Grado en Ingeniería Informática

# Comparación de algoritmos de pathfinding

## A comparison of pathfinding algorithms

Realizado por
Sergio Infante Paredes

Tutorizado por
Lorenzo Mandow Andaluz

Departamento
Lenguajes y Ciencias de la Computación
UNIVERSIDAD DE MÁLAGA

MÁLAGA, junio 2021

# Resumen

El problema de la búsqueda de caminos o pathfinding es muy recurrente en el mundo de los videojuegos. Estos en muchas ocasiones, tienen personajes que ya sea el jugador o la máquina determinan hacia adonde tienen que ir y estos deben moverse con una ruta relativamente realista, por lo que necesitamos que el coste computacional de la búsqueda de caminos sea el menor posible.

Este Trabajo Fin de Grado se centrará en el estudio, implementación y análisis de diferentes algoritmos de pathfinding para videojuegos. De esta manera, determinaremos cuál es el más rápido entre los comparados o cuál tiene menos coste computacional, además sirve como ampliación a los contenidos sobre pathfinding de la asignatura ”Inteligencia Artificial para Juegos” que he estudiado en este grado.

Se han estudiado e implementado 3 algoritmos: A*, HPA* y JPS, además de una interfaz gráfica para su ejecución. Se ha escogido A* como algoritmo de referencia y HPA* y JPS por ser algoritmos que mejoran el rendimiento de A*. HPA* aunque no obtenga el camino más corto, obtendrá buenos resultados en menos tiempo que A*. JPS por su parte, que es una modificación de A*, para explotar cualidades de los mapas con mallas de 8 vecinos.

Se ha desarrollado una aplicación para facilitar la interacción con los algoritmos mencionados anteriormente. La aplicación contiene 4 pestañas. Las 3 primeras sirven para ejecutar los algoritmos A*, HPA* y JPS respectivamente, y la última contiene las comparaciones de estos algoritmos. Además se ha añadido un manual que explica el uso de la aplicación y la implementación de nuevas funcionalidades.

**Palabras clave: Pathfinding, algoritmos de búsqueda, A*, HPA*, JPS**

# Abstract

The pathfinding problem is very recurrent in the world of video games, since these often have characters that either the player or the machine determine where they have to go and they must move with a realistic route. We need to reduce as much as possible the computational cost of the path search.

This daocument will focus on the study, implementation and analysis of different pathfinding algorithms for video games. The goal is determine wich is the fastest among those compared or which has the least computational cost, in addition it continues the content on pathfinding of the "Artificial Intelligence for Games" subject that I have studied in this degree.

Three algorithms have been implemented: A*, HPA* and JPS, as well as a graphical interface for their execution. A* has been chosen as the reference algorithm, HPA* for its operation, although it does not obtain the shortest path, but it will obtain good results in less time than A* and JPS which is a modification of A* to exploit qualities of maps with meshes of 8 neighbors.

An application has been programmed to allow easy interaction with the algorithms. The application contains four tabs. The first one is used to execute A*, the second is used to execute HPA*, and the third is used to execute JPS. The last tab contains all the comparisons between those algorithms. In addition, a user manual has been added that explains the use of the application and facilitates the implementation of new functionalities.

**Keywords: Pathfinding, search algorithms, A*, HPA*, JPS**

# Índice


**1. Introducción** **5**

1.1. Introducción . . . . . 5

1.2. Objetivos . . . . . 6

1.3. Estructura de la memoria . . . . . 6

1.4. Metodología . . . . . 7

1.5. Tecnologías usadas . . . . . 7

**2. Búsqueda de caminos en videojuegos** **9**

2.1. Representación . . . . . 9

2.2. Algoritmos . . . . . 10

2.2.1. A* . . . . . 11

2.2.2. HPA*(Hierarchical Pathfinding A*) . . . . . 11

2.2.3. JPS (Jump Point Search) . . . . . 12

**3. Algoritmos de búsqueda** **15**

3.1. Algoritmo A* . . . . . 15

3.2. Algoritmo HPA* . . . . . 17

3.3. Jump Point Search . . . . . 20

**4. Diseño e implementación** **27**

4.1. Diagrama de funcionamiento . . . . . 27

4.1.1. Útiles . . . . . 27

4.1.2. Heurísticos . . . . . 27

4.1.3. Nodo . . . . . 30

4.1.4. Mapa . . . . . 30

4.1.5. Grafo . . . . . 32

4.1.6. Tratador de mapas . . . . . 32

4.1.7. Ventana y Ventana-UI . . . . . 32

4.1.8. Controlador . . . . . 33

4.1.9. A*, HPA*, JPS, Dijkstra . . . . . 34
4.2. Visualización de los resultados . . . . . 34
4.3. Detalles de implementación . . . . . 38

**5. Análisis de rendimiento** **47**
5.1. Comparación entre A* y HPA* . . . . . 47
5.2. Comparación entre A* y JPS . . . . . 51
5.3. Comparación entre HPA* y JPS . . . . . 52
5.4. Conclusión de la comparación . . . . . 53

**6. Conclusiones y trabajo futuro** **57**
6.1. Conclusiones . . . . . 57
6.2. Trabajo futuro . . . . . 58

**Apéndice A. Manual de uso** **61**
A.1. Manual de uso . . . . . 61
A.2. Funcionamiento interno . . . . . 61
A.2.1. Funcionamiento de la interfaz . . . . . 61
A.2.2. Estructura de los algoritmos . . . . . 64
A.2.3. Llamando a la ejecución de los algoritmos mediante la clase controlador 68
A.2.4. Útiles, heurísticos y otras clases usadas en los algoritmos . . . . . 68
A.2.5. Implementando un nuevo algoritmo en el framework . . . . . 70

# 1 Introducción

## 1.1. Introducción

En este proyecto se aborda un problema frecuente en muchos videojuegos. Se trata del denominado ”pathfinding” o búsqueda de caminos entre dos puntos de un determinado entorno. Este problema aparece cuando uno o varios personajes de un videojuego deben moverse de manera autónoma entre dos puntos de un escenario. Un ejemplo típico se da cuando el jugador señala uno o varios personajes del juego e indica que desea desplazarlos a una posición indicada con un clic de ratón.

Los videojuegos necesitan cada vez más potencia computacional debido a la calidad visual y las propias dimensiones del juego, lo que nos lleva a la necesitad de optimizar al máximo el cómputo reservado para el cálculo de caminos. Por tanto, es necesario que el juego calcule una trayectoria eficiente, natural y libre de obstáculos para ir desplazando a los personajes, además calculan con frecuencia este tipo de trayectorias para personajes no jugadores (NPCs; Non Playable Characters).

Una aproximación básica del problema consiste en modelar el entorno del juego mediante algún tipo de grafo y calcular las trayectorias de los personajes dentro del mismo. Por tanto, pueden utilizarse algoritmos de inteligencia artificial, como el conocido algoritmo A*, para la tarea de la búsqueda de caminos.

El tipo de algoritmos utilizados en los videojuegos suele ser secreto comercial. Sin embargo, las descripciones de la literatura especializada apuntan al uso de algoritmos que obtienen una mayor eficiencia que A*, bien realizando pequeños sacrificios en la optimalidad de la solución alcanzada, o bien aprovechando características específicas de los grafos explorados, como puede ser el caso de las mallas cuadradas, el cual abordaremos más adelante.

## 1.2. Objetivos

El objetivo de este TFG es analizar diferentes algoritmos dentro del campo del pathfinding en la IA de videojuegos. Se ha utilizado como referencia el conocido algoritmo de búsqueda A*, por sus buenas propiedades teóricas y eficiencia práctica en muchos ámbitos. Además, se han estudiado, implementado y evaluado experimentalmente otros algoritmos más específicos que aumenten el rendimiento. El TFG se centra en algoritmos adecuados para representaciones basadas en mallas de 8 vecinos, ya que es la forma que toma el escenario en muchos videojuegos. Más concretamente se consideran el denominado algoritmo de búsqueda con punto de salto (JPS) y el algoritmo Hierarchical Pathfinding A* (HPA*).

Como parte de los objetivos del TFG se ha desarrollado un framework software que permita la importación de mapas de libre acceso, así como la incorporación y evaluación de los algoritmos mencionados anteriormente. Además, se ha añadido un manual de usuario detallado que explique el uso de dicho framework para facilitar la incorporación futura de otros algoritmos.

## 1.3. Estructura de la memoria

La memoria cuenta con 6 partes bien diferenciadas:

- **Capítulo 2.** Búsqueda de caminos en videojuegos: una introducción al funcionamiento de las búsqueda de caminos, algoritmos, alternativas y su representación.

- **Capítulo 3.** Algoritmos de búsquedas: una descripción más detallada y extensa del funcionamiento de los algoritmos expuestos anteriormente: A*, HPA* y JPS.

- **Capítulo 4.** Diseño e implementación: en esta sección se describirá el framework desarrollado, su funcionamiento, diagrama de funcionamiento, la visualización de los resultados y algunos detalles sobre la implementación.

- **Capítulo 5.** Análisis de rendimiento: en este apartado se expondrán los resultados de ejecución de los algoritmos, las comparaciones entre estos y una discusión a cerca de las ventajas de unos sobre otros.

- **Capítulo 6.** Conclusiones y trabajo futuro: se expondrán las conclusiones y las ideas a las que se han llegado durante la implementación de los diferentes algoritmos en el framework.
- **Anexo 1.** Manual de usuario: se incluye al final de la memoria un pequeño manual de usuario sobre el framework para su futuro uso y ampliación.

## 1.4. Metodología

Este trabajo es un desarrollo software por lo que lo apropiado es utilizar una metodología acorde al trabajo que se va a realizar.

La metodología acordada es la metodología SCRUM. Consiste en dividir el trabajo en hitos o metas y completarlo en un tiempo determinado llamado Sprint. Se realiza un seguimiento con reuniones para ver el avance del desarrollo y al final de cada Sprint otra para analizar el trabajo hecho y mejorarlo.

Se han realizado sprints de una semana y al final de este se ha hecho una reunión para evaluar el trabajo realizado y asignar el trabajo a realizar en el siguiente sprint.

## 1.5. Tecnologías usadas

Para realizar este trabajo se ha usado Python como lenguaje de programación con librerías como numpy, opencv, matplotlib para representar gráficos y PyQt5 y la aplicación Qt para crear la interfaz. El entorno de desarrollo que se ha utilizado es Pycharm en su versión EDU. Para la memoria se ha realizado un borrador en Microsoft Word por su facilidad de uso, pero posteriormente para la realización de la memoria se ha utilizado LaTeX.

# 2

# Búsqueda de caminos en videojuegos

## 2.1. Representación

Una de las técnicas más utilizadas para representar los mapas de un videojuego es el uso de mallas, una representación del espacio de juego donde cada casilla de la cuadrícula reproduce un punto donde puede estar un personaje o un obstáculo. Hay diversos tipos de mallas, pero las mas extendidas son las mallas cuadradas y las hexagonales.

La diferencia entre ambas, aparte de la forma que salta a la vista, es el coste de ir de un punto a otro. Mientras que en las mallas cuadradas, conocidas también como mallas de 8 vecinos, el coste de moverse en linea recta, es decir, en vertical u horizontal, es 1 y el coste de moverse en diagonal es $\sqrt{2} = 1,414$, en las mallas hexagonales no existe el término diagonal, puesto que podemos movernos de una posición a otra solamente en línea recta, como se puede ver en la figura 1. En ella, podemos apreciar los valores 1000 y 1414 aparte de los ya mencionados 1 y $\sqrt{2}$, las versiones enteras de los valores decimales anteriores son más sencillas de tratar para un ordenador.

Los escenarios que vamos a usar son mapas de libre acceso pertenecientes al videojuego Baldur's Gate II, cedidos por su desarrolladora Bioware para uso de investigación y disponibles en la web $www.movingai.com$ [8]. Estos escenarios tienen la estructura de una matriz.

Dicho esto, lo que mejor se adapta a nuestro problema son las mallas cuadradas, tanto por su uso tan extendido en la mayoría de videojuegos, como por su geometría. Así, podemos transformar un mapa de un videojuego en una matriz de $n \times m$ dimensiones, donde cada casilla tendrá un valor perteneciente a {0, 1}, 0 representará un obstáculo y 1 representará una posible

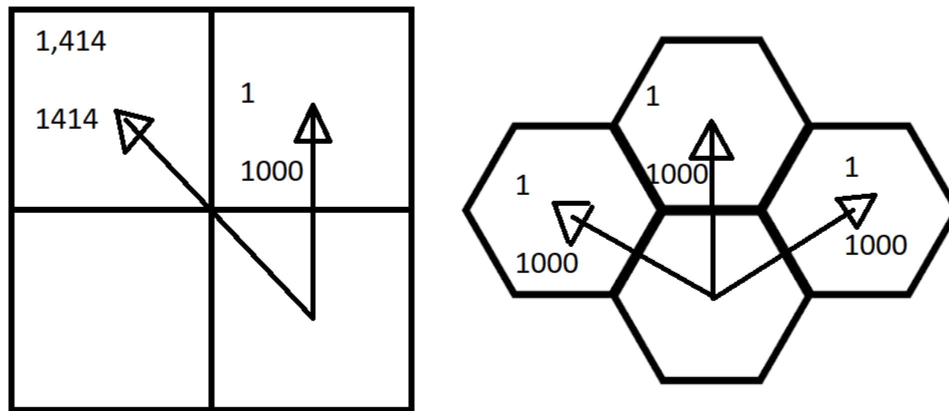



Figura 1: Mallas y sus costes de movimiento. [9]

casilla donde podemos ir como ocurre en la figura 2.

## 2.2. Algoritmos

Una malla cuadrada no es más que un grafo con $n \times m$ nodos, donde $n$ es el número de filas que tiene la malla y $m$ el número de columnas. Dicho esto, podemos encontrar básicamente dos tipos de algoritmos de búsqueda válidos para esta situación:

- Los algoritmos de búsqueda con árbol.
- Los algoritmos con retroceso.

En general, los algoritmos de búsqueda con retroceso se diferencian de los otros en que, si bien suelen ser más lentos y su complejidad en tiempo aumenta mucho con la longitud de la solución, su consumo de memoria se limita al camino recorrido, a diferencia de los algoritmos de búsqueda en árbol que tienen que guardar en memoria todas las ramas que se van generando, para poder volver a ellas, pero en un tiempo más reducido. Un problema añadido a la búsqueda con retroceso en mallas, es que en su forma más básica no se controla la aparición de estados repetidos. Esto puede incrementar exponencialmente el tiempo de ejecución. Es por eso por lo que vamos a usar algoritmos de búsqueda en árbol, en este TFG planteamos hacer los cálculos lo más rápido posible, teniendo en cuenta que el consumo de memoria de este tipo de algoritmos no será un problema, es por esto que usaremos el algoritmo A* como referencia,

aunque se considere un algoritmo lento en el pathfinding en videojuegos y además vamos a ver varias mejoras que hacen que A* se convierta en un algoritmo eficiente para estos.

No podemos hablar de algoritmos de pathfinding sin mencionar al algoritmo de Dijkstra, pues todos los algoritmos que vamos a estudiar se basan en este. Sin entrar mucho en detalle, el algoritmo de Dijkstra encuentra el camino más corto entre 2 nodos en un grafo mediante la expansión de los caminos que menos distancia llevamos recorrida, haciendo uso de una lista de nodos abiertos para saber por donde puede moverse y una lista de nodos cerrados para saber cuales ya ha visitado.

Los algoritmos que vamos a estudiar son los siguientes:

- A*
- HPA* (Hierarchical Pathfinding A*)
- JPS (Jump Point Search)

### 2.2.1. A*

Fue creado como parte del proyecto Shakey, que tenía como objetivo construir un robot móvil que pudiera planificar sus propias acciones convirtiéndose en una mejora del algoritmo de Dijkstra, pues si este hace uso de la distancia recorrida para seguir expandiendo nodos, A* hace uso de esta medida aparte de una estimación, haciendo más precisa la búsqueda en la malla. Hemos escogido este algoritmo porque se considera un algoritmo óptimo, pues encuentra el camino con menos coste en el menor número de expansiones posibles [7].

### 2.2.2. HPA*(Hierarchical Pathfinding A*)

Creado por Adi Botea, Martin Müller y Jonathan Schaeffer [1], es un añadido al algoritmo A* pues necesitamos de un procesamiento de la malla previo a la búsqueda en sí. Este algoritmo abstrae la malla, dividiéndola en ”clusters” y convirtiéndola en un grafo abstracto de los posibles caminos para ir de un cluster a otro. Esto se traduce en que en lugar de tener una matriz de $n \times m$ posiciones y por ende un grafo de $n \times m$ nodos, tendremos un grafo de muchos menos nodos, aumentando la velocidad de búsqueda a costa de no encontrar el camino más óptimo posible.

### 2.2.3. JPS (Jump Point Search)

Fue creado por Daniel Harabor y Alban Grastien [5], es una modificación del algoritmo A*, diseñado específicamente para ser usado en mallas cuadradas. Si bien es similar al algoritmo A*, su forma de expandir nodos cambia, este algoritmo va dando saltos y podando sucesores, con el fin de eliminar las simetrías dentro de una malla.

