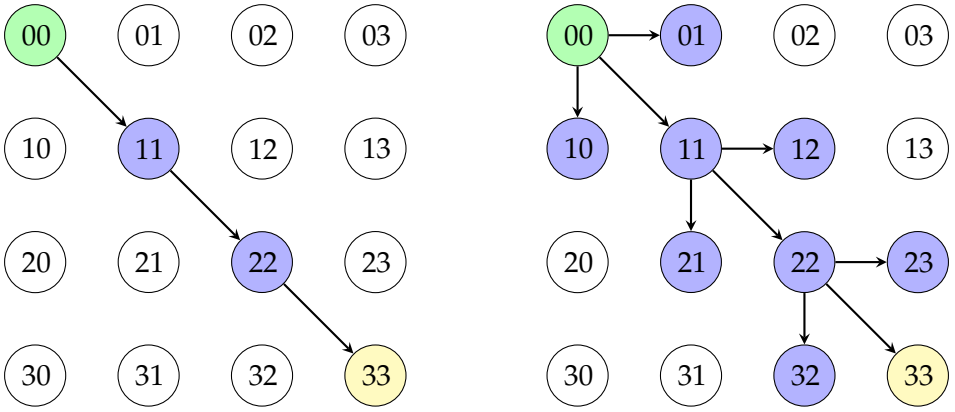


FIGURA 2.1: Representación del algoritmo A*

De cualquier forma, este diseño no tiene por qué necesariamente 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 Figura 2.2)



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

FIGURA 2.2: Comparativa entre algoritmos monohilo y multihilo

El diseño paralelo propuesto no es sin inconvenientes, tiene varias instancias de secciones críticas que suponen una amenaza para el rendimiento del algoritmo.

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. Sería necesario que dos hilos hallasen dos soluciones diferentes al problema al mismo tiempo.

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, es necesario que estos nuevos nodos sean insertados en orden, resultando en una complejidad $O(n)$ [2].

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

NOTA

La sección crítica correspondiente al `open_set` tiene una complejidad proporcional al tamaño del `open_set`. Esto es, si el `open_set` tiene pocos elementos, la sección crítica tendrá pocos efectos en el rendimiento del programa, pero si contiene un número mayor de elementos, será necesario más tiempo 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.

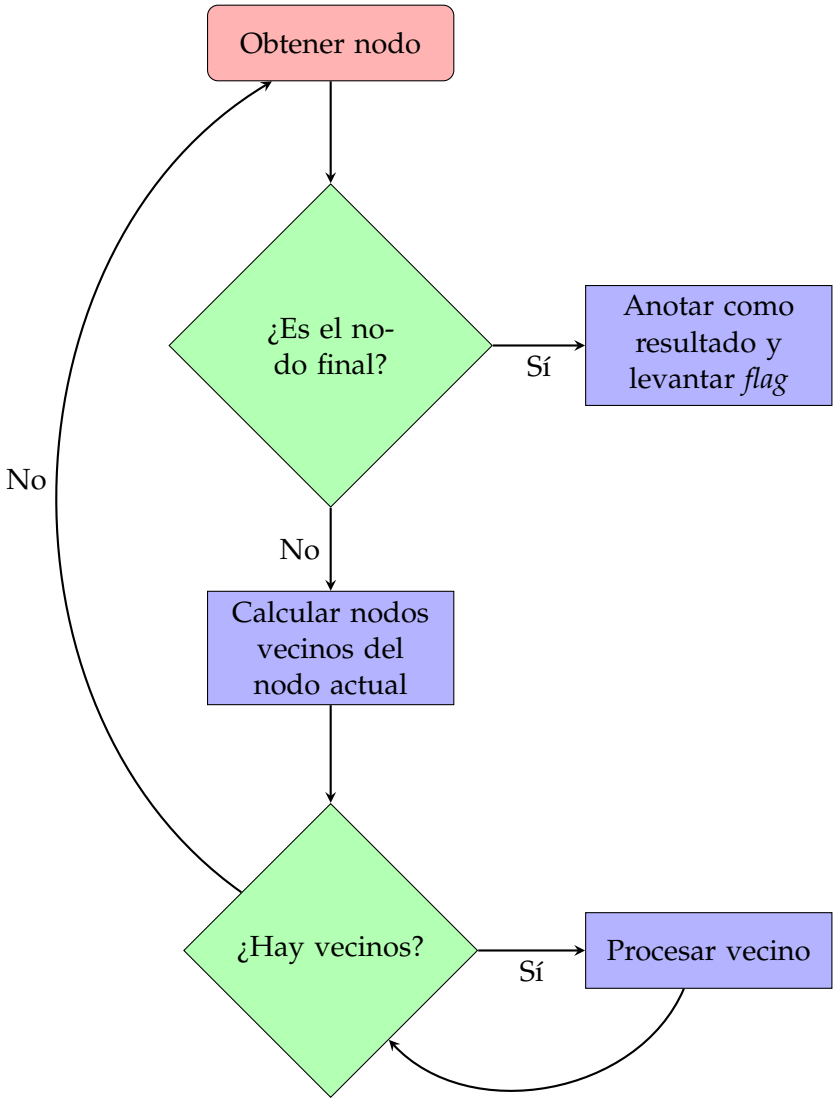


FIGURA 2.3: Representación de la sección paralela del algoritmo

Capítulo 3

Experimentos

Trabajo y Resultados

Capítulo 4

Conclusiones

Observaciones y Trabajos futuros

Bibliografía

[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.

[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.