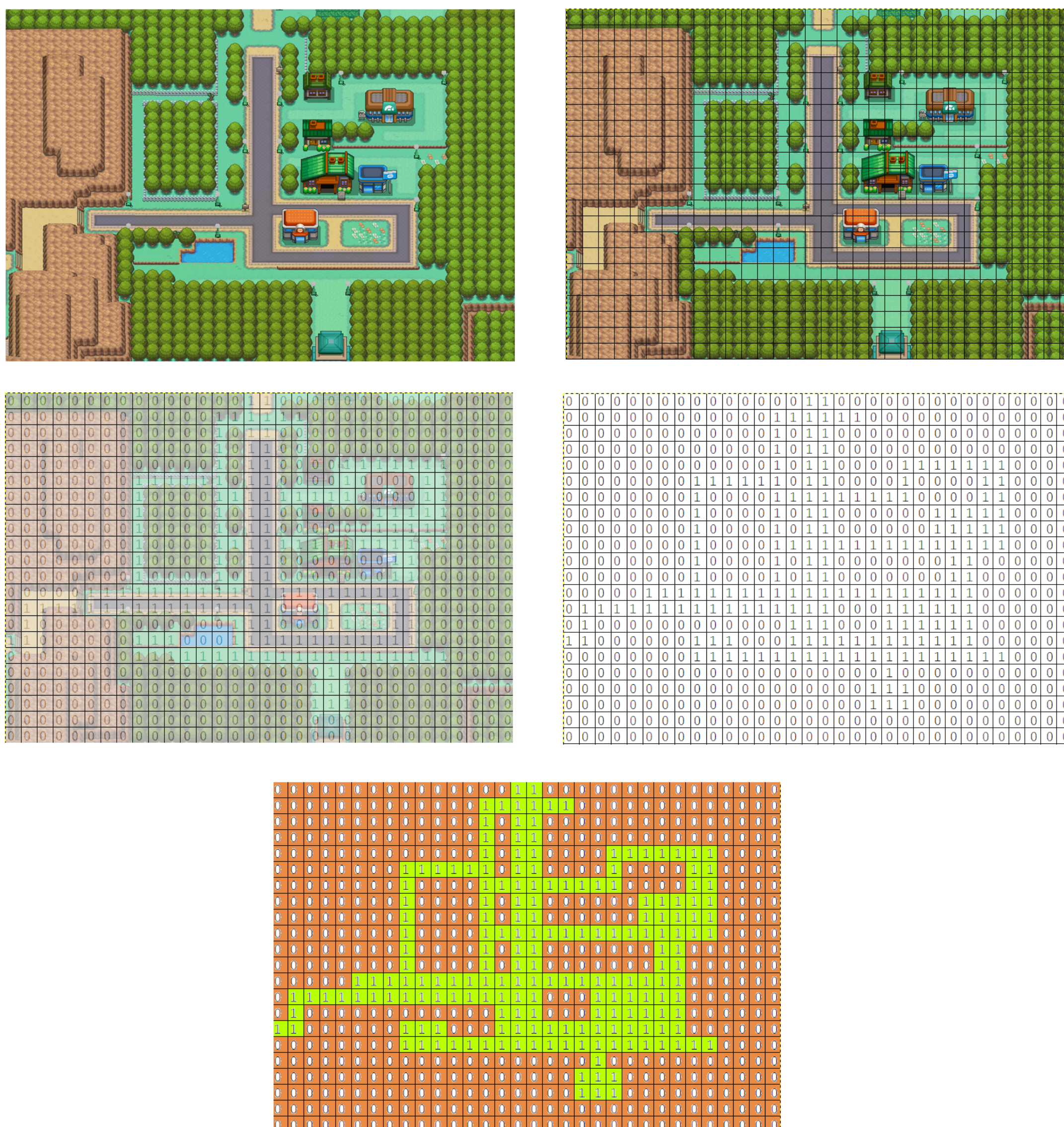


Figura 2: Transformación de un mapa de videojuego a una matriz con los valores de cada casilla. Imagen de un pueblo del videojuego Pokemon [2]

# 3
# Algoritmos de búsqueda

En este capítulo se expondrá el funcionamiento de cada uno de los algoritmos que se van a estudiar de una forma simplificada.

## 3.1. Algoritmo A*

Existen algoritmos que se guían solamente por la función heurística, que es una aproximación del coste de llegar desde el nodo actual al nodo objetivo, lo que hace que no siempre se indique el camino con coste más bajo, derivando en que, en caso de encontrar la solución, no sea la óptima. También existen los algoritmos que se guían por el camino ya recorrido, como es el caso del algoritmo de Dijkstra, que siempre encuentra la solución óptima, Es por eso por lo que A* no solo se basa en una función heurística o en el camino recorrido, sino que tiene en cuenta ambos costes, obteniendo así la función de evaluación del algoritmo:
$f(n) = g(n) + h(n)$,
donde $g(n)$ es el coste real del camino del nodo $n$ hasta el nodo objetivo y $h(n)$ es la función heurística del nodo $n$. En este caso usaremos la distancia octil como función heurística, ya que se adapta perfectamente a nuestro problema de malla cuadrada. Para calcular la distancia octil necesitamos saber cuántos pasos tenemos que dar en columnas y cuántos en filas. Como los movimientos en diagonal son más costosos que en horizontal o vertical vamos a usar el menor valor entre filas y columnas para movernos en diagonal y el resto nos moveremos en horizontal o vertical, como se puede ver en la figura 3

Las propiedades del algoritmo A* son las siguientes:

- Para que A* encuentre el camino óptimo, la función heurística debe ser admisible, es-

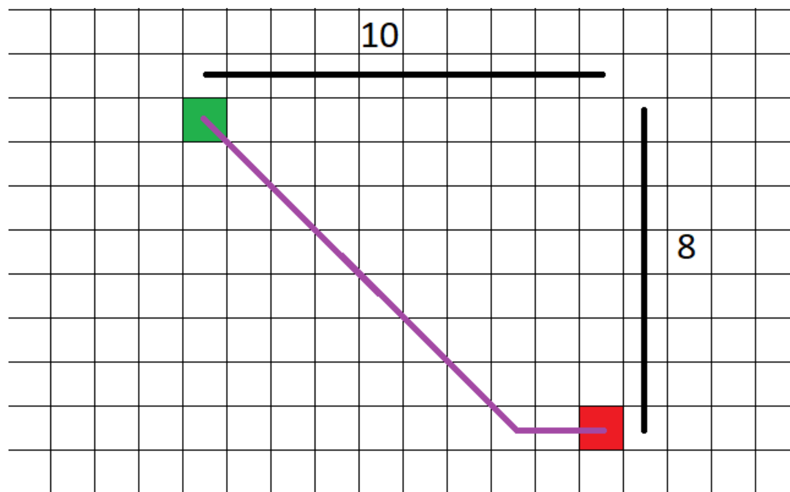



Figura 3: Cálculo de distancia octil

to significa que el valor de $h(n)$ no sobrepase al coste real de alcanzar el nodo objetivo(desde $n$).

- La restricción monótona se define de la siguiente manera: si tenemos un nodo $i$ y un nodo $j$ y $j$ es hijo de $i$, el coste del arco entre $i$ y $j$ debe ser mayor o igual a la diferencia del heurístico de esos dos nodos.

  Si un heurístico satisface la restricción monótona, A* encontrará un camino óptimo para todo nodo seleccionado que cumple que $g(n) = g^*(n)$, donde $g^*(n)$ es el coste óptimo.

Sabiendo esto, solo queda mostrar la forma en que trabaja A*. De una forma simplificada, trabaja con dos listas, una de nodos abiertos y otra de nodos cerrados. La lista de abiertos se ordena de menor a mayor mediante la función $f(n)$ vista anteriormente y el algoritmo busca por el primer nodo de la lista, que pasa a estar cerrado. El árbol lo definen los nodos, pues cada uno tiene un puntero que apunta a su ”padre” para después saber cuál es el camino solución. Si nos encontramos con que llegamos a un nodo por dos o más caminos distintos, su padre será con el que menor $f(n)$ tenga el nodo. Cuando el algoritmo encuentre el nodo objetivo y toque su expansión, se habrá acabado de ejecutar el algoritmo y por tanto encontrado encontrado la solución óptima. Existe la posibilidad de que no exista ningún camino solución, por lo que llegaría el momento en que recorriésemos toda la lista de abiertos quedándonos sin nodos que expandir y devolviendo un camino vacío.

Aunque ahora hablemos de una lista de abiertos y otra de cerrados, como nuestra imple-

mentación hace uso de una malla cuadrada, podemos usar matrices, lo que hace que buscar un nodo sea directamente una búsqueda en matriz y no una búsqueda dentro de una lista, lo que hace que sea más eficiente. El pseudocódigo de A* se puede observar en la figura 1.

**Algoritmo 1:** Pseudocódigo A*

```
   Result: Una lista con el camino recorrido o vacía en el caso que no encuentre ningún camino
 1 Crear un árbol de búsqueda A con raíz en s, y una lista de nodos ABIERTOS con s
 2 Crear una lista de nodos CERRADOS vacía.
 3 Si ABIERTOS está vacía, entonces devolver 'FRACASO'
 4 Si n es objetivo, entonces devolver el camino de s hasta n en A
 5 Expandir n, M <- calculaSucesores (n, G)
 6 for n2 en M do
 7     if n2 es nuevo (n2 no está ABIERTO ni CERRADO) then
 8         Poner un puntero de n2 ->n
 9         Añadir n2 a ABIERTOS
10     end
11     if n2 no es nuevo, y el valor de g(n2) es menor a través del nuevo camino then
12         Redirigir su puntero hacia n
13         Si n2 está en CERRADOS, entonces pasarlo a ABIERTOS
14     end
15 end
16 Ordenar ABIERTOS por orden creciente en el valor de f(n) = g(n)+h(n)
17 Volver a línea 3
```

## 3.2. Algoritmo HPA*

Como hemos dicho antes, A* es un algoritmo rápido, aunque dependiendo de nuestro espacio de estados puede llegar a ser lento y a ocupar mucha memoria. Imaginemos un espacio de estados de 512x512 posiciones, entonces HPA* se encarga de reducir el espacio de estados del problema de forma significativa, lo que supondría una rebaja bastante grande a la hora de la ocupación de memoria y la velocidad de búsqueda.

HPA* no es más que un añadido al algoritmo anterior, si bien al final del todo aplicaremos A*, la malla donde se moverá nuestro personaje se abstrae creando un grafo dividiéndose en

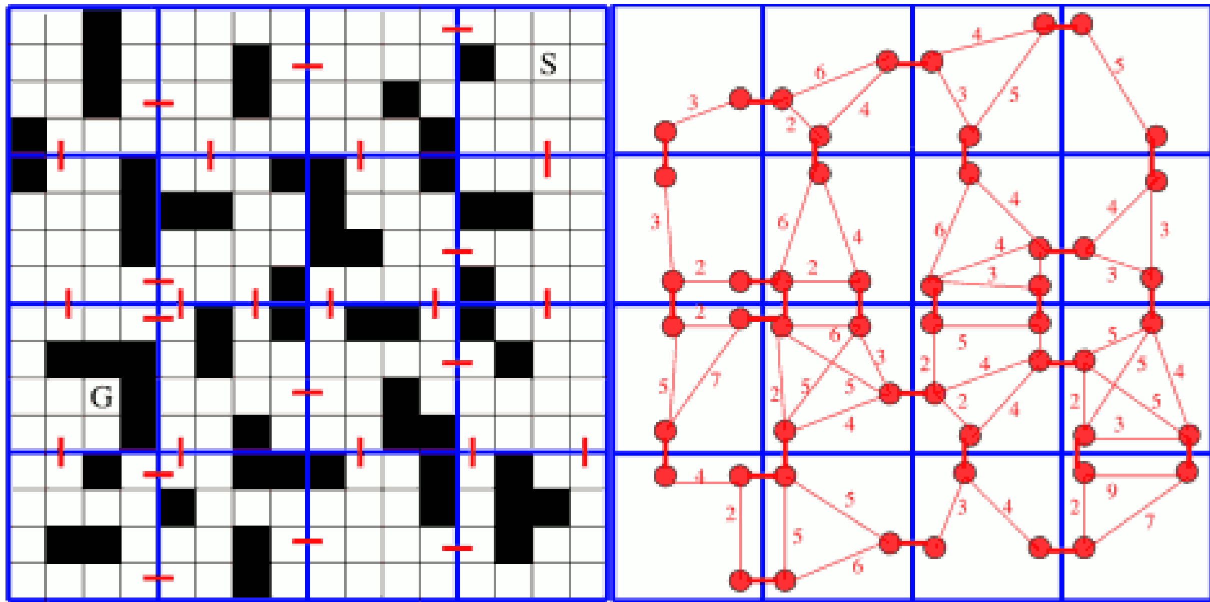



Figura 4: Ejemplo división en clusters.[4]

clusters de tamaño predefinido y solo formarán parte de nuestro grafo las uniones de estos clusters, nuestros puntos de inicio y de final como se puede ver en la figura 4. También estos clusters formarán grupos que a su vez serán clusters más grandes, por lo que podríamos escalar el problema sin aumentar mucho el número de nodos de nuestra malla. La consecuencia de esto es que obtenemos una solución en menor tiempo que el algoritmo A* a cambio de que esa solución puede que no sea la óptima. Estas divisiones en clusters y creación de grafos no se hacen en tiempo de ejecución, sino que se realizan de manera offline, es decir, están precalculadas cuando el algoritmo intenta hacer la búsqueda.

Si lo pensamos bien, este problema se puede extrapolar a viajar entre ciudades: por ejemplo, si queremos ir desde la Universidad de Málaga hacia la Universidad de Almería, podemos aplicar A* a un mapa con todas las carreteras de Andalucía, pero eso sería costoso computacionalmente. Entonces lo que haríamos es primero salir de Málaga y entrar en una autovía (salir de nuestro pequeño cluster), luego seguir esa autovía hasta llegar a Almería (viajar de un cluster a otro) y por último ya estando en Almería buscar la universidad en la ciudad (movernos dentro de otro cluster).

La forma de encontrar estas conexiones, que a partir de ahora llamaré entradas, es mediante el análisis de los bordes entre cluster y cluster (l1 y l2), ya sea horizontal o vertical. A la hora de identificar estas puertas, para una casilla $t \in l1 \cup l2$, definimos una función llamada

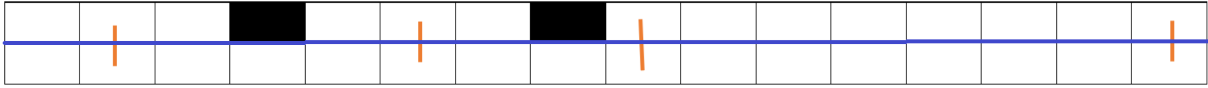


Figura 5: Conexiones de clusters.

$simetrico(t)$, que pertenece al otro cluster. Entonces necesitamos encontrar el tamaño de la entrada para poder construirla, para que la entrada exista tanto $t$ como $simetrico(t)$ deben ser casillas libres y no obstáculos, la entrada tampoco puede ser más grande que el propio cluster, entonces contamos todos los simétricos que vamos encontrando hasta que uno de ellos, $simetrico(t)$ o $t$ sean obstáculos, así podemos calcular los tamaños de las entradas. Como se puede ver en la figura 5, donde la línea azul es el borde entre clusters y las filas superior e inferior son l1 y l2 respectivamente. Las líneas rojas unen 2 clusters, con esas entradas, dependiendo del tamaño que nosotros asignemos, existirán 2 tipos de puertas, llamémoslas pequeñas y grandes. Si definimos 6 como el tamaño mínimo para una puerta grande, al encontrar una entrada de tamaño 5, que será una puerta pequeña, asignaremos una sola línea como paso de un cluster a otro que estará situada en la mitad de la entrada. Si por el contrario encontramos una entrada de tamaño 6 o más, podremos colocar una puerta grande, que consistirá en dos líneas rojas en los extremos de las entradas. Las líneas de la izquierda son dos entradas pequeñas independientes y las dos líneas de la derecha forman una puerta grande.
Tras preprocesar la malla y obtener el grafo, tenemos que aplicar A* para encontrar el camino óptimo sobre ese grafo, obteniendo así una búsqueda con menos nodos que con el algoritmo A* básico.

Después de aplicar A*, tenemos que aplicar un refinamiento al camino, pues sabemos que tenemos que ir por ejemplo del nodo $H$ al nodo $K$, por lo que tendremos que recuperar ese camino dentro del grafo obtenido tras preprocesar la malla. Por último, podríamos hacer un "suavizado" al camino obtenido consiguiendo así recorrer aún menos distancia y un movimiento más realista del personaje (nosotros por ejemplo no nos movemos en una malla cuadrada, por lo que, aunque nuestro escenario sí lo sea, el movimiento puede ser en línea recta desde el inicio al final), esta parte del algoritmo no se implementa en el framework debido a que solo estamos buscando el camino y no tenemos un personaje que pueda hacer ese movimiento suavizado. Por ejemplo, en la figura 6, se muestra en negro el camino obtenido por A* y en azul el obtenido tras suavizar el camino y en la figura 2 se puede observar el pseudocódigo.

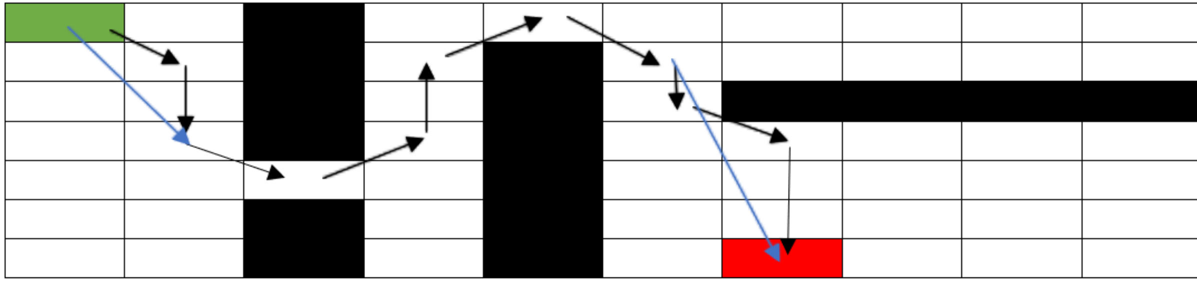


Figura 6: Suavizado con HPA*.

**Algoritmo 2:** Pseudocódigo HPA*

Preprocesar la malla
- Dividir la malla en clusters
- Establecer que nodos pertenecen a cada cluster
- Establecer la conexión interna de cada cluster y la unión entre estos

Aplicar el algoritmo A* al grafo obtenido en el paso 1

Si A* devuelve FRACASO acaba el algoritmo y devuelve FRACASO

Aplicar suavizado al camino obtenido por A*

## 3.3. Jump Point Search

A diferencia del algoritmo HPA*, que es un añadido al algoritmo base A*, JPS o jump point search es una modificación del propio A* pensada para mallas cuadradas. Con este algoritmo explotamos una cualidad que tienen este tipo de mallas, la existencia de simetrías. En una malla sin obstáculos no solo existe una manera de ir de un punto a otro, sino que tenemos varias formas de llegar, como es el caso de la figura 7.

Esta poda de simetrías se lleva a cabo a la hora de generar sucesores, pues es el único cambio que tiene este algoritmo respecto al algoritmo A*.

Para conseguir la poda necesitamos describir varios conceptos:

- **Vecino:** es el conjunto de nodos adyacentes al nodo actual. En una malla de 8 vecinos como es el caso tendremos las esquinas, los nodos superior e inferior y los laterales, tal y como se puede ver en la figura 8 a.

- **Vecino natural:** un camino $\pi = < a, b, c, d >$ significa que partimos desde $a$ hacia $d$ pasando por $b$ y $c$. Sabiendo esto, podemos definir un vecino natural comparando 2

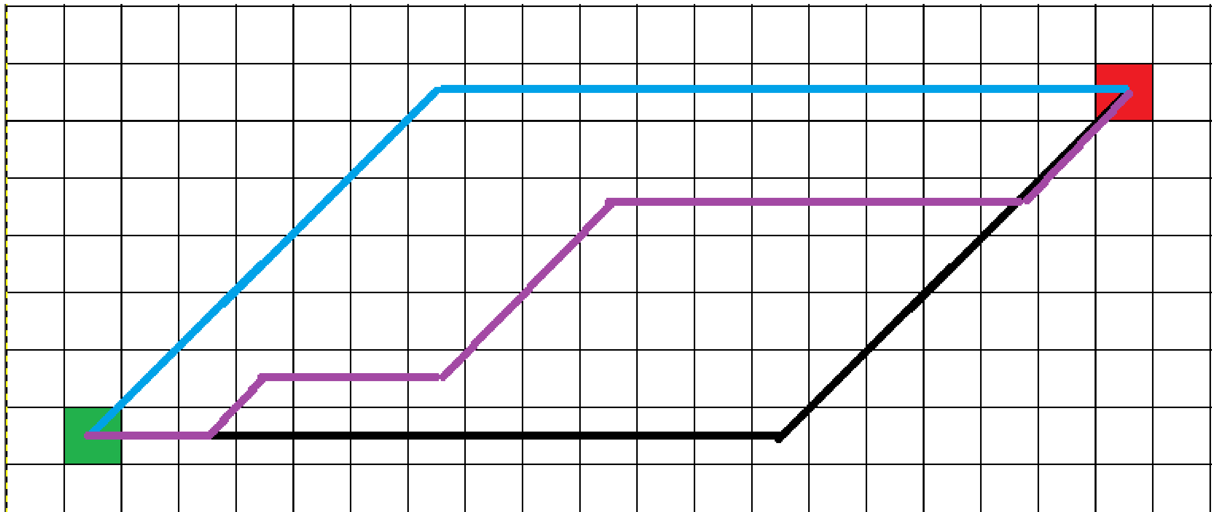


Figura 7: Simetrías en mallas cuadradas

caminos:

- A, que comienza con el nodo $p(x)$ (padre de x), visita $x$ y acaba en $n$.
- B, que empieza en $p(x)$ y acaba en $n$ pero no pasa por el nodo $x$.

Se da por hecho que todos los nodos visitados por A o B pertenecen a los vecinos de $x$. Si el nodo inicial es el nodo $x$, entonces $p(x)$ es null y no se poda nada. Las dos reglas que sigue la poda dependen de si el movimiento desde $p(x)$ hacia $x$ es diagonal u horizontal/vertical:

- Movimientos horizontales/verticales: Cualquier nodo $n \in vecinos(x)$ que cumpla la siguiente condición: $longitud(< p(x), ..., n > |x) \leq longitud(< p(x), x, n >)$, es un vecino natural. Por ejemplo, en la figura 8c siendo 4 $p(x)$, el único vecino natural es el 5.
- Movimientos diagonales: es similar a la regla anterior solo que en este caso la longitud debe ser estrictamente menor: $longitud(< p(x), ..., n >) < longitud(< p(x), x, n >)$. Por ejemplo, en la figura 8b siendo 6 $p(x)$ los nodos naturales son el 2, 3 y 5.

- **Vecino forzado:** es todo nodo vecino de $n$ que no es natural y cumple : $longitud(< p(x), x, n >) < longitud(< p(x), ..., n >)$, como se puede apreciar en la figura 9, donde

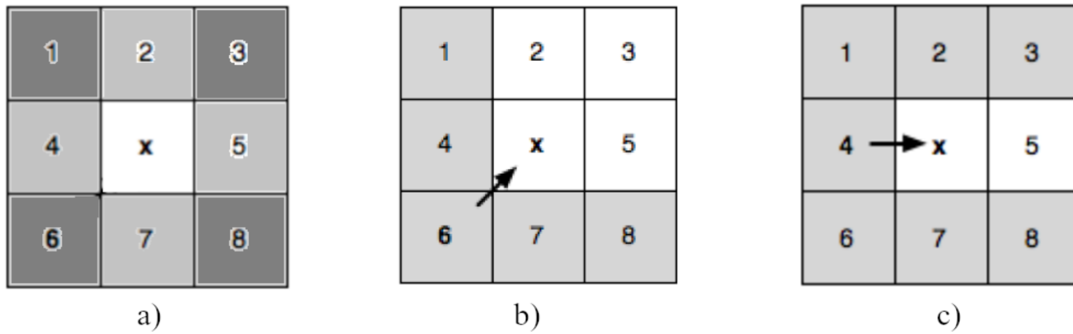



Figura 8: Vecinos naturales [5]

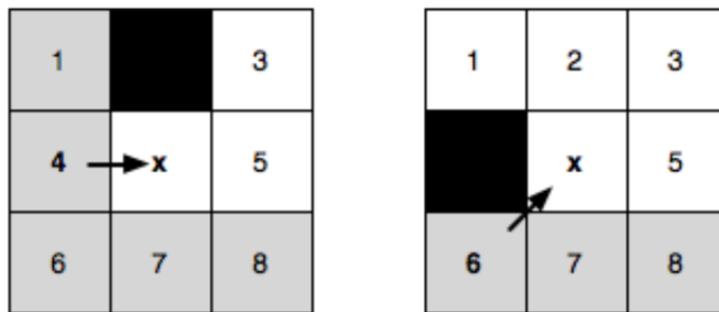



Figura 9: Vecinos forzados [5]

ocurre un movimiento horizontal, 3 es un vecino forzado de x puesto que $longitud(< 4, x, 3 >) < longitud(< 4, 7, 5, 3 >)$

- **Punto de salto:** $y$ es un punto de salto desde $x$ si avanzando en una dirección $\vec{d}$, $k$ minimiza la expresión $y(k) = x + k\vec{d}$, donde $k$ es el número de pasos y se cumple una de las siguientes condiciones:

  1. El nodo $y$ es el nodo objetivo.
  2. El nodo $y$ tiene un vecino forzado.
  3. $\vec{d}$ es un movimiento diagonal tal que $\vec{d} = (\vec{dx}, \vec{dy})$. Existe un nodo $z = y + k\vec{d_i}$ tal que $\vec{d_i} \in \{\vec{dx}, \vec{dy}\}$ y es un punto de salto por las dos primeras condiciones.

Ahora vamos a explicar el funcionamiento del algoritmo a partir de su pseudocódigo. Como hemos dicho antes, la única parte que cambia del algoritmo A* es el cálculo de sucesores, que se puede observar en la figura 3.

A simple vista, no difiere mucho del cálculo de vecinos normal de A*, pues tenemos que buscar los 8 vecinos. El cambio viene en la función ”salto”, que es el corazón del algoritmo JPS y se describe en la figura 4.

**Algoritmo 3:** Cálculo de sucesores JPS

```
Result: Una lista con los sucesores del nodo (fila, columna)
1 Direcciones = {Norte, Sur, Este, Oeste, NorEste, NorOeste, SurOeste, SurEste}
2 Vecinos = []
3 for i en Direcciones do
4     ps = salto(fila, columna, i) if ps != Null then
5         Vecinos.add(ps)
6     end
7 end
8 return Vecinos
```

Como vemos, este algoritmo funciona de forma recursiva tal y como lo describen originalmente sus creadores, de forma que se expande continuamente en una dirección buscando un punto de salto. En caso de que sea diagonal, cada expansión conlleva dos expansiones, una horizontal y otra vertical, tal y como se ve en la figura 10.

En este algoritmo se ve el uso de los vecinos naturales y forzados explicados anteriormente. Si en una expansión solo existen nodos naturales, no tendrá sentido explorar las filas o columnas adyacentes, puesto que con esa exploración ya tenemos un camino óptimo hacia todo nodo expandido. Sin embargo, en caso de que exista un vecino forzado, quiere decir que para llegar a ese vecino hay que pasar obligatoriamente por el nodo $x$ que corresponda en ese momento, por lo que ese nodo $x$ será un nuevo sucesor del nodo expandido. Esto se aprecia en la primera imagen de la figura 10. Estamos expandiendo el nodo $p(x)$ y hacemos un salto hacia la derecha, una vez alcanzado el punto $y$, reconocemos a $z$ como un vecino forzado de $y$, por lo que este se convierte en punto de salto.

En el caso diagonal, un punto de salto también será un sucesor, pero además, si en una de las expansiones rectas encontramos un punto que tiene vecinos forzados, quiere decir que nuestra expansión en diagonal es punto de salto y por tanto sucesor. Esta idea se puede ver en la segunda imagen de la figura 10, expandimos diagonalmente y en la última expansión

**Algoritmo 4:** Función salto utilizada en el algoritmo JPS. salto(fila, columna, dir)

```
nodoActual = fila, columna
filaP = fila-direccion.Y
columnaP =columna-direccion.X
if nodoActual no es una posicion válida (es un obstaculo o esta fuera del mapa) then
    return Null
end
if nodoActual es objetivo then
    return nodoActual
end
if vecinos(nodoActual) contiene vecinoForzado then
    return (fila, columna)
end
if direccion == diagonal then
    for direccion d∈ {dy, dx} do
        if salto(filaSig, colSig, d) != Null then
            return nodoActual
        end
    end
    return salto(filaSig, colSig, direccion)
end
```

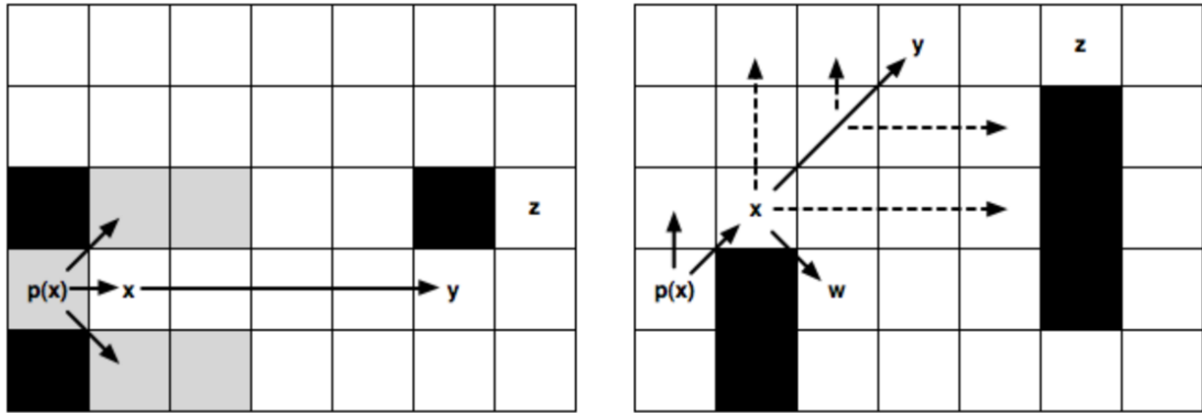



Figura 10: Saltos en diagonal y horizontal [5]

horizontal, reconocemos a $z$ como un punto de salto, por lo que nuestro nodo $y$ se convierte en un sucesor.

Una vez hayamos explorado las 8 direcciones con sus correspondientes saltos, los nodos resultantes serán los sucesores de nuestro nodo expandido, añadiéndose a la lista de abiertos correspondiente del algoritmo A*

# 4
# Diseño e implementación

En este capítulo se mostrarán ejemplos de las ejecuciones de los algoritmos, un diagrama que muestra el funcionamiento del framework y algunos detalles de la implementación de los algoritmos.

## 4.1. Diagrama de funcionamiento

Tras dar una visión global explicaremos que hace cada clase y cuales son las relaciones entre ellas. En general el framework, cuyo diagrama de clases se puede ver en la figura 11 de una manera simplificada o en la figura 12 con un diagrama UML, tiene como corazón los algoritmos JPS, A* y HPA*. Los usuarios interactúan con estos algoritmos mediante la clase Ventana, que es la interfaz. Los algoritmos usan las distintas clases para hacer sus búsquedas, ya sea recuperar el camino mediante la clase Útiles, usando los heurísticos de la clase con el mismo nombre o las clases Mapa y Nodo que son las herramientas principales que usan estos algoritmos.

### 4.1.1. Útiles

Este archivo no es una clase en sí, actualmente solo tiene 1 método, pero está pensado para en un futuro seguir añadiendo métodos conforme se expanda el framework y se vayan necesitando.

### 4.1.2. Heurísticos

Como su nombre indica, se trata de un archivo donde están distintos heurísticos que podemos usar en nuestras búsquedas. Los heurísticos actualmente implementados se pueden

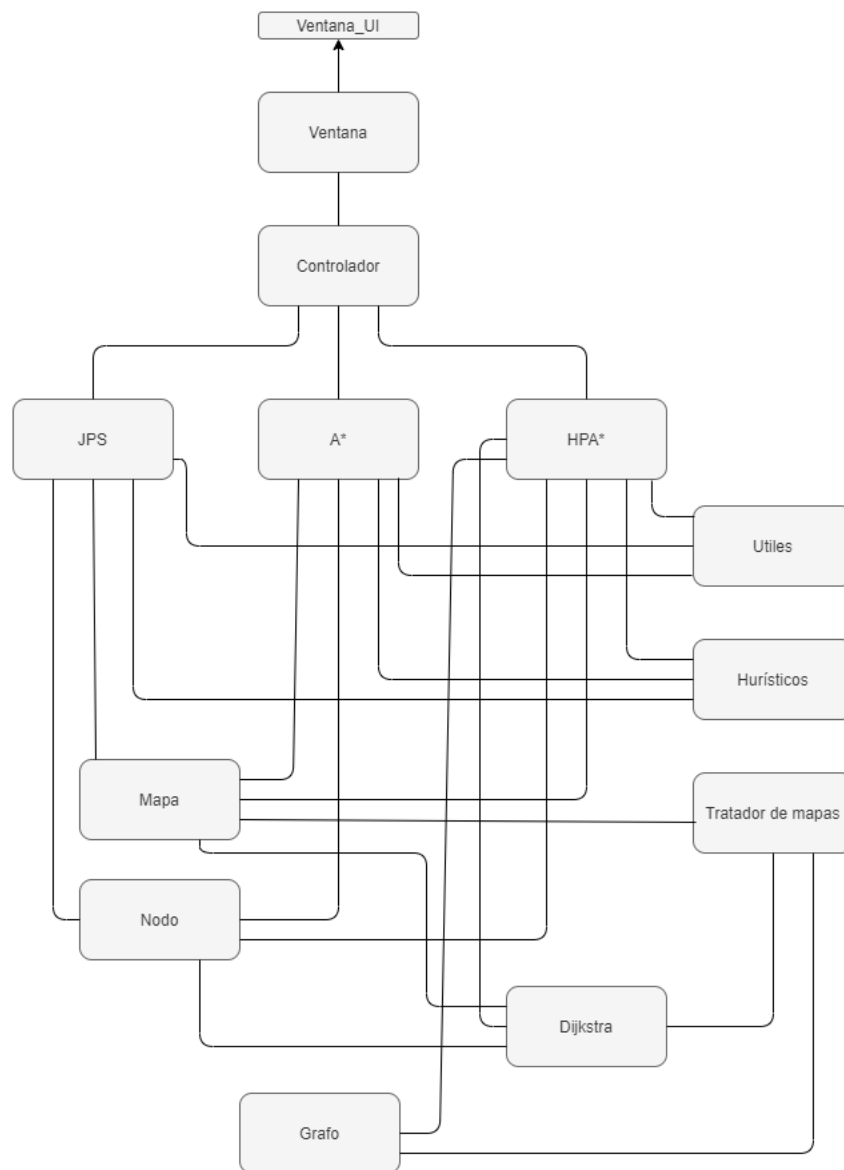



Figura 11: Diagrama de clases

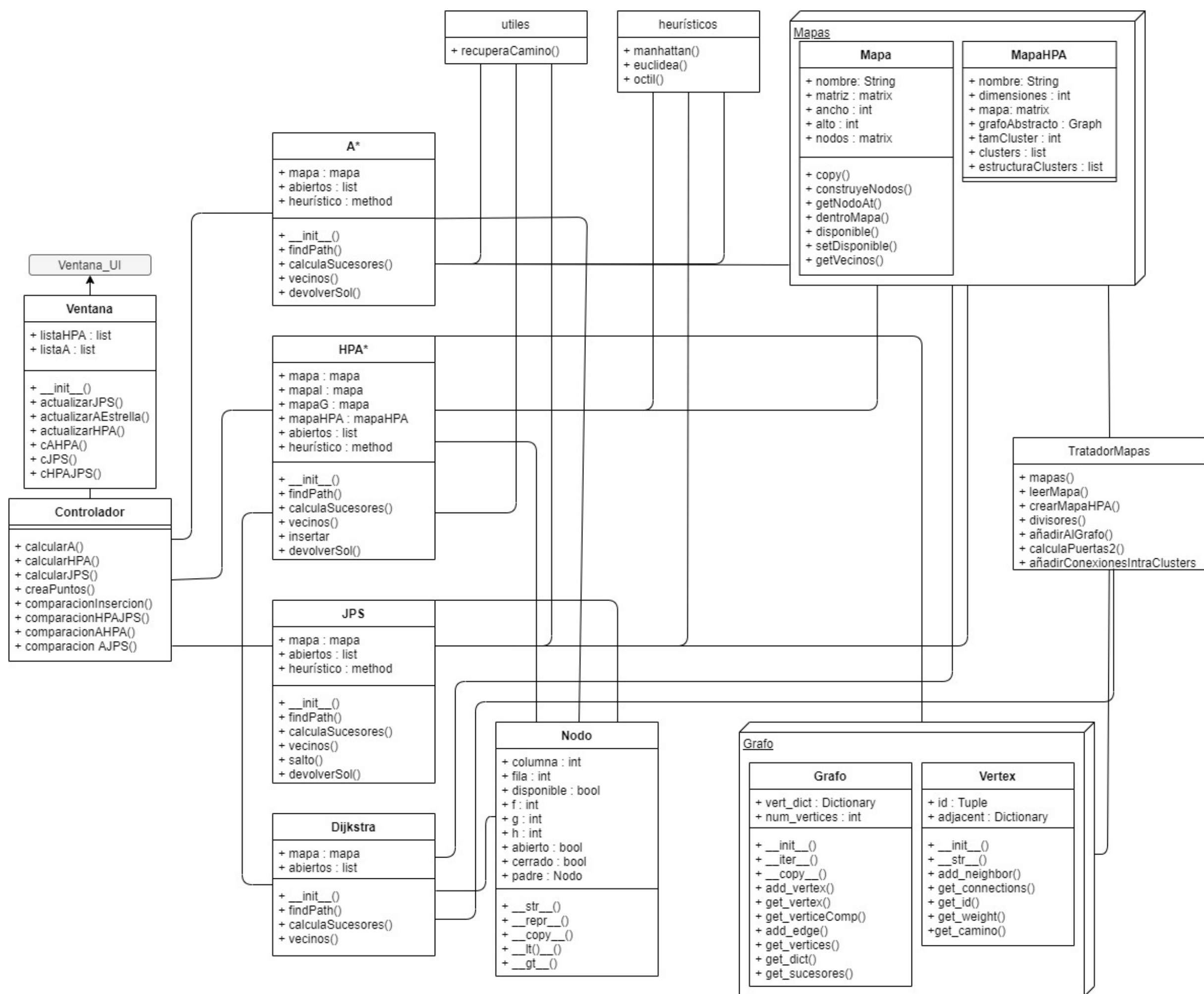



Figura 12: Diagrama UML

observar en la figura 5

**Algoritmo 5:** Heurísticos implementados en el framework

```
def manhattan(dx, dy) :
    return dx + dy
end
def euclidea(dx, dy) :
    return math.sqrt(dx * dx + dy * dy)
end
def octil(dx, dy) :
    F = 1414
    return F * dx + (dy-dx)*1000 if (dx <dy) else F * dy + (dx-dy)*1000
end
```

#### 4.1.3. Nodo

Mediante los nodos se construye el árbol de búsqueda, pues es esta clase la que tiene la información de si un nodo esta abierto o cerrado, cual es su padre para poder recuperar el camino una vez finalizada la búsqueda o ver si ese nodo es un obstáculo o es un punto libre para pasar, además de contener las evaluaciones necesarias para determinar el orden de expansión en los algoritmos. Los atributos de la clase nodo se pueden apreciar en la figura 6

#### 4.1.4. Mapa

Este archivo contiene 2 clases, una es la clase Mapa() y la otra MapaHPA(). La clase MapaHPA contiene el nombre, dimensiones, matriz, grafo, etc; necesarios para realizar la búsqueda jerárquica.

Por otro lado la clase Mapa() es la que contiene los mapas que se usan en la búsqueda con los diferentes algoritmos, siendo este el que se comprueba para obtener los sucesores físicos, obstáculos, etc. Los atributos de estas clases se observan en la figura 8.

Además de este constructor, contiene diferentes métodos que sirven para inicializar la matriz de nodos, comprobar si un punto esta dentro del mapa o si es un obstáculo, recuperar información de un nodo almacenado, etc.

**Algoritmo 6:** Atributos de la clase nodo

```
class Nodo(): :
    def __init__(self, fila, columna, walkable = True, f=0, g=0, h=0, abierto = False, cerrado =
     False, padre = None). :
        self.columna = columna
        self.fila = fila
        self.walkable = walkable
        self.f = f
        self.g = g
        self.h = h
        self.abierto = abierto
        self.cerrado = cerrado
        self.padre = padre
    end

end
```

**Algoritmo 7:** Atributos de la clase MapaHPA y la clase Mapa

```
class MapaHPA(): :
    def__init__(self, n, d, m, grafo, tamCluster=0, clusters=[], estructuraCluster=[]) :
        self.nombre = n
        self.dimensiones = d
        self.mapa = m
        self.grafoAbstracto = grafo
        self.tamCluster=tamCluster
        self.clusters=clusters
        self.estructuraCluster=estructuraCluster
    end
end
```

```
class Mapa(): :
    def__init__(self,nombre, alto, ancho, matriz, nodos = None, cop = False) :
        self.nombre = nombre
        self.matriz = matriz
        self.ancho = ancho
        self.alto = alto
        if cop then
            self.nodos = nodos
        else
            self.nodos = np.empty([ancho, alto], dtype=Nodo)
        end
    end
end
```

#### 4.1.5. Grafo

El archivo grafo contiene una clase Graph y otra Vertex. Se ha utilizado la implementación de K Hong, Ph.D. [6]. Es la clase que contiene las conexiones y los costes de los caminos dentro del mapa usado en el algoritmo HPA*.

#### 4.1.6. Tratador de mapas

Este archivo contiene todos los métodos necesarios para a partir de los mapas descargados de Baldur's Gate, transformarlos en archivos compatibles con el framework. En este archivo está el encargado de construir los grafos y por ende los clusters para el mapa de HPA*, haciendo uso del algoritmo de Dijkstra para crear las conexiones dentro de los propios clusters, mediante una búsqueda hacia todos los nodos restantes del cluster. Los métodos usados en el tratador de mapas se aprecian en la figura 8

#### 4.1.7. Ventana y Ventana-UI

Estas clases son las que implementan la interfaz desarrollada con la herramienta Qt y el paquete de Python PyQt5. La clase $VentanaUI$ es la interfaz en sí y la clase $Ventana$ es una

**Algoritmo 8:** Métodos usados para tratar los mapas

```
def mapas()
def leerMapa(nombreFichero)
def crearMapaHPA(nombre, m, grafo)
def divisores(n)
def añadirAlGrafo(grafo, fil, col, n1, n2, clusterActual, traspuesta, clustersPorFila,
  estructuraClusters)
def calculaPuertas2(mapa, mapaPintar, tamañoCluster, indice, pGrande, i0, i1, grafo,
  clusterActual, traspuesta, estructuraClusters)
def añadirConexionesIntraClusters(grafo, numClusters, tamCluster, conexiones, mapa)
```

clase que hereda de la anterior y su finalidad es separar la lógica del diseño. En esta clase se conectan los botones y cuadros para introducir datos con el código, esto se puede observar en la figura 9.

**Algoritmo 9:** Métodos que usa la clase ventana para conectar con el controlador.

```
class MainWindow(QMainWindow, Ui_MainWindow):
    def__init__(self, *args, **kwargs)
    def actualizarJPS(self)
    def actualizarAEstrella(self)
    def actualizarHPA(self)
    def cInseSG(self)
    def cAHPA(self)
    def cHPAJSP(self)
    def cJSP(self)
end
```

### 4.1.8. Controlador

Este archivo contiene métodos que se encargan de ejecutar un determinado algoritmo tras recibir una orden desde la clase Ventana(10), como pueden ser las búsquedas con los diferentes algoritmos (A*, HPA* y JPS) o alguna de las comparaciones contenidas en el framework.

**Algoritmo 10:** Métodos usados por el controlador para ejecutar los algoritmos y las comparaciones.

```
def calcularA(mapa, fI, cIni, fF, cF, comp)
def calcularHPA(mapa, fI, cIni, fF, cF, comp)
def calcularJPS(mapa, fI, cIni, fF, cF, comp)
def creaPuntos(mapa, nPuntos)
def comparacionAHPA(mapa, nPuntos)
def comparacionInseSG(mapa, nPuntos)
def comparacionAJPS(mapa, nPuntos)
```

#### 4.1.9. A*, HPA*, JPS, Dijkstra

Por último, el corazón o corazones del framework, los algoritmos A*, HPA*, JPS y Dijkstra. Como todos los algoritmos son de búsqueda con árboles, la implementación de los diferentes algoritmos es parecida, excepto por detalles puntuales de cada algoritmo, como es la función salto de JPS o la forma de recuperar el camino solución. Se profundizará más en el apartado 4.3.

## 4.2. Visualización de los resultados

En esta sección se mostrarán los resultados obtenidos tras la ejecución de los algoritmos en el framework. En todas la ejecuciones, aparte del camino resultante, aparecerá el coste del camino y el número de iteraciones que ha tenido que hacer el algoritmo.

- Resultado de ejecución del algoritmo A*. Figura 13
- Resultado de ejecución del algoritmo HPA*. Figura 14
- Resultado de ejecución del algoritmo JPS. Figura 15

El framework contiene otra pestaña donde incluye las comparaciones de cada algoritmo con A*. Las gráficas resultantes de la ejecución de estas comparaciones se explicarán en el apartado 5.

- Comparación de A* con HPA*, tanto del coste en tiempo, como en nodos expandidos, el

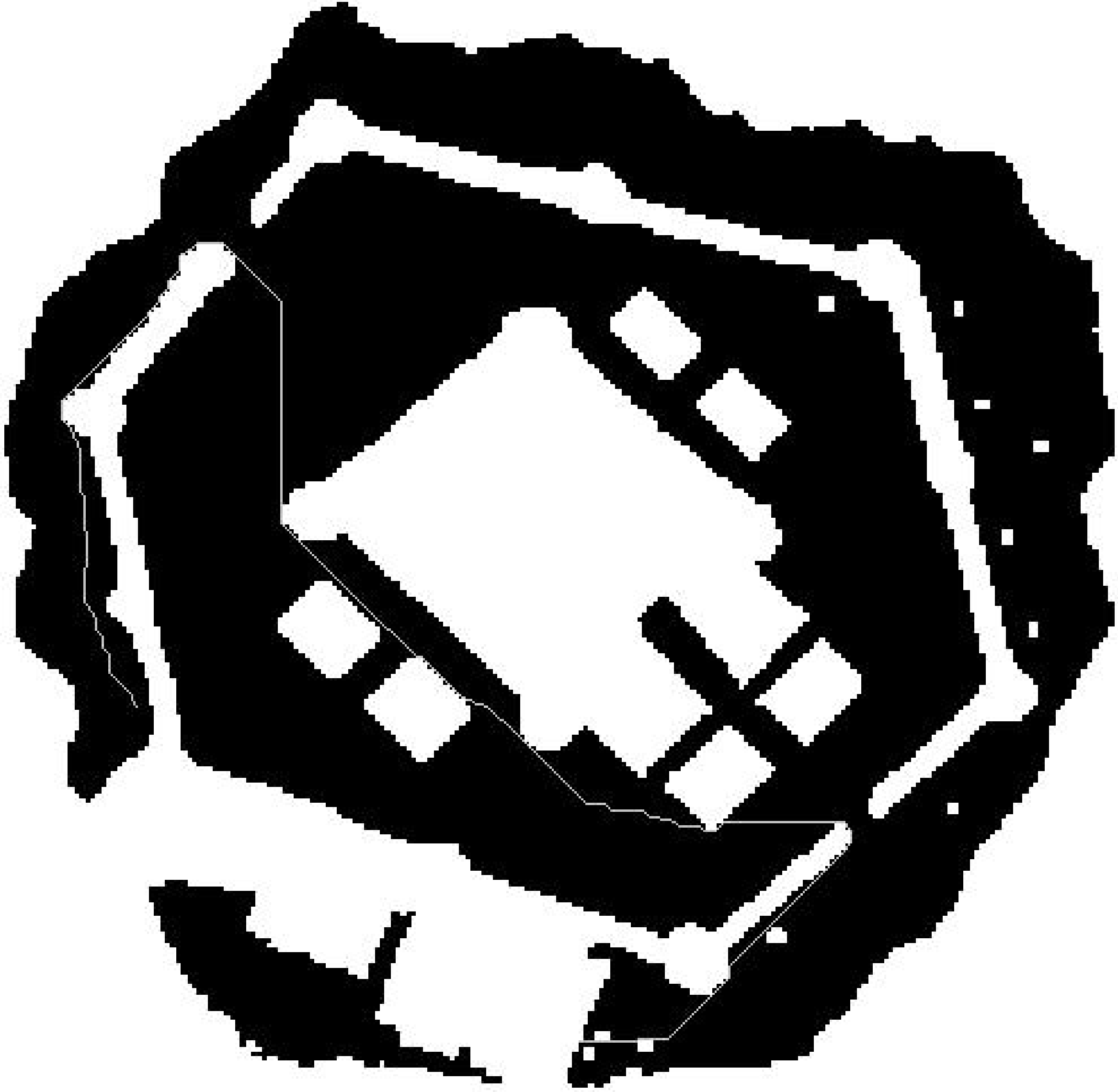


Figura 13: Camino resultante tras ejecutar el algoritmo A*

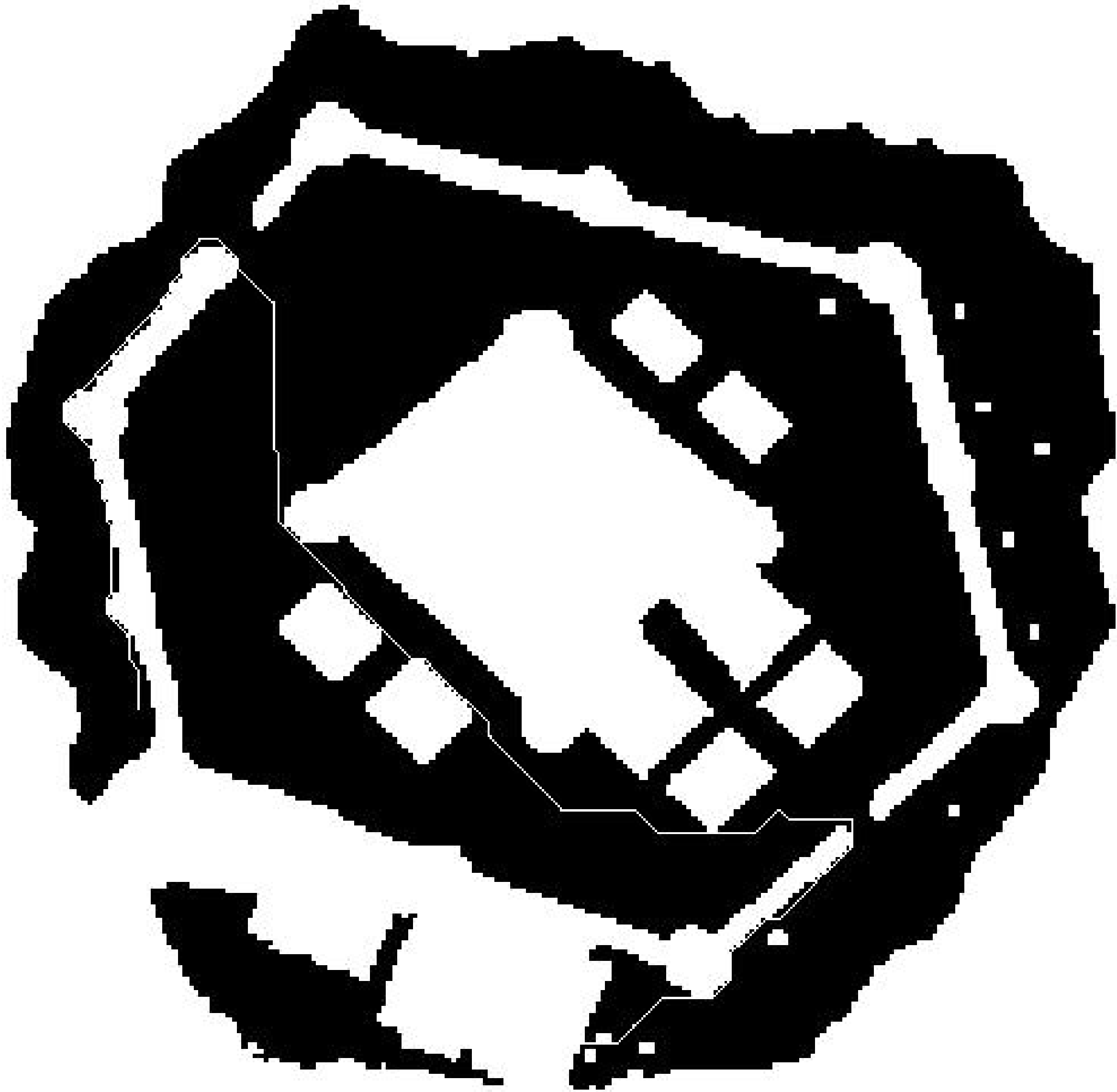


Figura 14: Camino resultante tras ejecutar el algoritmo HPA*

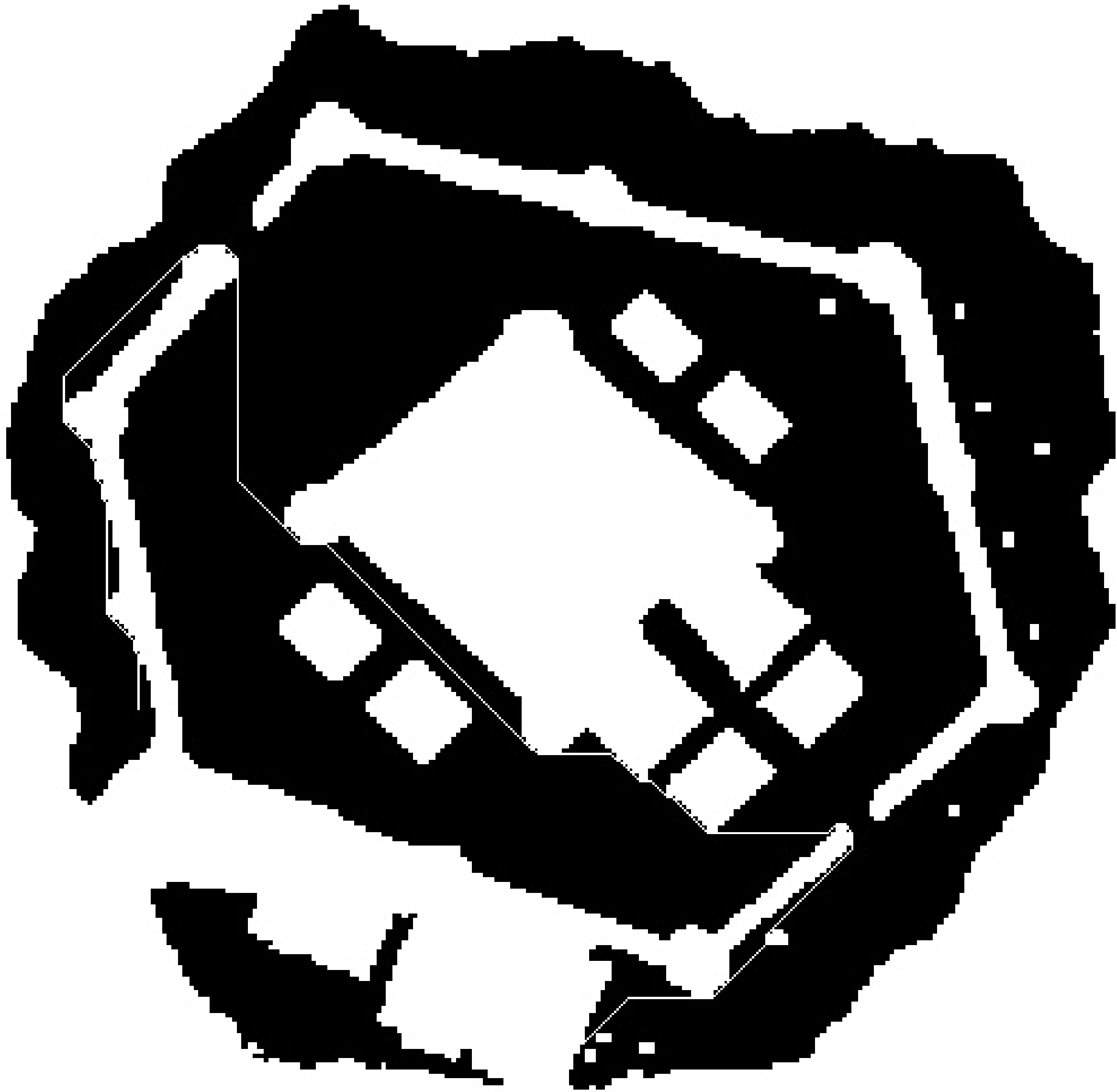


Figura 15: Camino resultante tras ejecutar el algoritmo JPS

error que comete el algoritmo HPA*. Estos resultados se pueden observar en la figura 16.

- Comparación del coste de inserción de los nodos iniciales y finales en el algoritmo HPA*. Pueden observarse los resultados en la figura 17

- Comparación del algoritmo A* con JPS*, tanto en nodos expandidos como en tiempo. Se observa en la figura 18.

- Comparación de HPA* y JPS, en nodos expandidos como en tiempo. El resultado se observa en la figura 19

## 4.3. Detalles de implementación

Como detalles en la implementación, inicialmente se desarrolló una implementación inicial basada en listas, tal y como se describe en los artículos referentes a los algoritmos. Esta implementación era ineficiente debido a su alto tiempo de ejecución, ya que necesitamos tener la lista de abiertos siempre ordenada y sacar nodos concretos de ella por lo que la implementación con listas era funcional pero ineficiente, llegando a ejecuciones de hasta 12 segundos. Posteriormente se realizó una segunda implementación basada en colas con prioridad, realizando las modificaciones que fueran necesarias para su ejecución, llegando a ejecutar las búsquedas en escasos 2 segundos. Esta segunda versión con colas de prioridad, combinados con matrices era más eficiente debido al uso de otras estructuras de datos. En las listas, el acceso a un punto concreto tiene complejidad temporal $O(n)$, para las colas con prioridad es $O(logn)$ y para las matrices $O(1)$. Este no es el punto donde más destaca la segunda implementación ya que como hemos mencionado anteriormente, la lista de abiertos siempre debe estar ordenada. Para la cola con prioridad no es problema ya que inserta los elementos ya ordenados, mientras con el uso de listas, habría que ordenarlas completamente después de cada inserción, es decir, complejidad en tiempo $O(n^2)$. Para llevar a cabo esta segunda implementación me he basado en la implementación de Xueqiao (Joe) Xu [10]. del algoritmo JPS y después modificado para adaptarla a los otros algoritmos y obtener mejores resultados.

Por otra parte, hemos usado números enteros para el cálculo del coste y de la función heurística para ahorrar trabajo en ejecución, ya que los números enteros son mas livianos de

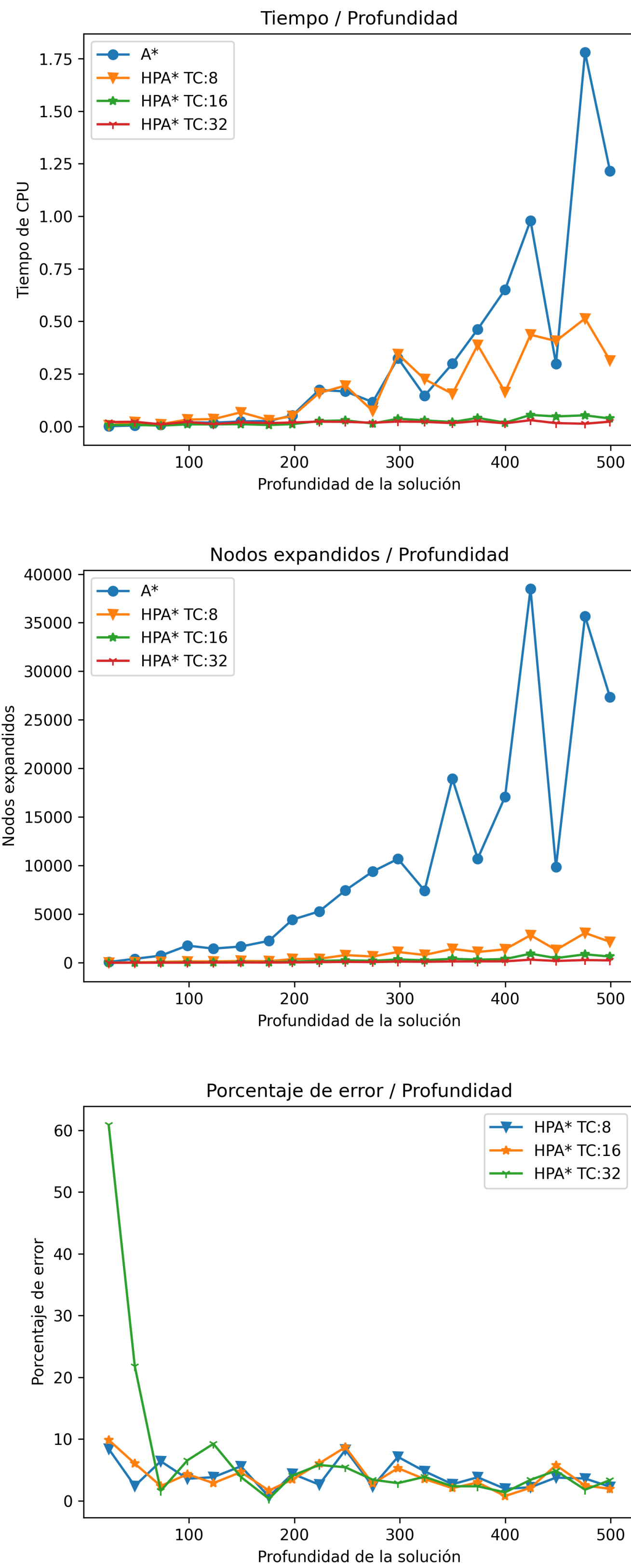



Figura 16: Gráficas resultantes de la ejecución de la comparación entre A* y HPA*.

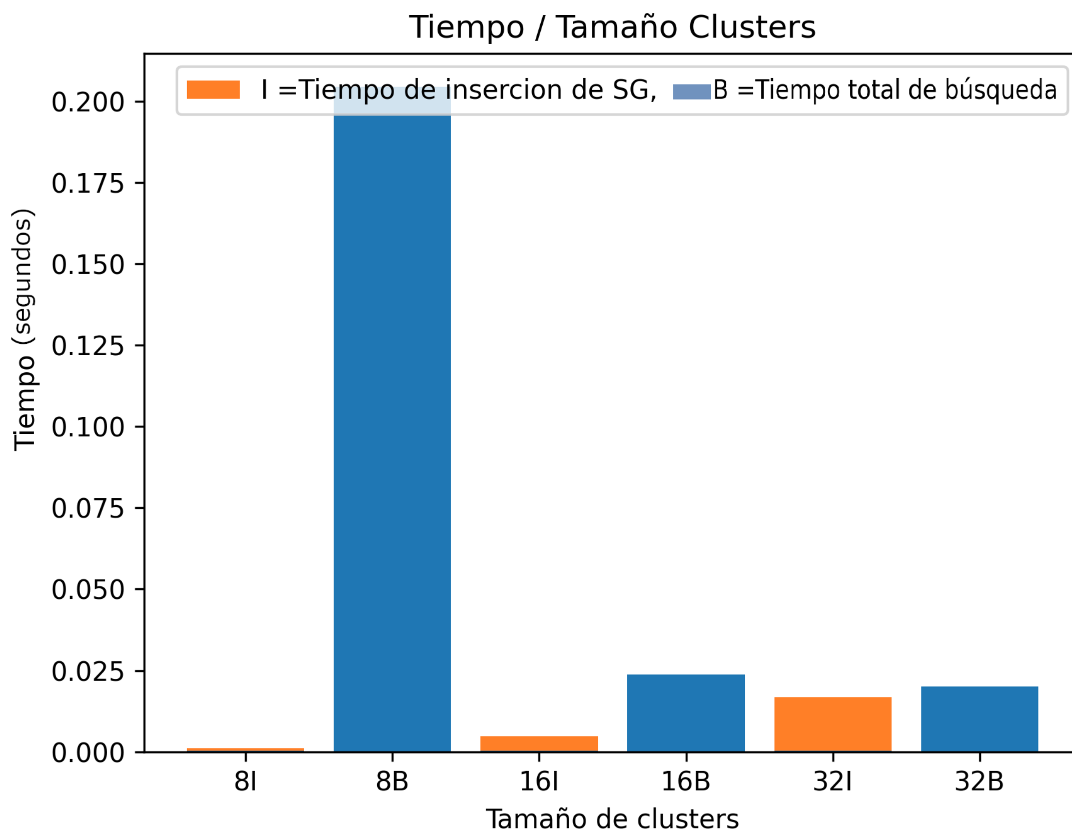
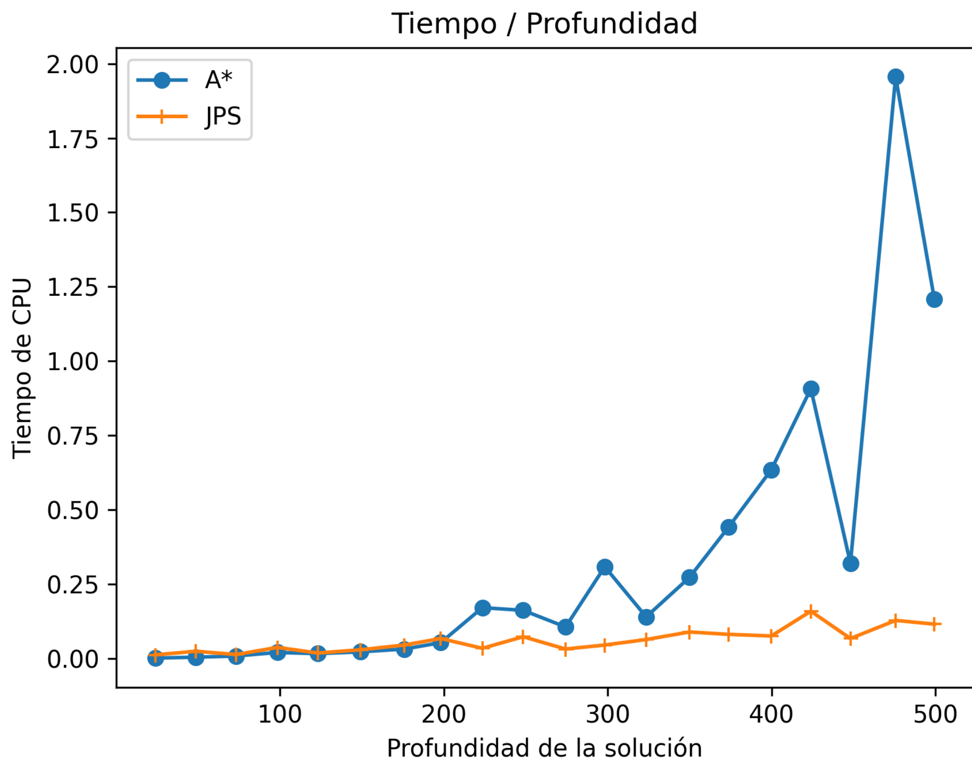



Figura 17: Coste en tiempo resultante de ka inserción de los nodos iniciales y objetivos en el algoritmo HPA*.



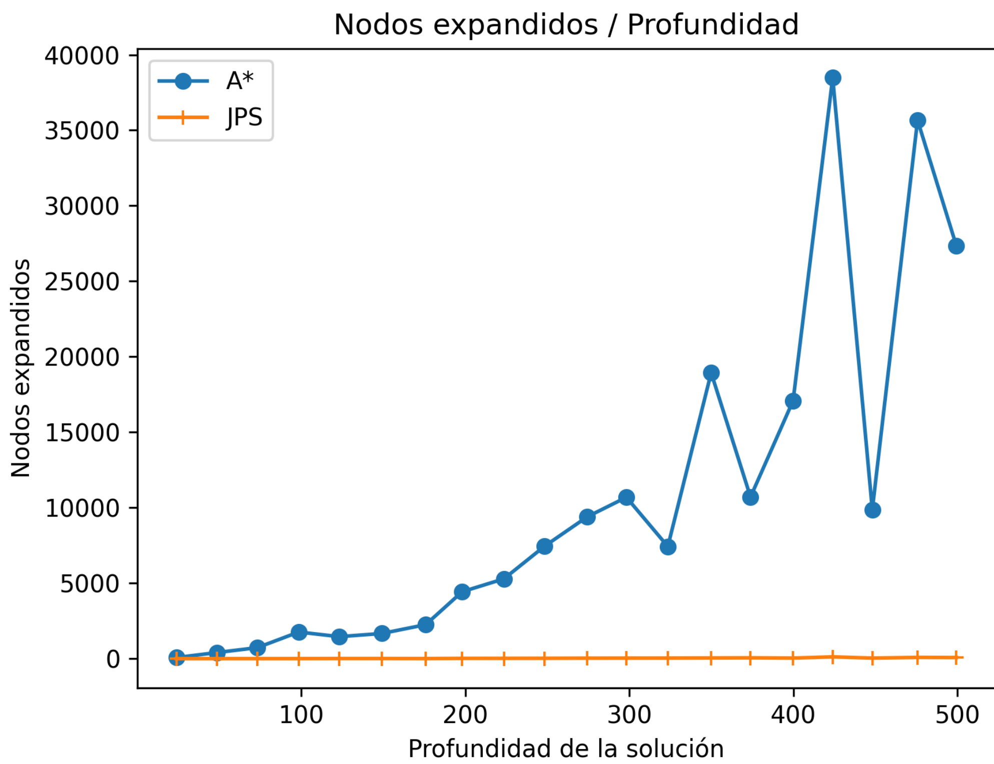
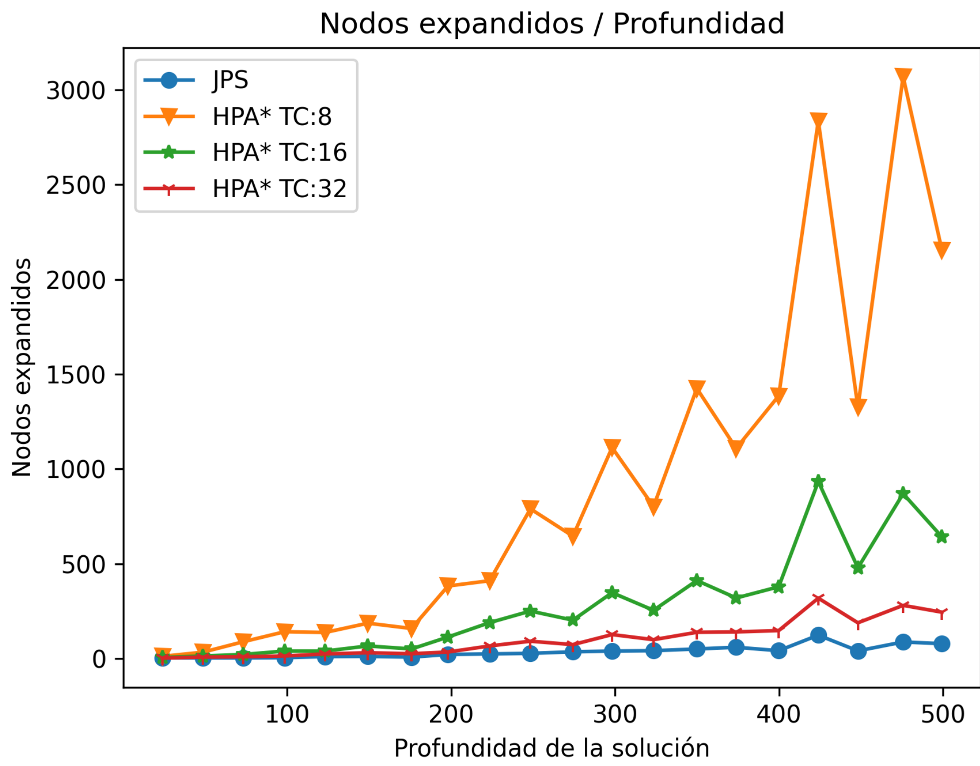




Figura 18: Gráficas resultantes de la ejecución de la comparación entre A* y JPS.



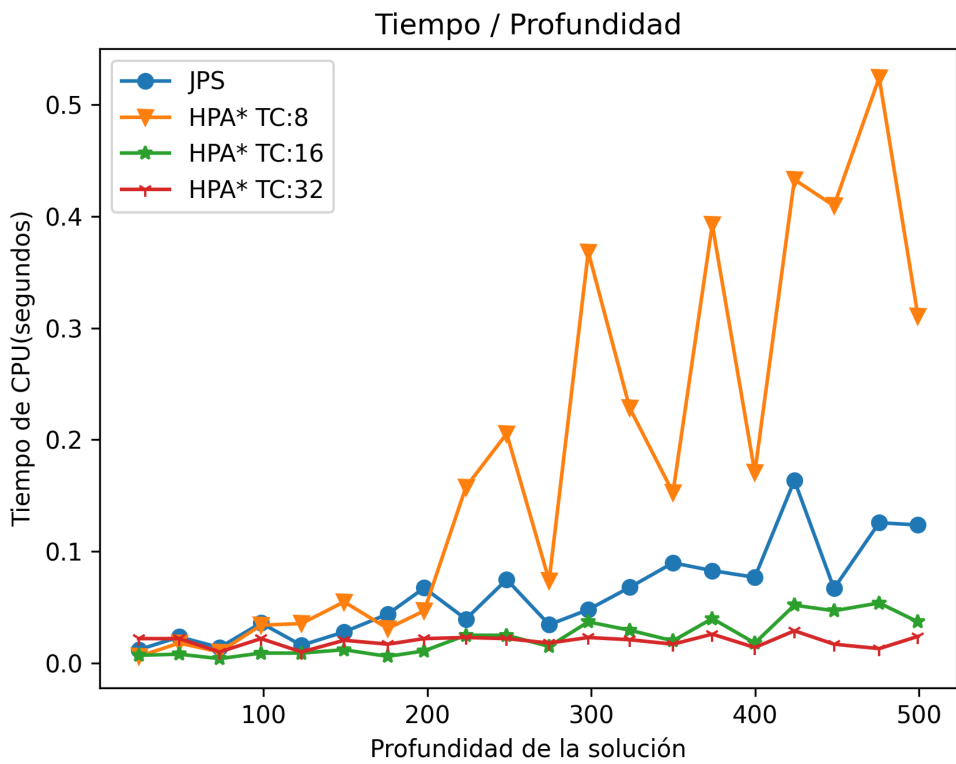




Figura 19: Gráficas resultantes de la ejecución de la comparación entre HPA* y JPS.

tratar que los decimales.

Como se menciono en la sección 4.1.9, todos los algoritmos son similares puesto que todos son algoritmos de búsqueda con arboles y derivan del algoritmo A*. Si tomamos como referencia a A*, podemos encontrar alguna que otra diferencia respecto a los otros algoritmos, pero mayormente en el método que se encarga de calcular los sucesores. En HPA* nuestros sucesores no son los 8 vecinos que tiene un punto en la malla, sino que tenemos que buscar sus sucesores en el grafo abstracto propio de este algoritmo, como se puede observar en la figura **??**. Además HPA* debe insertar los puntos de inicio y final en el grafo para poder realizar la búsqueda, tal y como se observa en la figura 12, mediante el uso del algoritmo de Dijkstra, que es una modificación del algoritmo A* donde nuestra función heurística es 0 y la búsqueda no va hacia 1 solo punto, sino que se hace hacia un conjunto de nodos como se observa en la figura 13.

**Algoritmo 11:** Obtención de los sucesores para HPA*

sucesores = self.mapaHPA.grafoAbstracto.get$_{s}ucesores(nodo.fila, nodo.columna)$

Por su parte JPS es similar en estructura al algoritmo A*, su única diferencia reside en el cálculo de sucesores, que JPS realiza mediante la poda de estos y la función salto expuesta en el apartado 3.3.

**Algoritmo 12:** Inserción de los puntos de inicio y fin en el grafo abstracto.

```
def insertar(self, filaI, colI, filaF, colF) begin
    grafo = self.mapaHPA.grafoAbstracto.copy()
    tamCluster = self.mapaHPA.tamCluster
    numClustersPorFila = self.mapaHPA.dimensiones//self.mapaHPA.tamCluster
    cI = cluster de inicio
    cD = cluster objetivo
    clusterInicio = self.mapaHPA.estructuraCluster[cI].copy()
    clusterDestino = self.mapaHPA.estructuraCluster[cD].copy()
    dijkstra = Dijkstra(self.mapaI)
    camino = dijkstra.findPath(filaI, colI, clusterInicio, (filaI//tamCluster)*tamCluster,
      (colI//tamCluster)*tamCluster, tamCluster)
    if camino != None then
        for r in camino do
            grafo.addedge((filaI, colI), r[2], r[1], r[0])
        end
    end
    dijkstra = Dijkstra(self.mapaG)
    camino = dijkstra.findPath(filaF, colF, clusterDestino, (filaF//tamCluster)*tamCluster,
      (colF//tamCluster)*tamCluster, tamCluster)
    if camino != None then
        for r in camino do
            grafo.addedge((filaF, colF), r[2], r[1], r[0])
        end
    end
end
```

**Algoritmo 13:** Modificación de Dijkstra para conseguir el camino hacia varios puntos.

```
if conjuntoObjetivos.__contains__((nodo.fila, nodo.columna)) then
    caminos.append((utiles.recuperaCamino(nodo), nodo.g, (nodo.fila, nodo.columna)))
    conjuntoObjetivos.remove((nodo.fila, nodo.columna))
    if len(conjuntoObjetivos)==0 then
        return caminos
    end
end
```

# 5

# Análisis de rendimiento

En este capítulo presentaremos los resultados expuestos en el capítulo anterior y los analizaremos con el fin de obtener una conclusión acerca de qué algoritmo es el más rápido.

Como ya hemos comentado en la sección **??** en el capítulo 4, **Diseño e implementación**, hemos planteado los siguientes problemas:

1. Comparación entre A* y HPA*
2. Comparación entre A* y JPS
3. Comparación entre HPA* y JPS

## 5.1. Comparación entre A* y HPA*

Antes de ver los resultados en la comparativa entre estos 2 algoritmos, recordemos la estructura y requisitos de cada uno. A* realiza la búsqueda sobre un grafo definido por una matriz, donde tenemos los puntos accesibles y los obstáculos, haciendo un cálculo de sucesores mediante una lista de abiertos y cerrados alcanzábamos el camino objetivo en el menor número de pasos posible.

Por otro lado, HPA* necesita de un procesamiento previo a la búsqueda, que puede llegar a ser bastante costoso dependiendo del tamaño de los clusters, aunque esto en nuestra comparativa no influye, ya que como hemos mencionado en la explicación del algoritmo, las divisiones de los clusters y sus uniones ya se realizan de forma offline. En primer lugar, el algoritmo tiene que insertar en el grafo abstracto los puntos de salida y objetivo mediante el algoritmo de Dijkstra, para más tarde ejecutar A* sobre el grafo abstracto, lo que hace un total de 3 búsquedas más rápidas y menos costosas que el algoritmo base A*.

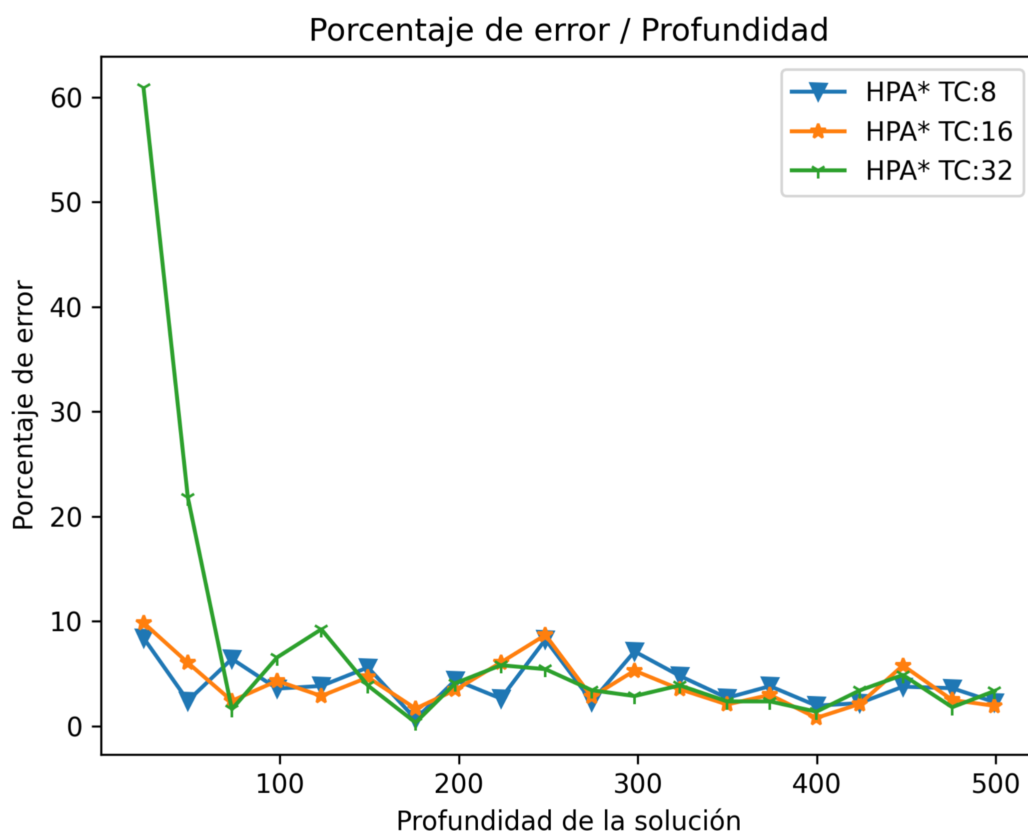



Figura 20: Error entre A* y HPA*

Como hemos dicho, HPA* se ejecuta sobre el grafo abstracto resultado de dividir el mapa en clusters, pero ¿cuál es el tamaño de cluster que queremos usar en nuestra comparativa? Para averiguar esto, necesitamos atender a la figura 20.

Esta gráfica muestra el error en la distancia recorrida respecto a A* que se comete con clusters de distintos tamaños. Se puede apreciar que en distancias de búsqueda pequeñas puede haber grandes errores en tamaños más grandes, ya que todo depende de dónde se encuentren nuestros puntos. Podemos observar que los tamaños 8 y 16 oscilan entre el 0 % de error y el 10 % a diferencia de las búsquedas con tamaño 32, que tiene algunas búsquedas con un error mayor.

En la figura 21 podemos observar por un lado una media del tiempo de inserción de los nodos de origen y destino y por otro lado una media del tiempo de búsqueda total. Como podemos observar, con cluster de tamaño 32, obtenemos tiempos de búsqueda muy bajos, ya que se reduce mucho el número de nodos posibles para expandir. Por otra parte es el que más

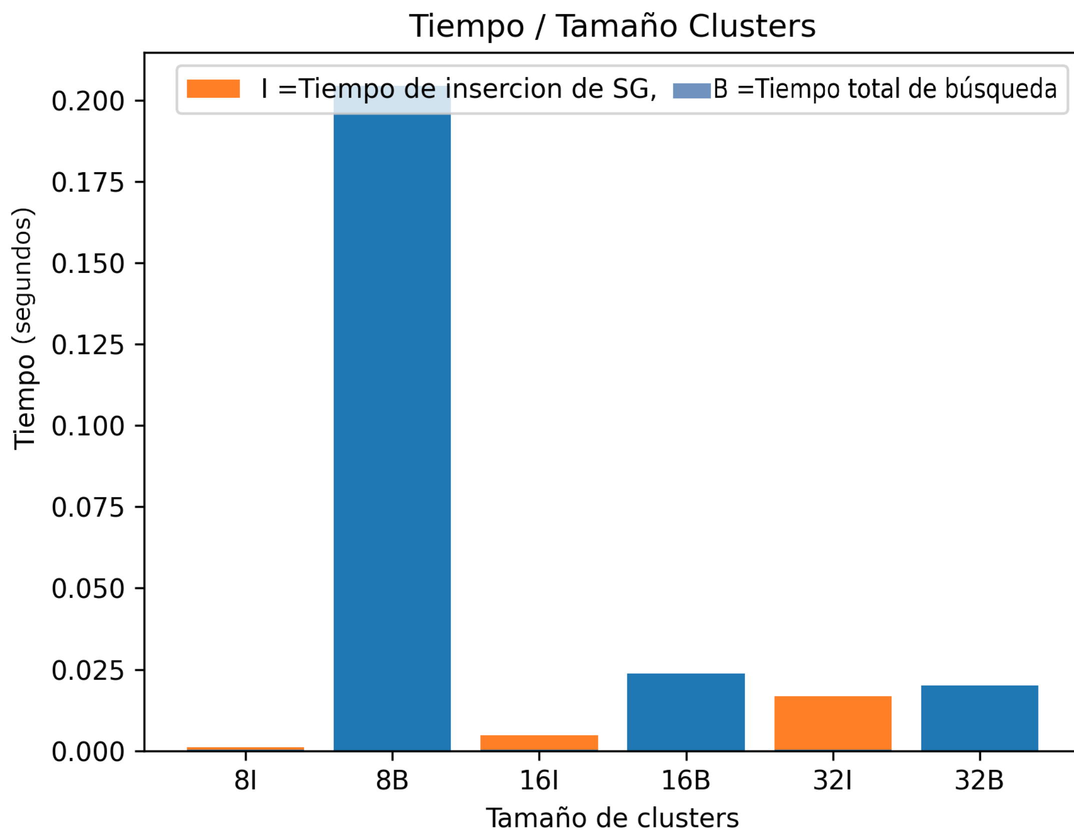



Figura 21: Costes de inserción del nodo objetivo y el nodo inicial.

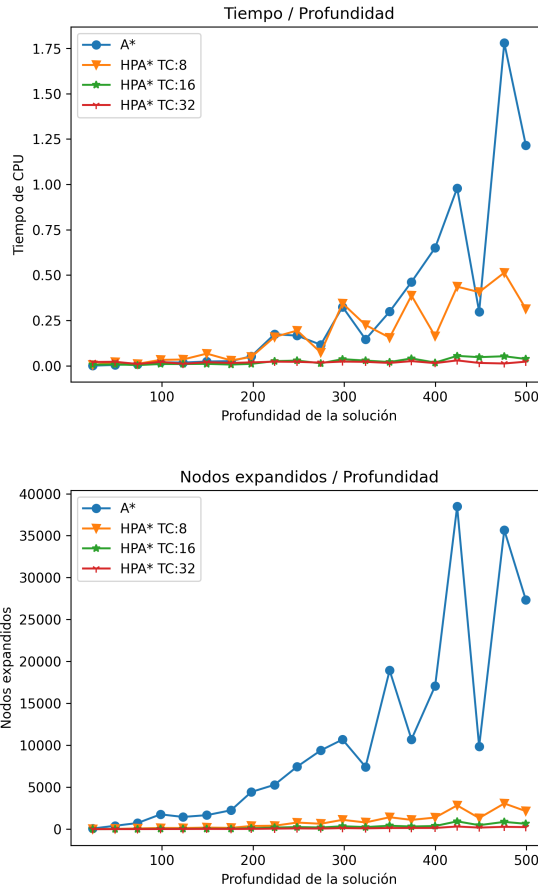



Figura 22: Comparación entre A* y HPA*

tiempo tarda en insertar los nodos objetivo y final, debido a que es el que tiene los clusters más grandes. Al contrario de lo que ocurre con el tamaño de cluster 8, que tarda más en ejecutar la búsqueda completa, pero menos en insertar los nodos. Por último nos encontramos el tamaño de cluster 16, que no es el que menos tarda en la búsqueda final, tampoco el que menos tarda en la inserción de los nodos, pero es el que haciendo una media entre el tiempo de inserción y el de búsqueda tiene mejor resultado.

Para acabar la comparativa, tenemos que realizar la búsqueda en sí, comparando todos los algoritmos. En la figura 22 podemos ver que prácticamente siempre merece la pena usar el algoritmo HPA* en lugar de A*. Observamos como HPA*, con cualquiera de sus variantes, es más eficiente en general que A*, tanto en tiempo de CPU como en nodos expandidos.

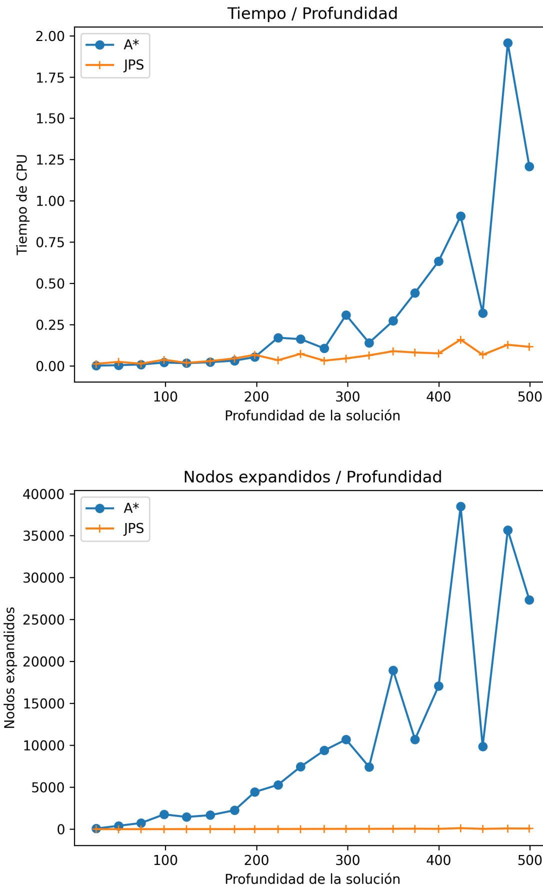



Figura 23: Comparación entre A* y JPS

## 5.2. Comparación entre A* y JPS

Al igual que en la comparativa de la sección 5.1, recordaremos primero brevemente el funcionamiento de JPS y posteriormente analizaremos los resultados. JPS como una modificación de A*, tiene la misma estructura cambiando la función del cálculo de sucesores, que se basaba en ir saltando hacia adelante hasta encontrar puntos concretos llamados puntos de salto, que son los que formarían el conjunto de sucesores del nodo actual y tras esto se sigue ejecutando normalmente el algoritmo A*.

La comparación de estos dos algoritmos es más sencilla que la de los dos anteriores, puesto que JPS mantiene las propiedades y características de A*. Eso quiere decir que encontrará el camino con coste mínimo en el mínimo número de pasos.

Como podemos observar, JPS es más eficiente tanto en tiempo como en nodos expandidos y en ninguna prueba supera los 0.25 segundos de ejecución, por lo que podemos dictaminar

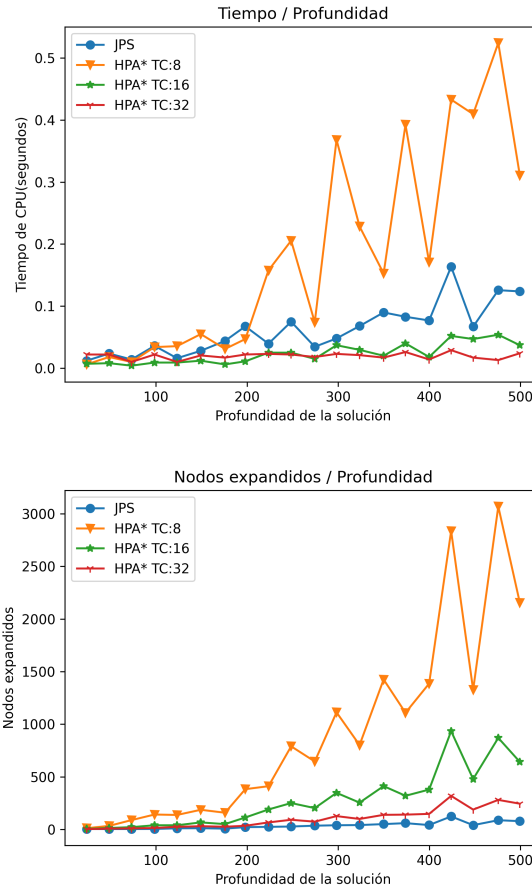



Figura 24: Comparación entre HPA* y JPS

que también es más eficiente que A*, tanto en tiempo como en nodos expandidos, aunque aquí JPS tiene un poco de trampa ya que se tratan de nodos expandidos de la lista de abiertos, lo que no quiere decir nodos visitados, porque al ir saltando JPS, puede llegar a comprobar bastantes más nodos que A*.

## 5.3. Comparación entre HPA* y JPS

La comparación de estos dos algoritmos sigue la línea de la comparativa entre A* y HPA*, puesto que JPS mantiene las propiedades y características de A*. Eso quiere decir que JPS encontrará el camino óptimo y HPA* no, por lo que el error entre estos algoritmos será el mismo que entre A* y HPA*.

Como podemos observar en la figura 24, JPS es el algoritmo que menos nodos expande en su búsqueda, mientras que no es el algoritmo más rápido respecto al tiempo. Mientras que JPS

si es más rápido que HPA* con tamaño de cluster 8, no lo es cuando usamos clusters de tamaño mayor, así que en estas pruebas con mapas de tamaño 512x512 sería más eficiente usar HPA* con clusters de tamaño 16 o 32.

## 5.4. Conclusión de la comparación

Una vez hechas las comparaciones, ¿podemos dictaminar cuál es el mejor algoritmo de búsqueda de los 3 mostrados? La respuesta simple sería sí, el mejor algoritmo es JPS, puesto que aunque en ocasiones sea un poco mas lento que HPA*, es mucho más sencillo de implementar, porque no necesita de un procesamiento previo o un grafo. Pero en realidad no es tan sencillo ya que cada algoritmo tiene usos donde son más útiles que los otros.

HPA* es muy eficiente cuando tenemos mapas muy grandes, pues realizamos una división del mapa en clusters y reducimos el número de nodos posibles considerablemente, pero este algoritmo no tiene sentido cuando nos encontramos en una habitación de 5x5 casillas, pues el coste de implementar HPA* para un mapa tan pequeño es superior al beneficio que obtendríamos si usáramos este algoritmo.

Por su parte, JPS es muy eficiente en mapas de tamaño pequeño o mediano, ya que su rendimiento es bastante superior al de A* en estos escenarios, pero reduce considerablemente su rendimiento en mapas muy grandes pues necesitamos saltar hasta conseguir un punto de salto o toparnos con un obstáculo, por lo que en escenarios muy grandes con muchos espacios abiertos, el tiempo en profundizar los saltos aumentaría, reduciendo así su eficiencia.

Por lo tanto no podemos decidir cual es el mejor algoritmo para el pathfinding, pero sí podemos dar una idea de como sería una implementación eficiente tanto para mapas pequeños como para escenarios enormes. Estamos hablando de una combinación entre HPA*+A* o preferiblemente, HPA*+JPS, la cual el siguiente ejemplo dejará bastante clara.

Imaginemos el mapa del videojuego Skyrim, que se aprecia en la figura 25. Como es un mapa muy grande, usaremos HPA* para la búsqueda de caminos. En la imagen se pueden ver las delimitaciones en rojo, que representarían a los clusters, luego tenemos las puertas marcadas en azul y unas líneas de color cian, que representan los caminos para ir de una puerta a otra. Además, tenemos el camino hacia la capital de cada zona. Con este algoritmo, para viajar de una ciudad a otra tendríamos que realizar la búsqueda sobre un grafo de 36 nodos en lugar de sobre una malla de miles de ellos.

Figura 25: Mapa del videojuego Skyrim, desarrollado por Bethesda [3]

Una vez que hemos llegado a la ciudad que deseamos(figura 26), es el momento de dejar la búsqueda en manos de otro algoritmo como es el caso de A* o JPS, pues se trata de un mapa mucho más pequeño además de no tener que preprocesar el mapa de ninguna ciudad del juego.

Figura 26: Mapa de Falkreath, una ciudad de Skyrim [3]

# 6

# Conclusiones y trabajo futuro

## 6.1. Conclusiones

En este proyecto se ha abordado el problema del pathfinding o búsqueda de caminos en videojuegos. Hemos estudiado diferentes algoritmos para abordarlo y poder sacar una idea de qué algoritmo es el mejor, facilitando su uso mediante una interfaz y un manual de usuario, cumpliendo así los objetivos planteados.

Los objetivos eran estudiar e implementar los algoritmos A*, HPA* y JPS, desarrollar un framework para la importación de los mapas y su uso y la realización de un manual de usuario explicando su uso y que facilitara la incorporación de nuevos algoritmos.

Como se ha comentado en el apartado de análisis de los resultados (5), se aprecia que HPA* es teóricamente el algoritmo más rápido en este tipo de pruebas, pero que realmente HPA* y JPS tienen finalidades diferentes, pues uno tiene mejor desempeño en mapas grandes y otro en mapas pequeños, por lo que se dio un posible uso eficiente para la búsqueda. Sería el uso de HPA* en espacios grandes y JPS en espacios pequeños.

Los resultados de las comparativas han sido sorprendentes desde mi punto de vista. A lo largo de la realización del proyecto y del estudio de los algoritmos he tenido bastantes sorpresas, primero con el uso de las colas con prioridad que mejora sustancialmente el rendimiento de los algoritmos y posteriormente con el resultado entre JPS y HPA*. HPA* tendrá un tiempo relativamente constante mientras que JPS depende mucho del tipo de mapa y su disposición llegando a ser en ocasiones más lento que HPA*

A nivel de aprendizaje, este proyecto me ha enseñado a buscar información de diferentes ámbitos, desde métodos e ideas para la optimización de los algoritmos hasta documentación

necesaria para abordar el uso de diferentes herramientas y librerías para la implementación de los algoritmos, ya que hasta ahora no lo había necesitado mas allá de lo proporcionado en los manuales de los lenguajes. Además del uso de un nuevo lenguaje para mi como es Python o la herramienta LaTeX. Pero lo más importante a leer documentación acerca de los algoritmos que hemos estudiado que es muy diferente a leer un manual de un paquete de Python y me ha ayudado a comprender mejor otros artículos que he leído tras la realización del TFG.

## 6.2. Trabajo futuro

Tras la realización de la comparativa entre los diferentes algoritmos han surgido nuevas líneas de estudio. Existen algunas mejoras del algoritmo JPS, por lo que un próximo trabajo sería la implementación de alguna mejora para este algoritmo. Además de seguir implementando diferentes algoritmos en el framework, como por ejemplo algoritmos de backtracking.

Como se ha expuesto en las conclusiones no hemos podido determinar cuál es el mejor algoritmo de búsqueda para cualquier escenario, sin embargo se ha expuesto la idea del uso combinado de HPA* y JPS, por lo que un trabajo futuro sería un estudio y comparativa más a fondo entre estos algoritmos, para ver si esta idea realmente sería funcional.

También existe la posibilidad de implementar una mejora en el framework para que realice búsquedas en tiempo real.

Otro posible trabajo futuro sería la ampliación de los algoritmos para realizar búsquedas en espacios tridimensionales e incluso la posibilidad de crear los mapas dentro de la aplicación.

# Referencias


[1] Adi Botea, Martin Muller y Jonathan Schaeffer. "Near optimal hierarchical path-finding". En: (2004).

[2] Pokemon fandom. *Imagenes del videojuego pokemon desarrollado por The pokemon company*. URL: https://pokemon.fandom.com/es/wiki/Ciudad_Verde.

[3] The elder scrolls fandom. *Imagenes del videojuego Skyrim desarrollado por Bethesda*. URL: https://elderscrolls.fandom.com/es/wiki/The_Elder_Scrolls_V:_Skyrim.

[4] Gamedev.stackexchange.com. *Imagen división clusters HPA**. 2017. URL: https://gamedev.stackexchange.com/questions/135967/hpa-pathfinding-building-the-hierarchical-graph-is-too-slow.

[5] Daniel Harabor y Alban Grastien. "Online graph pruning for pathfinding on grid maps". En: (2011).

[6] Ph.D K Hong. *Python Tutoial: Graph Data Structure*. URL: https://www.bogotobogo.com/python/python_graph_data_structures.php.

[7] Stuart Russell y Peter Norvig. *Artificial intelligence: a modern approach*. 2009.

[8] N. Sturtevant. "Benchmarks for Grid-Based Pathfinding". En: *Transactions on Computational Intelligence and AI in Games* 4.2 (2012), págs. 144-148. URL: http://web.cs.du.edu/~sturtevant/papers/benchmarks.pdf.

[9] Wikiwand. *Imagen del coste del movimiento en mallas cuadradas y hexagonales*. URL: https://www.wikiwand.com/en/Hex_map.

[10] Xueqiao (Joe) Xu. *Pathfinding.js*. 2017. URL: https://github.com/qiao/PathFinding.js.

# Apéndice A
# Manual de uso

Este es un manual de uso del framework desarrollado, contiene 2 partes, la primera es un manual de uso y la segunda es una explicación del funcionamiento interno del mismo, donde explicaré como habría que introducir nuevos algoritmos y funcionalidades.

## A.1. Manual de uso

Este apartado es un manual de uso de la interfaz del framework, que se ha desarrollado a través de la aplicación Qt.

La aplicación se divide en pestañas, una para cada algoritmo y una última pestaña donde se encuentran las comparaciones como se puede ver en la figura 27.

Las pestañas de ejecución de los algoritmos son todas iguales y contienen campos para introducir las coordenadas de los puntos que queremos hacer la búsqueda, una imagen del camino resultante de la búsqueda, los datos de ejecución(nodos expandidos, longitud y coste) y por último el botón para ejecutar la búsqueda. Se puede observar en la figura 27

La última pestaña se puede observar en la figura 28 y es la relativa a las comparaciones, donde actualmente están contenidas:

1. Comparación entre los algoritmos A* y HPA*.

2. Comparación entre los algoritmos A* y JPS.

3. Comparación entre los algoritmos HPA* y JPS.

4. Coste de inserción y búsqueda para distintos tamaños de clusters en HPA*.

## A.2. Funcionamiento interno

### A.2.1. Funcionamiento de la interfaz

Como hemos mencionado antes, la interfaz está hecha con la herramienta Qt, aunque podemos usar la aplicación o modificar a nivel de código en la clase ventana_ui.py.

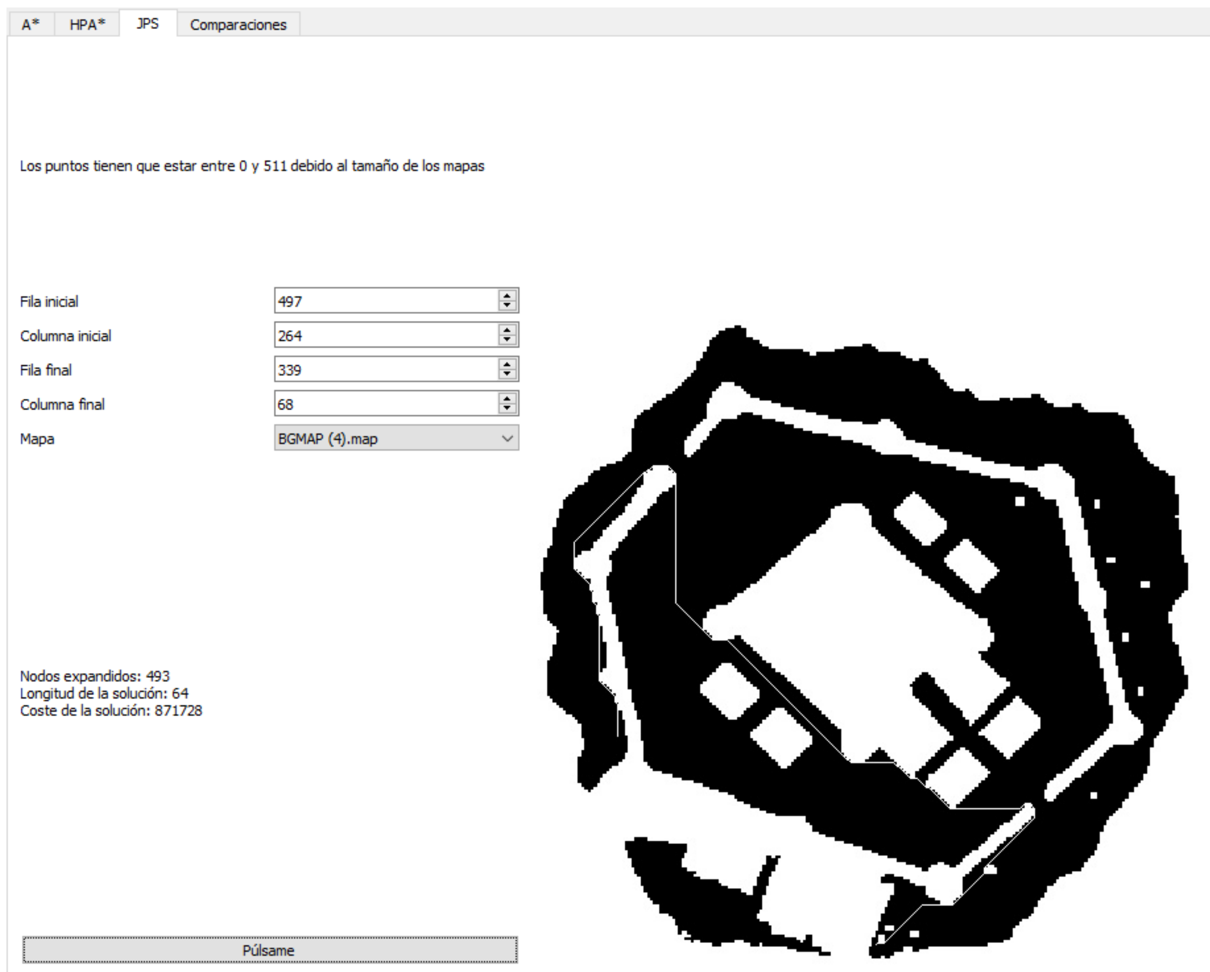



Figura 27: Ejemplo de pestaña de ejecución de uno de los algoritmos

Dependiendo del mapa, el numero de puntos posibles puede variar.

**Costes de inserción de los puntos de salida y destino en los grafos para HPA***

Mapa BGMAP (1).map Nº puntos 20 Separación 25 Ejecutar

**Comparación entre A* y HPA***

Mapa BGMAP (1).map Nº puntos 20 Separación 25 Ejecutar

**Comparación entre A* y JPS**

Mapa BGMAP (1).map Nº puntos 20 Separación 25 Ejecutar

**Comparación entre JPS y HPA***

Mapa BGMAP (1).map Nºpuntos 20 Separación 25 Ejecutar

Figura 28: Ejemplo de pestaña de comparaciones de los algoritmos

Para simplificar el código y separar el diseño de la interfaz de este, su diseño y su funcionamiento están separados en dos clases. La clase ventana_ui.py que es un cascarón vacío sin funcionalidad y la clase ventana.py donde se implementan los métodos y se le da funcionalidad a los botones y demás elementos de la interfaz. Esto se puede observar en la figura 14. Estas líneas establecen el texto que se verá en la pestaña de ejecución de JPS previa a la búsqueda y conecta el click del botón con el método $actualizarJPS()$

**Algoritmo 14:** Ejemplo de conexión de botones y textos en la interfaz

```
self.datosJPS.setText("Haz click en el botón para ejecutar JPS")
self.ejecutaJPS.setText("Púlsame")
self.ejecutaJPS.clicked.connect(self.actualizarJPS)
```

Existen métodos que se encargan de llamar a la ejecución del algoritmo, como es el caso del código que se observa en la figura 15. Con métodos propios de la librería PyQt5 conseguimos los valores introducidos y el mapa sobre el que queremos realizar la búsqueda. Además comprobamos en este momento si los puntos introducidos son puntos libres de obstáculos del mapa para realizar o no la búsqueda. No necesitamos comprobar si los puntos están dentro del mapa puesto que la propia interfaz ya limita los puntos introducidos a valores entre 0 y 511.

### A.2.2. Estructura de los algoritmos

Los algoritmos de búsqueda que hemos escogido son todos algoritmos de búsqueda en arboles, por lo que para cualquier algoritmo similar el esquema será similar al que se observa en la figura 16 .Los métodos $calculaSucesores(17), vecinos, devolverSol(18)$, cambian dependiendo del algoritmo que estemos programando, aunque para los 3 implementados son relativamente similares.

En caso que se quisiese implementar un algoritmo de búsqueda con retroceso, habría que diseñar un nuevo esquema. Si se quisiese seguir el estilo, tendría que existir una clase para el algoritmo, un constructor donde se le introduce el mapa, el método $findPath$ para realizar la búsqueda y $devolverSol$ que es el método que se encarga de devolver todos los datos relativos a la ejecución del algoritmo para que después, el controlador los muestre por pantalla.

**Algoritmo 15:** Ejemplo de método para la ejecución de la búsqueda con un algoritmo

```
self.datosJPS.setText(”Ejecutando algoritmo, por favor espere”)
posMapa = self.mapasJPS.currentIndex()
fI=self.filaIJPS.value()
cI=self.columnaIJPS.value()
fF=self.filaFJPS.value()
cF=self.columnaFJPS.value()
if self.listaA[posMapa].matriz[fI, cI]=='.' and self.listaA[posMapa].matriz[fF, cF]=='.' then
    escribir = calcularJPS(self.listaA[posMapa], fI, cI, fF, cF, False)
    if escribir == None then
        self.datosJPS.setText(”Error, no se encuentra camino entre los puntos introducidos”)
    else
        self.datosJPS.setText(escribir[0])
        self.mapaJPS.setPixmap(QtGui.QPixmap(escribir[1]))
    end
else if self.listaA[posMapa].mapa[fI, cI]=='.' then
    self.datosJPS.setText(”Error, el punto de destino no esta en una posición válida”)
else
    self.datosJPS.setText(”Error, el punto de origen no esta en una posición válida”)
end
```

**Algoritmo 16:** Clase y algoritmo básico de búsqueda A*

```
class AEstrella() :
    abiertos: list()
    mapa: Mapa
    def __init__(self, mapa) :
        self.mapa = mapa
        self.abiertos = []
        self.heuristico = heuristicos.octil
    end
    def findPath(self, filaI, colI, filaF, colF) :
        self.it=0
        self.nodoInicio = self.mapa.getNodoAt(filaI,colI)
        self.nodoObjetivo = self.mapa.getNodoAt(filaF,colF)
        self.nodoInicio.g = 0
        self.nodoInicio.f = 0
        heapq.heappush(self.abiertos,(self.nodoInicio.f,self.nodoInicio))
        self.nodoInicio.abierto = True
        while (len(self.abiertos) >0) do
            self.it+=1
            nodo = heapq.heappop(self.abiertos)[1]
            nodo.cerrado = True
            if (nodo == self.nodoObjetivo) then
                return self.devolverSol()
            self.calculaSucesores(nodo)
        end
        return []
    end
end
```

**Algoritmo 17:** Calculo de sucesores para A*

```
def calculaSucesores(self, nodo) :
    filaF = self.nodoObjetivo.fila
    colF = self.nodoObjetivo.columna
    x = nodo.columna
    y = nodo.fila
    vecinos = self.vecinos(nodo)
    for i in range(len(vecinos)) do
        neighbor = vecinos[i]
        jx = neighbor[1]
        jy = neighbor[0]
        sigNodo = self.mapa.getNodoAt(jy, jx)
        if sigNodo.cerrado then
            continue
        if abs(y-jy)==1 and abs(x-jx)==1 then
            ng = nodo.g + 1414
        else
            ng = nodo.g + 1000
        end
        if not sigNodo.abierto or ng <sigNodo.g then
            sigNodo.g = ng
            sigNodo.h = sigNodo.h or self.heuristico(abs(jy - filaF), abs(jx - colF))
            sigNodo.f = sigNodo.g + sigNodo.h
            sigNodo.padre = nodo
            if not sigNodo.abierto then
                heapq.heappush(self.abiertos, (sigNodo.f, sigNodo))
                sigNodo.abierto = True
            else
                for i in range(len(self.abiertos) do
                    if self.abiertos[i][1] == sigNodo then
                        self.abiertos[i]=(sigNodo.g-sigNodo.f,sigNodo)
                end
            end
    end
end
```

**Algoritmo 18:** Función que devuelve la solución con todos los datos necesarios

```
def devolverSol(self) :
    listaSol = utiles.recuperaCamino(self.nodoObjetivo)
    datos =”Nodos expandidos: -str(self.it)+”-”Longitud de la
      solución:-str(len(listaSol))+”-Çoste de la solución: - str(self.nodoObjetivo.g)
    return listaSol, datos, self.nodoObjetivo.g, self.it
end
```

### A.2.3. Llamando a la ejecución de los algoritmos mediante la clase controlador

Sabemos que la clase ventana ejecuta unos métodos(15) que le proporcionan lo necesario a la clase controlador para que esta ejecute los algoritmos que queramos. Un ejemplo de estos métodos de ejecución de los algoritmos en la clase controlador se puede observar en la figura 21

Los métodos que ejecutan los algoritmos crean un objeto del tipo de algoritmo y mapa que queramos utilizar, además es el que se encarga de calcular el tiempo de ejecución de estos.

Como este framework tiene una pestaña para comparar la ejecución de los algoritmos y realiza muchas búsquedas, tiene la opción de solamente ejecutar los algoritmos sin guardar en un fichero la ejecución de estos lo que hace que la ejecución sea más rápida.

### A.2.4. Útiles, heurísticos y otras clases usadas en los algoritmos

Como hemos dicho anteriormente en la sección 4.1, el archivo Útiles tiene almacenado actualmente un solo método, pero esta pensado para ir añadiendo toda clase de métodos genéricos que ayuden a la ejecución de los diferentes algoritmos que se implementen.

En la clase heurísticos se podrán añadir toda clase de heurísticos que se quieran implementar para probar con los algoritmos.

Tratador de mapas, es el script que se encarga de transformar los archivos .map de Baldur’s Gate II a una lista que usamos en el framework por lo que si necesitamos añadir distintos mapas será aquí donde implementemos los algoritmos necesarios para leerlos.

Por último las clases $Nodo$ y $Mapa$ son clases necesarias para la ejecución de los algoritmos implementados pero también son apropiadas para otro tipo de algoritmos, por lo que se

**Algoritmo 19:** calcularA(mapaI, fI, cIni, fF, cF, comp). Ejemplo de método perteneciente a la clase controlador que se encarga de ejecutar los algoritmos de búsqueda.

```
startTime = time.time()
aestrella = AEstrella(mapaI)
listaSol,datos, coste, it=aestrella.findPath(fI,cIni,fF,cF)
endTime = time.time()-startTime
if listaSol != None then
    if comp==False then
        mapaEscribir = mapaI.matriz.__copy__()
        for i in listaSol do
            mapaEscribir[i[0], i[1]]="X"
        end
        f= open("salidas/solucionA.txt", "w")
        for i in range(0, 512) do
            f.write(".join(mapaEscribir[i,:]))
        end
        f.close()
        mapa2 = np.empty([512, 512], dtype=int)
        for i in range(0, 512) do
            for j in range(0, 512) do
                if mapaEscribir[i,j]=='@' then
                    mapa2[i,j]=0
                else if mapaEscribir[i,j]=='.' then
                    mapa2[i,j]=255
                else
                    mapa2[i,j]=ord(mapaEscribir[i,j])
            end
        end
```

```
    plt.imsave("salidas/imagenA.jpg", mapa2)
    ventana=[datos, "salidas/imagenA.jpg"]
    return ventana
    else
        return len(listaSol), coste, it, endTime
    end
else
    return None
end
```

pueden usar también en algoritmos como IDA*, que es un algoritmo de búsqueda con retroceso.

### A.2.5. Implementando un nuevo algoritmo en el framework

Imaginemos que queremos implementar un nuevo algoritmo en nuestro framwework simplemente para ejecutarlo y no compararlo. Lo primero que tenemos que hacer es ejecutar la aplicación Qt para añadir lo necesario en la interfaz.

Pulsamos botón derecho en la pestaña que queremos insertar y en el menú desplegable → Intert Page → After current page, esto se puede apreciar en la figura 29. Con esto se nos añadirá una nueva pestaña tras la actual donde añadiremos todo lo necesario para la ejecución del algoritmo. Se añadirán los elementos que se necesiten para la ejecución del algoritmo. En este caso se han añadido varios textos, 4 cuadros donde poner las coordenadas, un botón, etc. Tras esto se distribuirá el layout para dejarlo similar a las otras pestañas como se observa en la figura 30.

Tras añadir y acomodar todos los elementos, necesitamos cambiar el nombre de los objetos que hemos añadido para poder usarlos cómodamente dentro del proyecto. Una vez guardados los cambios tenemos que ejecutar el siguiente comando en la consola para transformar nuestro archivo .ui a formato python: "pyuic5 -x ventana.ui -o ventana_ui.py".

Ahora que tenemos la interfaz lista podemos ponernos a añadir el código. Lo primero que tenemos que hacer es programar el algoritmo en concreto, este ejemplo será con el algoritmo de

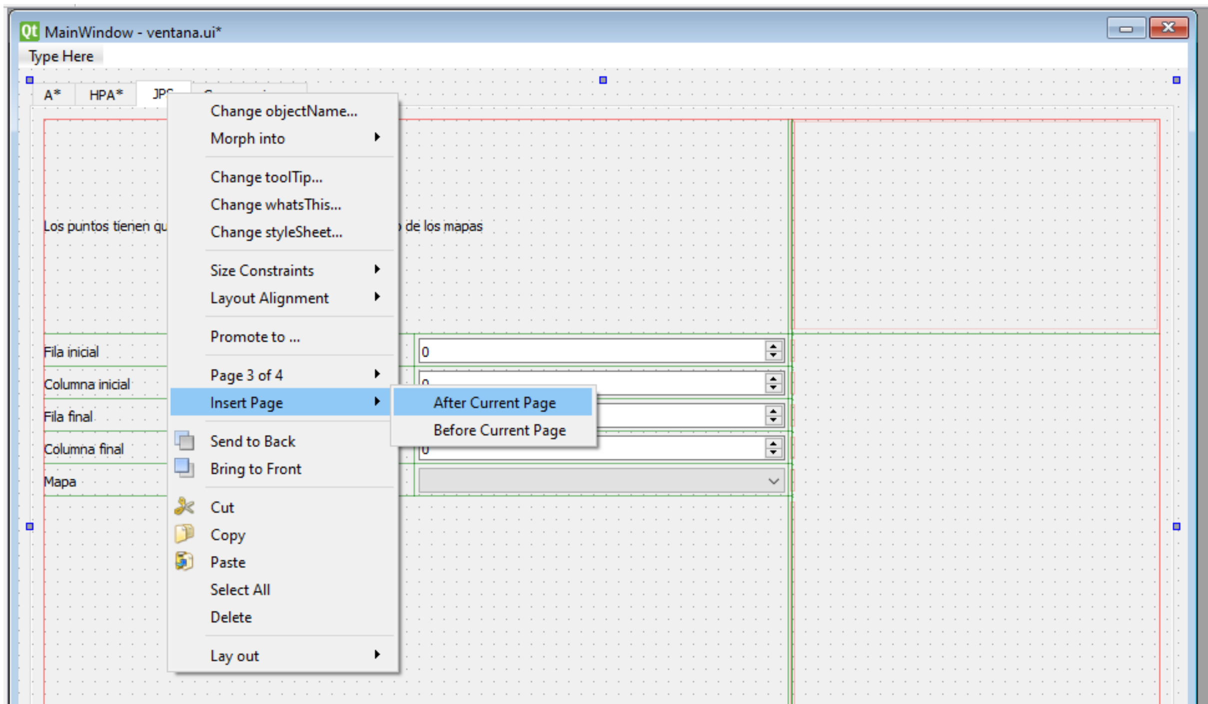



Figura 29: Añadir pestaña en la interfaz mediante Qt

Dijkstra que tenemos programado para su uso con HPA*, pero que para este ejemplo podemos usarlo.

No entraré en detalle de la implementación del algoritmo pues no es importante para esta explicación, lo que si es importante es la forma en la que devolvemos los resultados, por ejemplo, podemos devolver el número de iteraciones que hace el algoritmo, el coste final del camino y el camino en sí.

Ahora que tenemos la interfaz preparada y el algoritmo implementado toca hacer los métodos necesarios para ejecutarlo a través de la aplicación. Lo primero que tenemos que hacer es entrar en el archivo *ventana.py*, aquí en el constructor tenemos que conectar todo lo que tenemos en la interfaz con el código. En este caso se añadirían las lineas de la figura 22. DatosD y ejecutaD son los nombres de los objetos que hemos añadido en la interfaz, con la tercera línea conectamos el botón con el método que luego ejecutará el algoritmo.

```
self.datosD.setText(”Haz click en el botón para ejecutar Dijkstra”)
self.ejecutaD.setText(”Púlsame”)
self.ejecutaD.clicked.connect(self.actualizarD)
```

El método actualizarD se muestra en la figura 20. Tenemos que tener en cuenta que este

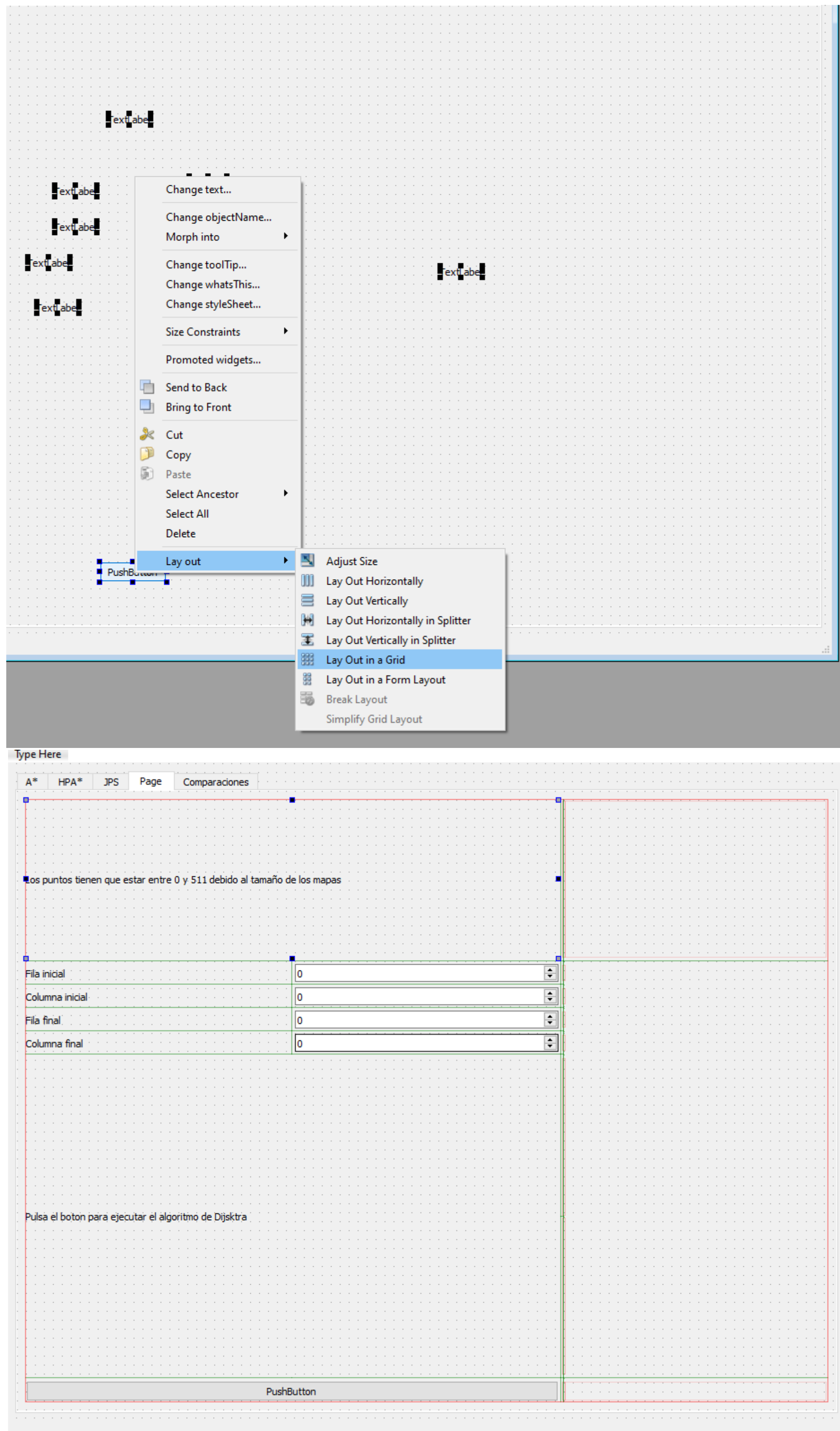



Figura 30: Añadir elementos y cambiar layout de la pestaña

método le mandará los datos necesarios a otra que tenemos que crear en la clase controlador para que esta ejecute el algoritmo correctamente, además tenemos que saber que ese otro método nos devolverá dos cosas, los datos de ejecución y una imagen del recorrido.

**Algoritmo 20:** ActualizarD, contenido en la clase ventana llama al controlador para ejecutar los algoritmos

```
En algunas líneas se ha simplificado el código para mejorar su legibilidad.
self.datosD.setText(”Ejecutando algoritmo, por favor espere”)
posMapa = self.mapasD.currentIndex()
fI=self.filaID.value()
cI=self.columnaID.value()
fF=self.filaFD.value()
cF=self.columnaFD.value()
if fI,cI,fF,cF están dentro del mapa y no son obstáculos then
    escribir = calcularD(self.listaA[posMapa], fI, cI, fF, cF)
    if escribir == None then
        self.datosD.setText(”Error, no se encuentra camino entre los puntos introducidos”)
    else
        self.datosD.setText(escribir[0])
        self.mapaD.setPixmap(QtGui.QPixmap(escribir[1]))
    end
else if self.listaA[posMapa].mapa[fI, cI]==’.’ then
    self.datosD.setText(”Error, el punto de destino no está en una posición válida”)
else
    self.datosD.setText(”Error, el punto de origen no está en una posición válida”)
end
```

Una vez tenemos listo este método nos queda la última parte, crear lo necesario en la clase controlador. La siguiente función contenida en la clase controlador y que se aprecia en la figura 21 es la que se encarga de crear el objeto del algoritmo, y mandarle lo necesario para que realice la búsqueda. Posteriormente se encargará de guardar en un archivo y una imagen el camino resultante de la búsqueda. Con este método listo solo nos queda abrir la aplicación y ejecutar el algoritmo para comprobar que todo ha ido correctamente.

**Algoritmo 21:** Método calcularD(mapaIntroducido, fI, cIni, fF, cF), que se encarga de ejecutar el algoritmo.

```
startTime = time.time()
dijkstra = Dijkstra(mapaIntroducido)
resultado = dijkstra.findPath(fI, cIni, fF, cF)
endTime = time.time() - startTime
if resultado != None then
    it, coste, camino = resultado
    mapaEscribir = mapaIntroducido.matriz.__copy__()
    for i in camino do
        mapaEscribir[i[0], i[1]]="X"
    end
    f= open("salidas/solucionD.txt", "w")
    for i in range(0,512) do
        f.write(".join(mapaEscribir[i,:]))
    end
    f.close()
    mapa2 = np.empty([512, 512], dtype=int)
    for i in range(0, 512) do
        for j in range(0, 512) do
            if mapaEscribir[i,j]=='@' then
                mapa2[i,j]=0
            else if mapaEscribir[i,j]=='.' then
                mapa2[i,j]=255
            else
                mapa2[i,j] =ord(mapaEscribir[i,j])
            end
        end
    end
    plt.imsave("salidas/imagenD.jpg", mapa2, cmap='Greys')
    return (it, coste), "salidas/imagenD.jpg"
else
    return "No se ha encontrado resultado"
end
```

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S. DE INGENIERÍA INFORMÁTICA

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga