UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADO EN INGENIERÍA INFORMÁTICA

SIMULADOR PARA EL APRENDIZAJE DEL ALGORITMO DE PATHFINDING HPA*

A SIMULATOR FOR THE LEARNING OF THE HPA* PATHFINDING ALGORITHM

Realizado por
David Díaz Roldán

Tutorizado por
Lorenzo Mandow Andaluz

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, JUNIO DE 2022

Fecha defensa: Julio de 2022

UNIVERSIDAD DE MÁLAGA

Resumen

El problema de búsqueda de caminos en tiempo real es de suma importancia en los videojuegos. El coste computacional requerido por el algoritmo A*, uno de los más conocidos, aumenta considerablemente cuanto mayor es el espacio de búsqueda, y es por ello que surgen alternativas como el algoritmo HPA* (Hierarchical Path-Finding A*), con el que se reduce el esfuerzo computacional obteniéndose una muy buena aproximación de la solución óptima.

Este proyecto consiste en el desarrollo de una aplicación didáctica creada en Java, para simular y mostrar el comportamiento de los algoritmos A* y HPA*.

Al mismo tiempo, se han obtenido resultados experimentales para testear el programa y comparar el rendimiento entre ambos algoritmos de pathfinding, llegando a la conclusión de que HPA* supera a A* en cuanto a velocidad y menor cantidad de número de nodos expandidos, con un error mínimo en el resultado. Asimismo, se realiza una comparativa del tiempo consumido en distintas fases que conforman al algoritmo HPA*.

Palabras clave: Pathfinding, A*, HPA*, simulador, grafo abstracto

Abstract

The real-time pathfinding problem is of paramount importance in video games. The computational cost required by A* algorithm, one of the best known, increases considerably the larger the search space, and that is why alternatives arise such as HPA* (Hierarchical Path-Finding A*) algorithm, with which the computational effort is reduced and the optimal solution is obtained in a very approximate way.

This project consists of the development of a didactic application created in Java, to simulate and show the behavior of A* and HPA* algorithms.

At the same time, experimental results have been obtained to test the program and compare the performance between both path-finding algorithms, reaching the conclusion that HPA* exceeds A* in terms of speed and fewer number of expanded nodes, with a minimum error in the result. Also, a comparison of the time consumed in different phases that make up HPA* algorithm is made.

Keywords: Pathfinding, A*, HPA*, simulator, abstract graph

Índice

Resumen 1
Abstract 2
Índice 3
1. Introducción 5
1.1 Motivación 5
1.2 Objetivos 5
1.3 Estructura de la memoria 6
1.4 Metodología 6
1.5 Tecnologías empleadas 8
2. Antecedentes 9
2.1 Representación gráfica del mapa 9
2.2 Algoritmos de pathfinding 9
2.2.1 Heurísticos 10
2.2.1.1 Distancia Manhattan 10
2.2.1.2 Distancia octil 11
2.2.2 Algoritmo de Dijkstra 11
2.2.3 Algoritmo A* 12
2.2.4 Algoritmo HPA* 15
2.2.4.1 Preprocesamiento 15
2.2.4.2 Introducción de los puntos inicial y final 20
2.2.4.3 Búsqueda en el grafo abstracto 20
2.2.4.4 Refinamiento 21
2.2.4.5 Suavizado 21
2.3 Simulador AIDA-UMA 23
2.4 Librerías y clases de Java 24
2.4.1 JFileChooser 24
2.4.2 Graphics y Graphics2D 24
2.4.2.1 Clases complementarias 25
2.4.2.2 Funciones 26
2.4.3 Layouts 26
2.4.3.1 Layout null 26
2.4.3.2 BorderLayout 27
2.4.3.3 GridLayout 27
2.4.3.4 GridBagLayout 27
2.4.3.5 GroupLayout 28
2.4.4 Timer 28
2.5 Matriz dinámica de botones 29
2.6 Repositorio de mapas: Pathfinding Benchmarks 30
2.7 Amateras Modeler 30
3. Diseño del programa 31
3.1 Funcionamiento 31
3.2 Diseño de la aplicación 33
3.2.1 Simulador 35

3.2.2 Interfaz 35
3.2.3 Test 35
3.2.4 Mapa 36
3.2.5 Punto 36
3.2.6 Direccion 36
3.2.7 Edge 36
3.2.8 Arco 36
3.2.9 Cluster 37
3.2.10 Astar 37
3.2.11 HPAstar 38
3.2.12 Dijkstra 39
3.2.13 PanelArcos 39
3.3 Detección de errores 39
4. Resultados experimentales 41
4.1 Datos empleados 41
4.2 Comparativa entre A* y HPA* 41
4.3 Comparativa entre fases de HPA* 44
5. Conclusiones y trabajo futuro 47
6. Bibliografía 49
Apéndice A. Manual de Usuario 51
7.1 Introducción 51
7.2 Ejecución del programa 51
7.3 Uso del simulador 51
7.3.1 Pantalla principal del simulador 51
7.3.2 Gestión del mapa 53
7.3.2.1 Consulta 55
7.3.2.2 Inicial 55
7.3.2.3 Final 55
7.3.2.4 Obstáculo 55
7.3.2.5 Otras opciones 56
7.3.3 Gestión de ficheros 56
7.3.3.1 Formato del fichero 56
7.3.3.2 Guardar fichero 57
7.3.3.3 Abrir fichero 58
7.3.3.4 Nuevo 58
7.3.4 Selección de algoritmo 58
7.3.4.1 Selección del algoritmo A* 59
7.3.4.1.1 Modalidad 59
7.3.4.1.2 Control de la simulación 61
7.3.4.2 Selección del algoritmo HPA* 61
7.3.4.2.1 Selección de parámetros 62
7.3.4.2.2 Control de la simulación 62
7.3.4.2.3 Visualizar nodos internos 63
7.3.4.2.4 Simulación por pasos 63
7.4 Uso de la clase para pruebas experimentales 67
7.4.1. Realización de pruebas aleatorias 67
7.4.2. Realización de pruebas específicas 68

1

Introducción

1.1 Motivación

La industria de los videojuegos se presenta actualmente como una de las más importantes en el ámbito de la informática, contando con un mayor impacto o equiparándose a nivel económico y social a otras grandes como el cine, la música o la literatura. Con un público cada vez más exigente, se requiere encontrar formas de mejorar el rendimiento en sus aplicaciones.

En este ámbito, surge la necesidad de encontrar algoritmos a la par rápidos y precisos. Una alternativa general al algoritmo A* es el uso de grafos jerárquicos. HPA* [1] fue el primer algoritmo publicado con esta técnica, y es, por tanto, una referencia básica en su campo. Con el fin de ilustrar cada uno de los pasos de este algoritmo, surge la motivación de desarrollar una herramienta pedagógica.

1.2 Objetivos

En este trabajo, el objetivo principal es la implementación de un simulador gráfico donde se muestre una matriz cuadrada con obstáculos, que corresponde con un mapa sobre el que se va a aplicar un algoritmo de pathfinding, entre cuyas opciones se encuentran el clásico algoritmo A* y el algoritmo jerárquico HPA* (Herarchical Pathfinding A*).

La aplicación permitirá al usuario definir los distintos parámetros del mapa, es decir, las dimensiones y la ubicación de los puntos inicial y final, y de los distintos obstáculos, si existen; guardar en un fichero los datos del mapa definido, para abrirlo y así mostrarlo nuevamente cuando se requiera, y escoger el algoritmo a aplicar, estableciéndose por defecto A*, y pudiendo definir distintos parámetros conforme al

algoritmo a aplicar, como las dimensiones de los clusters dentro de HPA*, entre otros. Una vez definido el escenario y los distintos parámetros, se podrá comenzar la simulación, ver paso a paso el funcionamiento del algoritmo seleccionado, y detener la ejecución en cualquier momento.

Por otro lado, se realizan pruebas para verificar el correcto funcionamiento del programa y la vez extraer datos con los que comparar el rendimiento de ambos algoritmos.

1.3 Estructura de la memoria

Este proyecto cuenta con varios capítulos a diferenciar:

- **Capítulo 1: *Introducción*.** Se pone en contexto el trabajo.
- **Capítulo 2: *Antecedentes*.** Se explican los conceptos necesarios para comprender y desarrollar el trabajo.
- **Capítulo 3: *Diseño del programa*.** Se indica de qué manera interactúa el usuario con la aplicación, y cómo se ha diseñado, de cara al desarrollo del software.
- **Capítulo 4: *Resultados experimentales*.** Se muestran gráficamente los resultados obtenidos con los algoritmos A* y HPA*, tras la realización de pruebas con un mapa del videojuego *Baldurs Gate II*, para comparar el rendimiento entre ambos algoritmos de pathfinding y cada fase del algoritmo HPA*.
- **Capítulo 5: *Conclusiones y trabajo futuro*.** Se resumen algunas conclusiones y posibles mejoras de este proyecto.
- **Capítulo 6: *Referencias*.** Corresponde con la bibliografía del proyecto, utilizando el formato para referenciar establecido por las normas APA [34].
- **Anexo: *Apéndice A: Manual de usuario***. Al final de este proyecto, se adjunta el manual de uso de la aplicación.

1.4 Metodología

Para el desarrollo del software de este proyecto, se ha empleado la metodología SCRUM [2], que consiste en dividir cada tarea del proyecto en hitos u objetivos a cumplir en una fecha estipulada, junto a un seguimiento periódico para verificar que se completan de manera satisfactoria.

Al tratarse de un desarrollo software realizado por una sola persona, el profesor asume el rol de Scrum Master y Product Owner, aportando ayuda, definiendo los distintos hitos, y asegurando la calidad del proyecto; mientras que el alumno asume el rol de Desarrollador, trabajando en estos hitos para completarlos en las fechas estipuladas.

Cada cierto tiempo, en concreto, cada par de semanas, se han tenido reuniones donde se muestran y evalúan los avances, se consultan dudas y una vez cumplidos los hitos, se van estableciendo nuevos, hasta completar el programa.

Los hitos concretos que se han realizado durante el desarrollo del software han sido los siguientes:

1. **Crear diagramas de clases y de casos de uso**, para, de manera orientativa, tener claras las funcionalidades del simulador y las distintas clases. Se han ido modificando los diagramas a medida que avanzaba el proyecto.
2. **Definición de la interfaz gráfica**, es decir, de la ubicación y la forma de sus elementos.
3. **Creación de la matriz cuadrada**, que corresponde con el mapa en el que se ubican los puntos de interés (puntos inicial y final, y obstáculos).
4. **Gestión de ficheros** en los que poder almacenar la definición de los datos del mapa.
5. **Implementación de A*.**
6. **HPA*: División en clusters.**
7. **HPA*: División de puntos de entrada de cada cluster.** Incluye la creación de los arcos externos.
8. **HPA*: Construcción del grafo abstracto.** Se divide en:
    a. **Construcción del grafo abstracto.** Para este hito, se aplica el algoritmo de Dijkstra, que se explicará más adelante, para la creación de los arcos internos.
    b. **Definir una tabla para mostrar los nodos internos de cualquier cluster que se seleccione.**
    c. **Mostrar gráficamente los arcos externos,** de manera similar a la Figura 4(a) del documento *Near Optimal Hierarchical Path-Finding* [1].

9. **HPA*: Refinamiento.** Se aplica A* al grafo abstracto para encontrar la solución.
10. **Realización de las pruebas.** Se divide en:
    a. **Depuración de errores.**
    b. **Creación de los gráficos comparativos.**

1.5 Tecnologías empleadas

Para la programación del simulador de este proyecto, se ha utilizado el entorno Eclipse IDE con Java como lenguaje de programación. Dentro de este, se han utilizado principalmente las librerías java.awt y javax.swing para el desarrollo de la interfaz gráfica, y java.io para la gestión de ficheros.

Para la representación gráfica de los resultados experimentales, se ha aplicado MATLAB a un fichero con los datos obtenidos de las pruebas.

Para el diagrama de clases, se ha utilizado el plugin Amateras Modeler [18], diseñado para Eclipse, y para el diagrama de casos de usos, MagicDraw.

2

Antecedentes

2.1 Representación gráfica del mapa

Una manera muy recurrente de representar un mapa es mediante mallas, que son estructuras formadas por casillas que representan coordenadas en las cuales pueden existir objetos u obstáculos.

Existen mallas de diversos tipos, pero en este proyecto utilizaremos las mallas cuadradas. Este tipo de mallas es muy utilizado y existen incluso algoritmos de pathfinding específicamente diseñados para ellas. Tal y como se puede apreciar en la Figura 1, donde se muestra una imagen publicada por el usuario Trollkin en 2018 [21], los movimientos posibles son en horizontal y vertical, con un coste de 1, y en diagonal, con un coste de $\sqrt{2}$≈1.41. Para operar con números enteros en vez de decimales, se multiplican estos costes por 100.

Figura 1 Movimientos posibles en un mapa de mallas cuadradas.

2.2 Algoritmos de pathfinding

Se denomina pathfinding a la búsqueda del camino más corto entre dos puntos. Para encontrar este camino óptimo, se utilizan algoritmos, dentro de los cuales existen

los de tipo heurístico o informado, que emplean heurísticos como herramienta para estimar el coste de las posibles soluciones.

2.2.1 Heurísticos

En pathfinding, un heurístico es un estimador del coste desde un nodo al objetivo. Es necesario encontrar una heurística admisible, es decir, que no sobrestime el coste real de alcanzar el nodo objetivo, ya que en caso contrario, puede empeorar la solución. En este trabajo, se emplearán la distancia Manhattan y la distancia octil.

2.2.1.1 Distancia Manhattan

La distancia Manhattan estima el coste de un punto a otro considerando exclusivamente movimientos horizontales y verticales. Este heurístico es muy utilizado en mallas cuadradas de 4-vecinos, ya que nunca sobreestima el coste. Sin embargo, en mallas de 8-vecinos, como las empleadas en este proyecto, la distancia Manhattan puede sobreestimar el coste. En la Figura 2, podemos ver una imagen publicada por el usuario de GitHub Chris3606 en su artículo *Measuring Distance* [19]. El recuadro rojo representa la posición de partida; los azules, indican 1 unidad de distancia, los grises, 2 unidades, y los verdes, 3 unidades. Esta unidad de distancia, representa, a su vez, el coste desde el punto de partida.

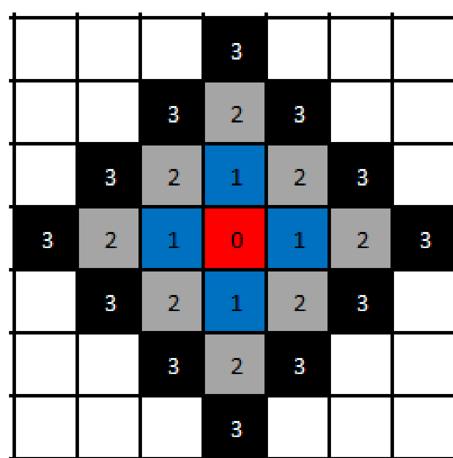


Figura 2 Cálculo del coste con la distancia Manhattan.

2.2.1.2 Distancia octil

La distancia octil estima el coste entre dos puntos considerando también el movimiento en diagonal, es decir, en 8 direcciones. En este caso, los costes en movimientos horizontales y verticales se evalúan de la misma forma que en la distancia Manhattan. Sin embargo, los movimientos diagonales tienen un valor de coste $\sqrt{2}$≈1.41.

La distancia octil entre dos puntos *x*, definido por sus coordenadas en fila y columna*(fx, cx)*, e *y*, definido por sus coordenadas en fila y columna *(fy, cy)*, se calcula de la siguiente forma:

$$dOctil(x,y) = c_{diagonal} * \min(\Delta f, \Delta c) + c_{Manhattan} * |\Delta f - \Delta c|$$

Donde:

$$\Delta f = |fx - fy| \text{ y } \Delta c = |cx - cy|$$

Siendo $c_{diagonal}$ el coste de moverse diagonalmente, y $c_{Manhattan}$ el coste de realizar un movimiento en dirección horizontal o vertical.

2.2.2 Algoritmo de Dijkstra

En textos de Inteligencia Artificial, se tratan los grafos en términos de nodos y arcos, mientras que en algunos libros matemáticos y de algoritmia, se utilizan los términos equivalentes de vértices y aristas.

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos o algoritmo de coste uniforme, determina el camino más corto desde un vértice o nodo origen hacia el resto en un grafo que tiene pesos en cada arista o arco.

Este algoritmo utiliza un árbol de búsqueda y una lista de nodos abiertos, y en él, se define la función $g(n)$, que representa el coste del camino guardado hasta el vértice *n* en el árbol. La lista de nodos abiertos se ordena según el valor de esta función, y el árbol guarda siempre el camino hasta cada nodo con menor $g(n)$.

A partir de un nodo inicial, se añaden sus adyacentes a la lista de nodos abiertos, de la cual se van tomando los siguientes. Se trata de un proceso recursivo cuya ejecución finaliza, normalmente, una vez alcanzado el camino con $g(n)$ mínimo hacia un nodo final o destino, aunque también puede utilizarse para hallar el camino de menor coste entre la totalidad de los vértices.

A continuación, en la Figura 3, se muestra un pseudocódigo del algoritmo, con comentarios en color verde, para su correspondiente explicación.

```
 # Sea V el conjunto de vértices del grafo G, Q una cola de prioridad
para vértices, según el menor valor de la distancia e inicial el
vértice que se establece como punto de partida:
 DIJKSTRA (Grafo G, Vértice inicial):
    para cada v ∈ V:
       # Se inicializan todos los nodos.
       inicializar(v)
    fin para cada
    # Se asignan los valores al nodo inicial, y se introduce este en
la cola.
    inicial.distancia = 0;
    Q.introducir(inicial)
    # El algoritmo comprueba hasta que no quedan nodos.
    mientras ¬Q.vacia:
       # Se coge aquel con menor distancia y se apunta como visitado.
       v = Q.menor_distancia
       v.visto = true
       # Se analizan todos sus adyacentes
       para cada ady ∈ v.adyacentes:
          # Si no se ha visitado y su distancia es mayor a la del
camino de llegar desde el vértice anterior.
          si ¬ady.visto & ady.distancia > v.distancia + peso(ady, v):
             # Se asigna esta nueva distancia, se asigna el vértice
anterior como padre y se añade el vértice actual a la cola.
             ady.distancia = v.distancia + peso(ady, v)
             ady.padre = v
             Q.introducir(ady)
          fin si
       fin para cada
    fin mientras
 fin DIJKSTRA

 inicializar (Vértice v):
    # Se asigna valor infinito a la distancia y se apunta como no
visitado al vértice.
    v.distancia = ∞
    v.visto = false
 fin inicializar
```

Figura 3 Algoritmo de Dijkstra con cola de prioridad en pseudocódigo. Basado en el del artículo sobre Dijkstra existente en Wikipedia [26].

2.2.3 Algoritmo A*

El algoritmo A* es un algoritmo de pathfinding de tipo heurístico o informado, que encuentra el camino de menor coste entre dos puntos, bajo condiciones razonables.

En él, se emplea una función de evaluación $f(n) = g(n) + h(n)$, donde $h(n)$ representa el valor heurístico o estimación de coste de un camino desde el nodo actual al nodo final, y $g(n)$ representa el coste del camino recorrido en el árbol desde la raíz hasta n, es decir, el acumulado de llegar desde el nodo inicial al nodo actual. Si el

heurístico empleado fuera $h(n) = 0$, este algoritmo se comportaría de forma idéntica al algoritmo de Dijkstra.

El algoritmo A* trabaja con un par de listas de nodos: la de nodos abiertos, que se va rellenando con los adyacentes de los nodos visitados, y que, además, representa a los nodos pendientes de explorar y se ordena conforme al menor coste de la función $f(n)$; y la lista de nodos cerrados, que representa los nodos visitados, los cuales se van seleccionando de la lista de nodos abiertos siguiendo su orden. Una vez se alcance el nodo final u objetivo, o no se encuentre ninguna solución, finaliza la ejecución del algoritmo.

A continuación, se muestra el pseudocódigo del algoritmo en la Figura 4(a), cuya explicación puede verse en los comentarios en color verde.

```
 # Sea A la lista de nodos abiertos, ordenada según el menor valor de
f(n) y C la lista de nodos cerrados, ambas a priori vacías:
 A* (Grafo G, Nodo inicial, Nodo final):
    # Se añade el nodo inicial a la lista de nodos abiertos (la de
nodos cerrados permanece vacía aún).
    A.introducir(inicial)
    # Se asigna h(n) del nodo a su correspondiente f(n), ya que g(n)
es 0 en este caso.
    inicial.f = inicial.h
    # Se coge el nodo n de la lista de abiertos (en este caso
corresponde con el nodo inicial).
    n = A[0]
    # Mientas que queden elementos en A y que el nodo con menor f(n)
no sea el nodo final.
    mientras ¬A.vacia | n ≠ final:
       # Se quita el nodo n de la lista de nodos abiertos y se
introduce en la lista de nodos cerrados.
       A.quitar(n)
       C.introducir(n)
       # Si no coincide el nodo n con el nodo final.
       si n ≠ final:
          # Se llama a la función tratar_sucesor (aparece en la
segunda parte del pseudocódigo de A*).
          para cada suc ∈ n.sucesores:
             tratar_sucesor(suc, n)
          fin para cada
          # Se coge el nodo n de la lista de nodos abiertos (en este
caso corresponde con el nodo con menor f(n)).
          n = A[0]
       fin si
    fin mientras
 fin A*
```

Figura 4(a) Algoritmo A* en pseudocódigo. Basado en el del artículo sobre A* en Wikipedia [25].

En la Figura 4(b), puede verse la segunda parte del pseudocódigo de A*, donde se muestra el pseudocódigo de la función *tratar_sucesor*, que se llama en la primera parte del pseudocódigo de A*. Nuevamente, su explicación viene dada en los comentarios en color verde.

```
# Sea n el mejor nodo cogido de la lista de nodos abiertos A, y suc
uno de sus nodos sucesores:
 tratar_sucesor (Nodo suc, Nodo n):
    # Asignamos los valores de f(n) y de g(n) al nodo suc.
    asignar_f(suc, n)
    # Si el nodo suc se encuentra en la lista de nodos cerrados.
    si suc ∈ C:
       # Se coge el nodo ant de la lista de nodos cerrados, según la
posición que debe tener el nodo suc en la misma.
       ant = C[posición(suc)]
       # Si el nodo sucesor presenta un menor g(n) que el nodo ant.
       si suc.g < ant.g:
          # Actualizamos los valores de f(n) y g(n) del nodo ant, le
asignamos como padre el nodo n y propagamos el valor de g(n) para
todos sus sucesores de ant.
          ant.f = suc.f
          ant.g = suc.g
          ant.padre = n
          para cada s ∈ ant.sucesores:
             s.propagar_g
          fin para cada
       fin si
    # Si el nodo suc se encuentra en la lista de nodos abiertos.
    Si suc ∈ A:
       # Se coge el nodo ant de la lista de nodos abiertos, según la
posición que debe tener suc en la misma.
       ant = A[posición(suc)]
       # Si g(n) de suc es menor que g(n) de ant.
       si suc.g < ant.g
          # Le asignamos a ant los valores de f(n) y g(n) del nodo
suc, le asignamos como padre el nodo n, y ya con los datos
actualizados ordenamos la lista de nodos abiertos según f(n) mínimo.
          ant.f = suc.f
          ant.g = suc.g
          ant.padre = n
          A.ordenar
       fin si
    # Si el nodo no se encuentra en ninguna de las dos listas.
    si no:
       # Se introduce el nodo suc a la lista de nodos abiertos, y se
ordena según f(n) mínimo.
       A.introducir(suc)
       A.ordenar
    fin si
 fin tratar_sucesor
```

Figura 4(b) Segunda parte del pseudocódigo de A*, que corresponde con la función *tratar_sucesor*.

Basado en el pseudocódigo de *TRATAR_SUCESOR* dentro del artículo sobre A* en Wikipedia [25].

Por último, en la Figura 4(c) podemos ver la tercera parte del pseudocódigo de A*, que corresponde con la función *asignar_f*, que se llama desde *tratar_sucesor*. Su explicación viene dada, una vez más, en los comentarios color verde.

```
 # Sea n el mejor nodo cogido de la lista de nodos abiertos A, y suc
uno de sus nodos sucesores:
 asignar_f (Nodo suc, Nodo n):
    # Se asigna el valor de f(n) del nodo suc como la suma entre su
valor de h(n) más g(n) del nodo n anterior más el coste de llegar de n
a suc
    suc.f = n.g + coste(n, suc) + suc.h
 fin asignar_f
```

Figura 4(c) Tercera parte del pseudocódigo de A*, que corresponde con la función *asignar_f*.

2.2.4 Algoritmo HPA*

El algoritmo HPA* se define por primera vez en el artículo *Near Optimal Hierarchical Path-Finding* [1]. Es un algoritmo de búsqueda de caminos jerárquico basado en el algoritmo A*. Parte del problema de encontrar la ruta más corta entre dos ciudades, considerando que hay múltiples carreteras intermedias posibles, y lo abstrae en un algoritmo con distintas fases que se explicarán a continuación.

2.2.4.1 Preprocesamiento

En esta primera fase, se divide el mapa en grupos o secciones disjuntos llamados clusters, que se usarán para la creación del grafo abstracto.

Entre las fronteras de cada par de clusters adyacentes, identificamos un conjunto de entradas, que puede estar vacío. Llamamos entrada a la unión entre dos puntos adyacentes, ubicados detrás de las líneas limítrofes y libres de obstáculos. Considerando un par de clusters, c_1 y c_2, y las líneas que delimitan los bordes, l_1 y l_2, tenemos una unión $t \in l_1 \cup l_2$ y definimos symm(t) como el simétrico de t. La entrada e debe cumplir las siguientes condiciones:

- **Condición de limitación fronteriza**: $e \subset l_1 \cup l_2$. Una entrada debe situarse y nunca exceder el borde entre dos clusters adyacentes.
- **Condición simétrica**: $\forall\, t \in l_1 \cup l_2 : t \in e \Leftrightarrow \text{symm}(t) \in e$.

- **Condición de frontera libre de obstáculos**: una entrada no puede contener obstáculos.
- **Condición de maxilidad**: una entrada se extiende en ambas direcciones siempre que se cumplan las condiciones anteriores.

Más adelante, en la Figura 7(a), podremos ver los nodos creados entre cuatro distintos clusters.

En la Figura 5, puede verse el mapa de un problema de ejemplo definido, creado en nuestro simulador, tomando como referencia la Figura 1 (a) del documento *Near Optimal Hierarchical Path-Finding* [1]. Los recuadros negros representan los obstáculos, y los grises, casillas vacías. Los puntos inicial y final están representados en color verde y rojo, respectivamente.

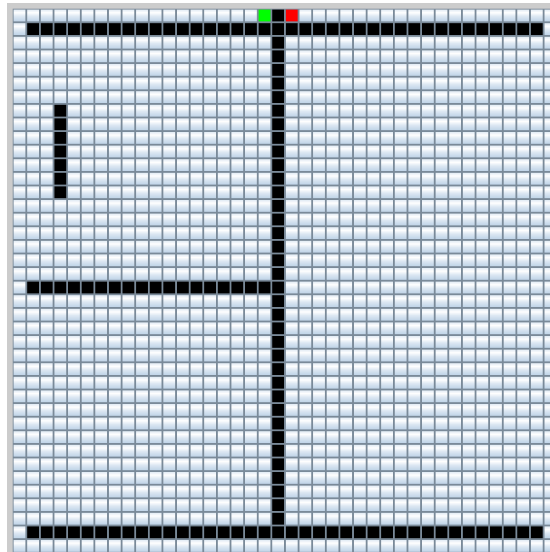


Figura 5 Mapa de 40x40 con el problema definido.

A continuación, en la Figura 6, puede verse el mismo mapa con clusters de 10x10 creados en él, basándonos en la Figura 1(b) del documento *Near Optimal Hierarchical Path-Finding* [1].

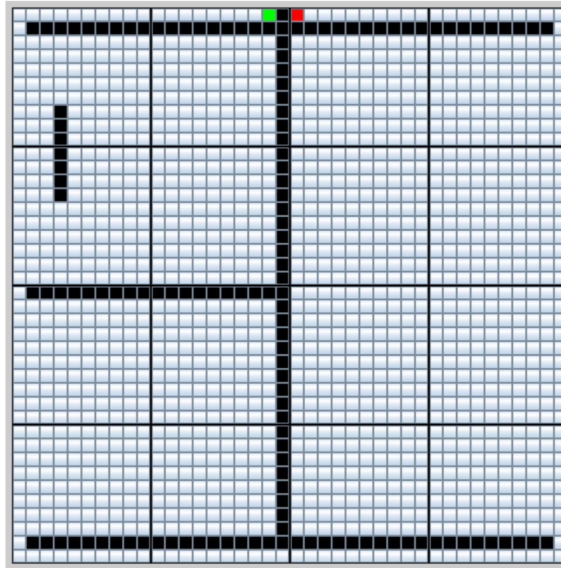


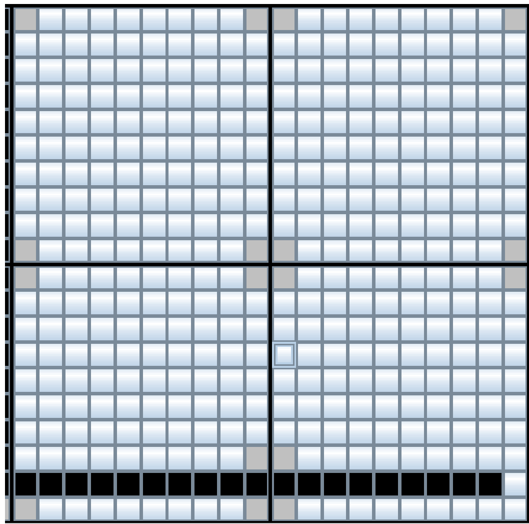
Figura 6 Clusters de 10x10 creados en el mapa.

Para la creación de nodos, se utiliza una constante como umbral: Si el ancho de la entrada es menor que este umbral, definimos dos nodos adyacentes a cada lado fronterizo de la mitad de la entrada; en caso contrario, se definen en los extremos de la entrada.

Estos nodos se utilizan para construir el grafo abstracto. Entre cada par de estos nodos adyacentes, se define un arco para unirlos: los arcos externos, que siempre tienen longitud 1. Entre cada par de nodos dentro de un mismo cluster, se define un arco interno que los una. El camino entre estos nodos y su longitud queda definido mediante una búsqueda con un algoritmo de pathfinding.

En la Figura 7(a), se pueden ver los nodos creados en cada entrada de un segmento del mapa antes mostrado. Para ello, se ha establecido un umbral con valor 6.

En la Figura 7(b), podemos ver una representación del grafo abstracto de ese mismo segmento, y en él pueden apreciarse los arcos externos e internos.



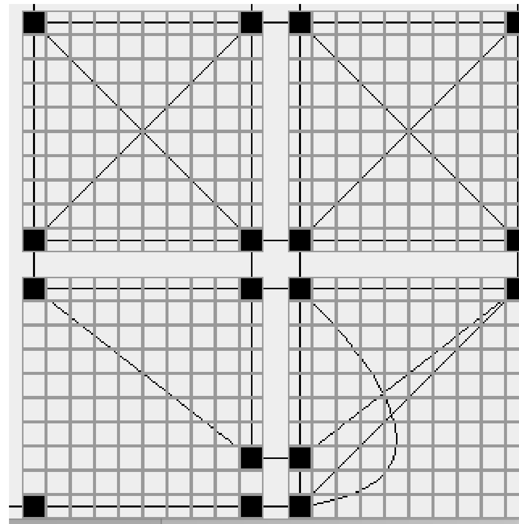
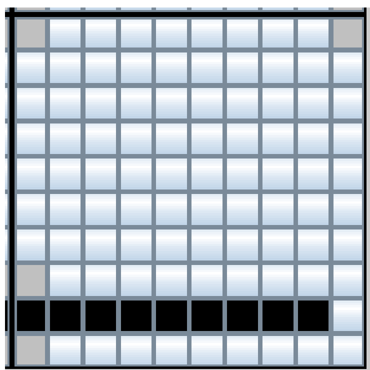


Figura 7(a) Nodos definidos. **Figura 7(b)** Arcos externos e internos.

A continuación, en la Figura 8, podemos ver la información del grafo abstracto interno de un cluster de este mismo problema.



Lista de nodos

n15,0: (30,30)
n15,1: (30,39)
n15,2: (37,30)
n15,3: (39,30)

Arcos internos

	n15,0	n15,1	n15,2	n15,3
n15,0	X	900	700	2169
n15,1	X	X	1187	1741
n15,2	X	X	X	1882
n15,3	X	X	X	X

Figura 8 Información de un cluster.

A continuación, en la Figura 9, podemos ver el pseudocódigo de la parte de preprocesamiento, cuya explicación viene dada en los comentarios en color verde.

```
 # Sea E el conjunto de entradas, a priori vacío, C el conjunto de
clusters y M el mapa del problema
 abstracciónMapa (Mapa M):
    # Primeramente, construimos los clusters en el mapa y los añadimos
al conjunto C.
    C.construirClusters(M)
    # Después, vamos cogiendo pares de clusters de C y comprobamos si
son adyacentes.
    para cada c1, c2 ∈ C:
          # En caso de ser adyacentes, añadimos al conjunto E las
entradas que creemos entre el par de clusters.
       si adyacentes(c1, c2):
          E.introducir(construir_entradas(c1, c2))
       fin si
    fin para cada
 fin abstracciónMapa

 # Sea N el conjunto de nodos de los clusters:
 construirGrafo ():
    # Para cada entrada e del conjunto E
    para cada e ∈ E:
       # Cogemos los dos cluster de la entrada e y creamos nodos para
cada lado de e. Los añadimos al conjunto N y creamos un arco externo
entre ellos, con valor de distancia 1.
       c1 = getCluster1(e)
       c2 = getCluster2(e)
       n1 = nuevoNodo(e, c1)
       n2 = nuevoNodo(e, c2)
       N[c1].introducir(n1)
       N[c2].introducir(n2)
       añadirArco(n1, n2, 1, EXTERNO)
    fin para cada
    # Para cada cluster del conjunto C, cogemos pares de nodos
distintos entre sí.
    para cada c ∈ C:
       para cada n1, n2 ∈ N[c], n1 ≠ n2):
          # Mediante un algoritmo de pathfinding, buscamos un camino
entre ellos, y devolvemos el valor de su distancia.
          d = calcularDistancia(n1, n2, c)
          # Si esta distancia es menor que infinito, es decir, si
existe un camino entre ambos nodos, creamos un arco interno el valor
de esta distancia entre ellos.
          si d < ∞:
             añadirArco(n1, n2, d, INTERNO)
          fin si
       fin para cada
    fin para cada
 fin construirGrafo

 # Ejecutamos las dos funciones definidas anteriormente:
 preprocesamiento (Mapa M):
     abstracciónMapa(M)
     construirGrafo()
 fin preprocesamiento
```

Figura 9 Pseudocódigo de la fase de preprocesamiento. Basado en la Figura 9 del documento *Near Optimal Hierarchical Path-Finding* [1].

2.2.4.2 Introducción de los puntos inicial y final

En este paso, se introducen los puntos inicial y final en el conjunto de nodos dentro de los clusters que, respectivamente, los contienen.

2.2.4.3 Búsqueda en el grafo abstracto

Para poder encontrar un camino en el grafo abstracto, es necesario que los puntos inicial y final sean parte del mismo.

Al igual que en la etapa de preprocesamiento, se utiliza un algoritmo de pathfinding para crear arcos internos que unan estos puntos con el resto de nodos dentro de sus respectivos clusters.

En la Figura 10, podemos apreciar todos los nodos creados en el mapa del problema que hemos definido en la Figura 6.

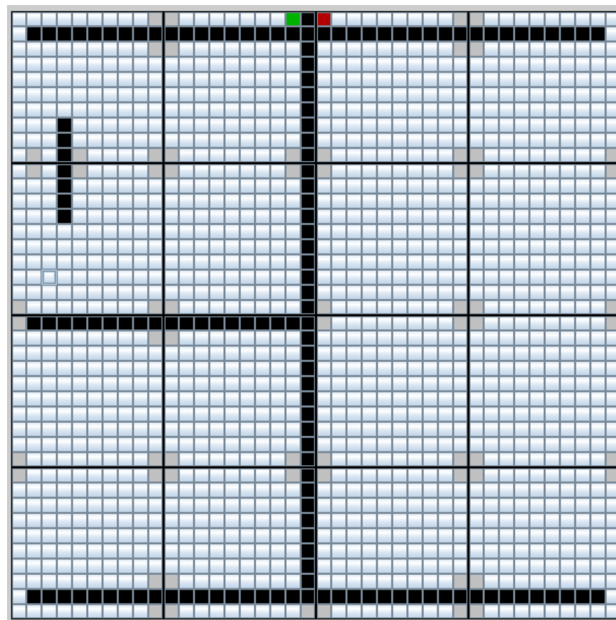


Figura 10 Nodos en el mapa.

En la Figura 11, podemos ver una representación completa del grafo abstracto del problema, ya incluyendo los puntos inicial y final. También está basado en un ejemplo del documento *Near Optimal Hierarchical Path-Finding* [1]; concretamente, en la Figura 4(a) del mismo.

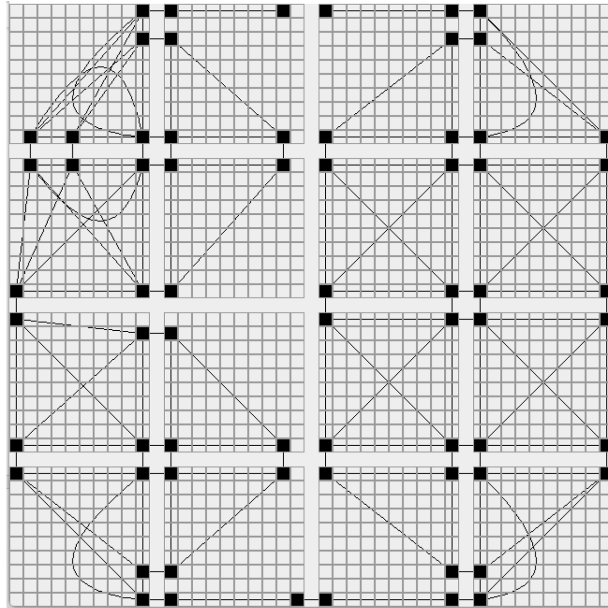


Figura 11 Grafo abstracto.

2.2.4.4 Refinamiento

En esta etapa, a partir del grafo abstracto, se aplica el algoritmo A* para encontrar el camino que una al punto inicial con el final.

La Figura 12 representa la solución encontrada para el problema de ejemplo definido en la Figura 6. El color salmón corresponde con el camino recorrido, y el azul oscuro, con los nodos explorados.

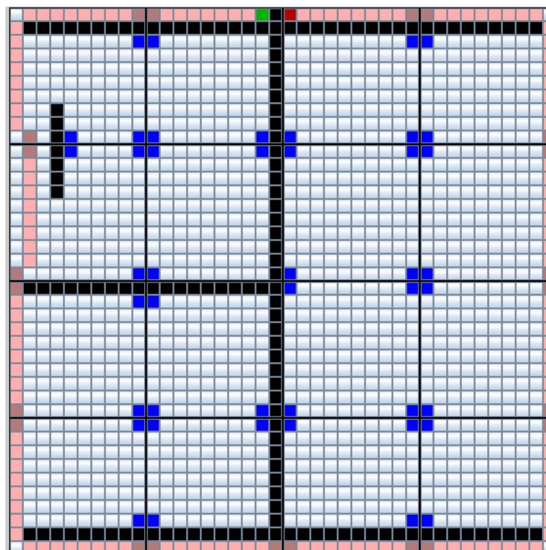


Figura 12 Solución del problema.

2.2.4.5 Suavizado

Este paso, que es opcional, suele aplicarse debido a que la solución obtenida es óptima para el grafo abstracto, pero no dentro del grafo del problema inicial, además,

de porque el movimiento en 8-direcciones impuesto por la malla puede no resultar natural al ejecutarlo en un videojuego.

Consiste en reemplazar las partes subóptimas locales por líneas rectas, comenzando por un extremo de la solución. Para cada nodo de la solución, se comprueba si puede alcanzarse un nodo posterior de la ruta en línea recta. En caso afirmativo, se utiliza esta recta para reemplazar la secuencia subóptima entre los nodos, liberando al camino suavizado de la rigidez de la malla.

En la Figura 13, podemos ver estas cuatro últimas partes (también conocidas en conjunto como búsqueda online) del algoritmo de HPA* en pseudocódigo. La explicación del mismo viene dada en los comentarios en color verde.

```
# Dados un nodo s (inicial o final) y un cluster c:
 conectarAFrontera (Nodo s, Cluster c):
    # Para cada nodo n del cluster c.
    para cada n ∈ N[c]:
       # Calculamos la distancia d entre s y n dentro del cluster c, y si es menor que infinito, es decir, si encuentra solución, se añade un arco interno entre ellos cuya distancia es el valor de d.
       d = calcularDistancia(s, n, c)
       si d < ∞:
          añadirArco(s, n, d, INTERNO)
       fin si
    fin para cada
 fin conectarAFrontera
 # Método para insertar el nodo s en el grafo abstracto:
 insertarNodo (Nodo s):
    # Determinamos el cluster al que pertenece el nodo s, y llamamos a la función conectarAFrontera, definida anteriormente.
    c = determinarCluster(s)
    conectarAFrontera(s, c)
 fin insertarNodo
 # Dados un par de nodos inicial y final, que corresponden con los puntos de inicio y objetivo, respectivamente; y una variable booleana, que determina si se desea aplicar el suavizado:
 búsquedaJerárquica (Nodo inicial, Nodo final, Boolean suavizado):
    # Se insertan los nodos inicial y final en el grafo abstracto
    insertarNodo(inicial)
    insertarNodo(final)
    # Se realiza una búsqueda en el grafo abstracto, seguido del refinamiento, para finalmente, de forma opcional, aplicar suavizado
    caminoAbstracto = buscarCamino(s, g)
    caminoRefinado = refinarCamino(caminoAbstracto)
    si suavizado:
       caminoSuavizado = caminoSuavizado(caminoRefinado)
    fin si
fin búsqueda Jerárquica
```

Figura 13 Pseudocódigo de la fase de búsqueda online. Basado en la Figura 10 del documento *Near Optimal Hierarchical Path-Finding* [1].

2.3 Simulador AIDA-UMA

AIDA-UMA [3], creado para uso didáctico en la UMA, es un simulador de algoritmos de búsqueda de caminos. Está desarrollado en Common Lisp, usando el paquete de Lispworks CAPI para interfaces gráficas de usuario. Permite la selección de distintos algoritmos (Amplitud, A*, Búsqueda Frontera…) y heurísticos (h=0, distancia euclídea y distancia Manhattan), mostrando paso a paso su ejecución en una malla de 4 vecinos sobre el que han podido establecerse los puntos inicial y final, y los obstáculos. Además, permite el guardado y la apertura de ficheros en los que se almacena la información de los puntos establecidos sobre el mapa.

Tiene especial mención en este proyecto, dado que ha servido como inspiración para la parte de la interfaz gráfica del simulador. En la Figura 14, podemos ver cuál es su aspecto.

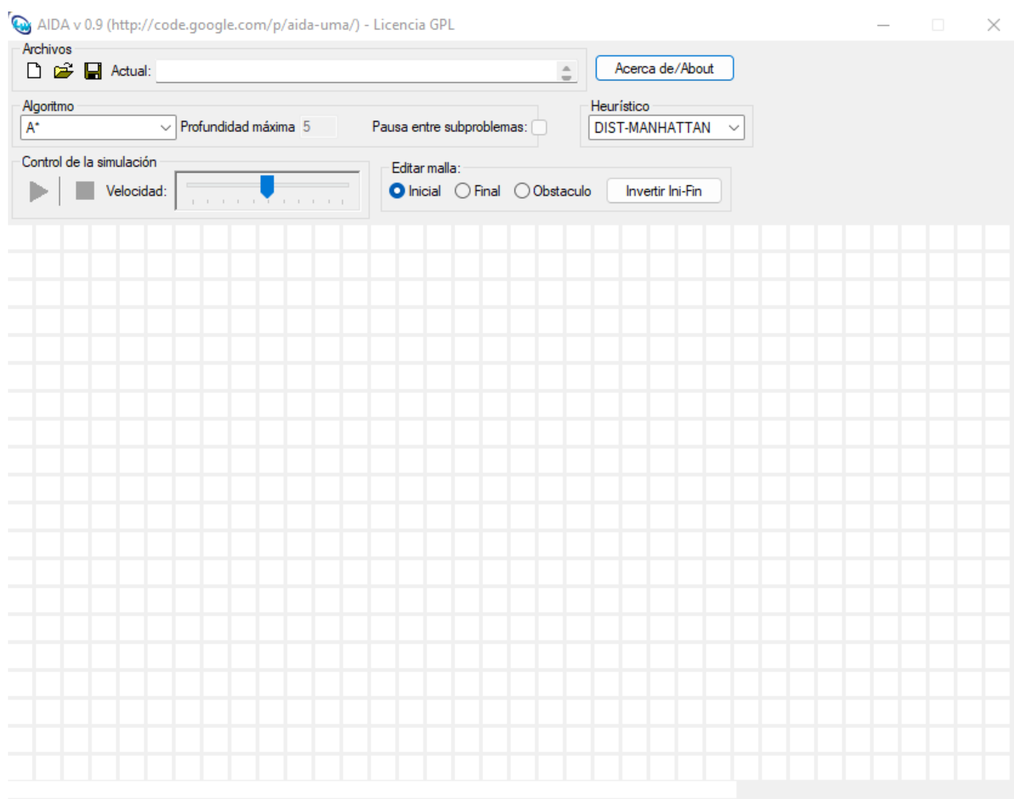


Figura 14 Interfaz de AIDA-UMA.

2.4 Librerías y clases de Java

El software de este proyecto se desarrolla principalmente en Java, teniendo como librerías principales java.awt y javax.swing. Se explicarán a continuación algunas de las clases utilizadas.

2.4.1 JFileChooser

JFileChooser es una clase de la librería java.swing que cuenta con un mecanismo simple de apertura de ficheros a través de una ventana de selección.

El directorio que se abre por defecto es el HOME del usuario (*C:\Documents and Settings\usuario* en Windows, */home/usuario* en Linux). Inicialmente, solo permite la selección de ficheros, aunque puede establecerse que abra también (o solamente) directorios, con la función *setFileSelectionMode()*.

Mediante el método *setFilter()*, pueden establecerse filtros para mostrar únicamente un tipo de fichero. Podemos ver como ejemplo la Figura 15, en la que se muestra una forma de establecer un filtro con el que mostrar únicamente archivos *.jpg* y *.gif*.

```
JFileChooser jf = new JFileChooser();
FileNameExtensionFilter filter = new FileNameExtensionFilter("JPG &
GIF", "jpg", "gif");
jf.setFilter(filter);
```

Figura 15 Sintaxis para establecer un filtro en JFileChooser. Ejemplo tomado del artículo sobre JFileChooser en Chuwiki [27].

2.4.2 Graphics y Graphics2D

Ambas clases forman parte de la librería java.awt. La clase Graphics es una clase básica que permite pintar sobre componentes. Graphics2D es una clase que extiende a Graphics y que permite una gestión más sofisticada sobre geometría, transformación de coordenada, colores y textos.

2.4.2.1 Clases complementarias

Algunas clases muy útiles de java.awt que se utilizan junto a Graphics y Graphics2D son las siguientes:

- **Color**: Representa el color con el que se muestra en la interfaz los objetos. Algunas funciones muy útiles son:
  - *darker()* para mostrar el color en un tono más oscuro.
  - *brighter()* para mostrar el color en un tono más claro.
  - *getColor()* para obtener el color de un objeto.
  - *setColor()* para modificar el color de un objeto.

A continuación, en la Figura 16, puede verse la lista completa de colores predefinidos en esta clase, sacada del curso de Java de Ángel Franco García [28].

Nombre	Red (rojo)	Green (verde)	Blue (azul)
white	255	255	255
lightGray	192	192	192
gray	128	128	128
drakGray	64	64	64
black	0	0	0
red	255	0	0
pink	255	175	175
orange	255	200	0
yellow	255	255	0
green	0	255	0
magenta	255	0	255
cyan	0	255	255
blue	0	0	255

Figura 16 Lista de colores predefinidos en la clase Color.

- **Point**: Representa las coordenadas (x,y) de un objeto en la interfaz. Una función muy útil para obtener estas coordenadas de un objeto es *getLocation()*.
- **Rectangle**: Como su nombre indica, representa un rectángulo. Tiene como parámetros las coordenadas (x, y) del objeto, y sus dimensiones. Es una subclase de Rectangle2D, y se utiliza porque algunos objetos devuelven como parámetro un objeto de la clase Rectangle si se llama a la función *getBounds()*.
- **Rectangle2D**: Representa en alto nivel lo mismo que el anterior. A partir de un objeto Rectangle, se puede obtener un Rectangle2D mediante la función *getBounds()*.

- **QuadCurve2D**: Representa una curva comprendida entre dos puntos, con un punto intermedio que establece el grado de curvatura. Esta curva se define a través del método *setCurve()*.

2.4.2.2 Funciones

Algunas funciones muy útiles que se utilizan tanto en Graphics, como en Graphics2D son las siguientes:

- **paintComponent()**: Define la forma en que se van a pintar los objetos en la interfaz.
- **setColor()**: Establece el color con el que va a pintar el objeto de la clase Graphics.
- **setPaint()**: Establece el color con el que va a pintar el objeto de la clase Graphics2D.
- **drawline()**: Se usa en Graphics y en Graphics2D, para pintar una línea recta entre dos puntos.
- **draw()**: Se usa en la clase Graphics2D para pintar el contorno de un objeto gráfico.
- **fill()**: Se usa en la clase Graphics2D para pintar el contenido, y no solo el contorno, de un objeto gráfico.

2.4.3 Layouts

En Java, un Layout es la clase que define la manera en la que se distribuyen los objetos dentro de la ventana. Tanto las librerías java.awt, como javax.swing contienen objetos Layout. A continuación, se explican algunos de los tipos que existen.

2.4.3.1 Layout null

En este caso, no se utiliza Layout, sino que se establece como null. Al no haber gestión automatizada de los componentes, es el programador quien debe definir la coordenada exacta y el tamaño de cada elemento. No es recomendable, ya que en caso de modificar la ventana, los componentes permanecerán en el mismo lugar y con las mismas dimensiones, es decir, no se adaptarán al nuevo tamaño de la ventana.

2.4.3.2 BorderLayout

El BorderLayout pertenece a la librería java.awt. Divide la ventana en 5 partes: arriba, abajo, centro, derecha e izquierda (NORTH, SOUTH, CENTER, EAST Y WEST, respectivamente). Es el que se establece por defecto para los JFrame y JDialog, y el más adecuado para ventanas en las que existe un componente central importante, con menús o barras de herramientas a su alrededor. Debe tenerse en cuenta lo siguiente:

- Los componentes de las **partes superior e inferior** ocuparán el alto que necesiten, pero se estirarán horizontalmente hasta ocupar toda la ventana.
- Los componentes de las **partes derecha e izquierda** ocuparán el ancho que necesiten, pero se estirarán verticalmente hasta ocupar toda la ventana.
- El componente **central** se estirará en ambos sentidos hasta ocupar toda la ventana.

2.4.3.3 GridLayout

El GridLayout pertenece a la librería java.awt. Coloca los componentes en forma de matriz o cuadrícula, estirándolos hasta igualar su tamaño al de los demás. Es adecuado en aplicaciones en las que todos los botones son idénticos.

2.4.3.4 GridBagLayout

El GridBagLayout pertenece a la librería java.awt. Se asemeja al GridLayout, pero permite definir de forma independiente los tamaños de las celdas.

Admite como parámetro un objeto de la clase GridBagConstraints, donde se establece la forma en que se almacenan los componentes:

- **GridBagConstraints.gridx** nos indica la posición en columna en la que se encuentra el componente. Si ocupa varias columnas, se debe indicar la que se ubica en la esquina superior izquierda.
- **GridBagConstraints.gridy** nos indica la posición en fila en la que se encuentra el componente. Si ocupa varias filas, debe indicarse la que se ubica en la esquina superior izquierda.
- **GridBagConstraints.gridwidth** nos indica la cantidad de celdas que debe ocupar horizontalmente el componente, es decir, su anchura.

- **GridBagConstraints.gridheight** nos indica la cantidad de celdas que debe ocupar verticalmente el componente, es decir, su altura.

2.4.3.5 GroupLayout

El GroupLayout pertenece a la librería javax.swing. Trabaja con grupos de layouts verticales y horizontales de forma separada, empleando dos formas de trabajo: secuencial y paralela, combinadas con una composición jerárquica:

- Con la forma de trabajo secuencial, los componentes simplemente se colocan uno detrás de otro, es decir, su posición es relativa con respecto a la del componente anterior.
- Con la forma de trabajo paralela, los componentes se alinean con respecto al eje vertical u horizontal, o el centro.

2.4.4 Timer

La clase Timer nos permite realizar una tarea cada cierto tiempo. Existe un par de clases Timer en Java; una pertenece a la librería javax.swing, y la otra, a java.util. En este trabajo, se tratará la primera.

La clase javax.swing.Timer es la más sencilla de usar de las dos. Basta con instanciarla indicando cada cuanto tiempo necesitamos que se ejecute, y definiendo un ActionListener, cuyo método actionPerformed() se ejecutará periódicamente. Para iniciarla, simplemente debe llamarse al método start(), y para detenerla, al método stop(). En la Figura 17, se muestra una declaración de ejemplo de un objeto de esta clase.

```
Timer timer = new Timer (tiempoEnMilisegundos, new ActionListener ()
{
    public void actionPerformed(ActionEvent e)
    {
        // Aquí el código que queramos ejecutar.
     }
});
...

timer.start();
```

Figura 17 Ejemplo de declaración de la clase Timer. Tomada del tutorial de ejemplos de la clase Timer en Java publicado por Chuidiang Roxas [12].

El UI (User Interface) de Java no colorea los objetos hasta finalizar la ejecución de los métodos que lo requieran, aunque se indique pintar varias veces dentro de un bucle. Es por ello que se recurre a un objeto de esta clase Timer, ya que ejecuta el método que se defina una vez cada cierto tiempo, sustituyendo así una ejecución completa por varios pasos que puede ir pintando el UI de Java una vez finalicen.

2.5 Matriz dinámica de botones

Para la creación del mapa de la interfaz gráfica, se ha utilizado como base una matriz dinámica de botones en Java, cuyo código abierto se puede descargar en el repositorio GitHub del usuario DevJTello [9].

Utiliza GridLayout para gestionar de manera dinámica los botones y mostrarlos con tamaño idéntico. Además, en el programa se implementan funciones para definir el número de filas y de columnas de la matriz, y métodos para crearla y destruirla. En la Figura 18, podemos ver el aspecto de la interfaz de esta matriz de botones [33].

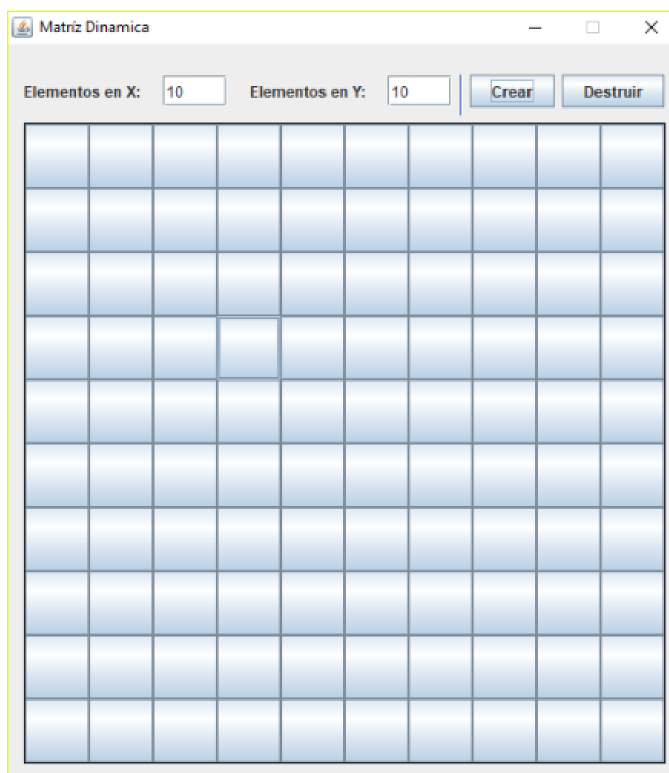


Figura 18 Matriz gráfica de botones.

2.6 Repositorio de mapas: Pathfinding Benchmarks

Pathfinding Benchmarks [15] es un repositorio de mapas destinados al estudio de problemas de pathfinding. Nosotros, en concreto, utilizamos un mapa de *Baldurs Gate II* [16] para problemas de mapas bidimensionales, que se muestra en la figura 19.

Figura 19 Mapa de 320x320 de *Baldurs Gate II*.

2.7 Amateras Modeler

Amateras Modeler es un plugin de Eclipse que permite la creación y edición de diagramas entidad-relación y de diagramas UML, cuyo código abierto se encuentra en el repositorio de GitHub del usuario takezoe [18]. Se han importado las clases de nuestro proyecto en él para crear de manera automatizada el diagrama UML de clases.

3

Diseño del programa

3.1 Funcionamiento

En este programa distinguimos dos partes: la del simulador, y la de ejecución de pruebas. Para comprender el comportamiento del software de cara al usuario, hemos creado los diagramas de casos de uso, utilizando MagicDraw. Se mostrarán a continuación y servirán como una introducción del funcionamiento de la aplicación, ya que se explicará más detalladamente en el manual.

En la Figura 20(a), podemos ver el diagrama de casos de uso del simulador. A grandes rasgos, el usuario puede gestionar ficheros, configurar el mapa, seleccionar el algoritmo y escoger sus parámetros, y ejecutar, pausar o detener la aplicación.

Figura 20(a) Diagrama de casos de uso del simulador.

En la Figura 20(b), se muestra el diagrama de casos de uso de la ejecución de pruebas experimentales. A grosso modo, el usuario puede ejecutar un número determinado de pruebas aleatorias, cuyos resultados se pasan a un fichero. Cada problema viene definido por un punto de origen y otro de destino dentro de un mapa. El usuario puede realizar una prueba específica definiendo el problema y especificando el algoritmo de pathfinding a aplicar, que en nuestro caso puede ser A* o HPA*. Una vez finalizada esta prueba concreta, se mostrará por pantalla el mapa con la solución.

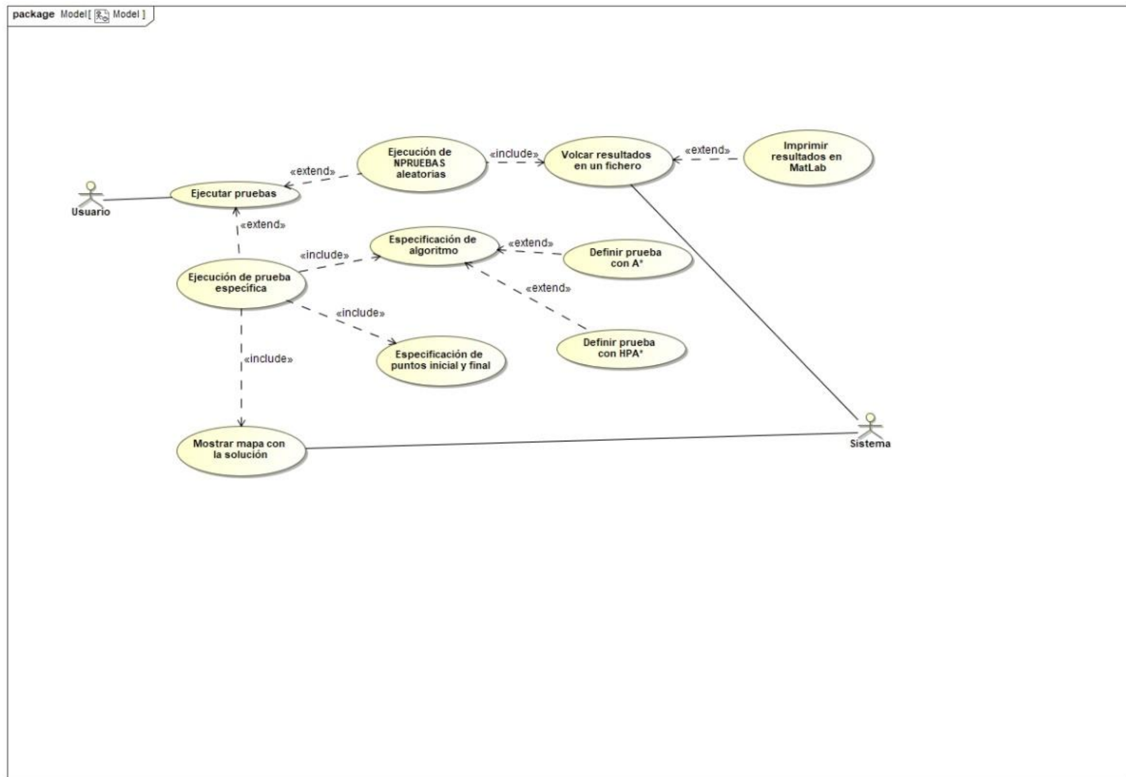


Figura 20(b) Diagrama de casos de uso de ejecución de pruebas.

3.2 Diseño de la aplicación

En la Figura 21(a), podemos ver el diagrama UML de clases de la aplicación, creado automáticamente a través del plugin Amateras Modeler de Eclipse [18]. Las distintas clases se comentarán más adelante.

Figura 21(a) Diagrama UML de clases.

3.2.1 Simulador

Esta clase contiene el main, es decir, desde ella se ejecuta la aplicación, llamándose, por defecto, a la clase Interfaz, que permite el uso del simulador, o, si así se indica, a la clase Test, que permite la realización de pruebas experimentales.

3.2.2 Interfaz

Es la clase sobre la que se ha desarrollado la aplicación, es decir, la que corresponde con la interfaz del simulador. Desde ella, se ejecutan los algoritmos de pathfinding llamando a métodos de las clases Astar y HPAstar. Para interactuar con la interfaz, se utilizan varios objetos de la clase JButton y JComboBox, entre otros. Cabe destacar, el objeto de la clase Mapa, donde se muestra el mapa con los puntos inicial, final y obstáculos, y en el que se visualizan las simulaciones; y también el objeto de la clase JFileChooser para la gestión de ficheros.

3.2.3 Test

Como bien hemos mencionado, esta clase permite la realización de pruebas experimentales de A* y HPA*. Contiene una función para cargar el fichero del mapa del repositorio de mapas, y a partir de ahí generar parejas de puntos, que constituyen problemas concretos, para la realización de las pruebas, que consisten en ejecutar un número determinado de veces el algoritmo A*, y las fases de preprocesamiento, introducción de los puntos inicial y final, y búsqueda en el grafo abstracto y refinamiento de HPA*. Una vez finalizada la ejecución, se vuelcan los resultados del tiempo de ejecución, nodos expandidos en la búsqueda de la solución, longitud del camino obtenido, y el porcentaje de error en la solución de HPA*, sobre un fichero que posteriormente puede ser tratado desde MATLAB para imprimir gráficas.

Otra opción que contiene, y que nos ha permitido la depuración de errores, es, una vez cargado el mapa, pasarle como parámetros una pareja de puntos (inicial y final) concreta que se quiera analizar, para mostrarlo y visualizar la solución.

3.2.4 Mapa

Basándonos en la matriz dinámica de botones, se ha generado esta clase, que contiene una matriz de JButton para representar a los puntos del mapa. Para que las casillas sean cuadradas, y no rectangulares, se ha sustituido el uso de GridLayout por el de GridBagLayout, ya que permite un mayor nivel de configuración, como, en este caso, el tamaño de los elementos.

Dentro de él, existen métodos para la creación y eliminación de la matriz de botones, para la gestión del mapa y para el coloreado de sus casillas.

3.2.5 Punto

Es de las clases más recurridas en el programa. A cada botón dentro de la clase Mapa se le asigna un objeto de esta clase.

Contiene los heurísticos de las distancias Manhattan y octil, métodos para comprobar sus adyacentes (teniendo en cuenta el heurístico que va a aplicarse), y métodos para añadirle arcos a los puntos, entre otros.

3.2.6 Direccion

Esta clase simplemente contiene rutas a objetos utizados por el programa: imágenes, mapas…

3.2.7 Edge

Es una clase en que representa un arco entre dos puntos. En ella, se almacena tanto el camino, como el coste del mismo, además de dos objetos de clase Punto.

3.2.8 Arco

Es una clase que contiene una pareja de objetos de clase Punto. Representa de manera simplificada un arco, sin tener en cuenta el camino entre los dos puntos ni el coste del mismo.

3.2.9 Cluster

Representa los clusters en los que se divide el mapa en la fase de preprocesamiento del algoritmo HPA*. En esta clase, se almacenan sus dimensiones y un punto de inicio, que te indica la casilla en la que comienza.

Contiene métodos para obtener la sección que ocupa en el mapa, y los límites o fronteras en todas o en cualquier dirección, aparte de los clusters adyacentes.

3.2.10 Astar

En esta clase, se encuentran los métodos relacionados con el algoritmo A*. Para poder ejecutar este algoritmo, contiene una subclase Datos en la que se almacenan el coste, el punto actual, la longitud del camino y, en caso de existir, los datos del punto anterior. Además, la clase Astar contiene un objeto de la clase Timer para mostrar paso a paso el algoritmo en la interfaz, y variables para la gestión de sus estadísticas. Sus métodos se aplican en función de si se trata del propio algoritmo A*, de una llamada desde HPAstar para la fase de refinamiento de HPA*, o de una invocación desde la clase Test, o desde la clase Interfaz.

En la Figura 21(b), podemos ver el código de la subclase Datos, con todos sus parámetros.

```java
static class Datos {
    int coste;
    Punto p;
    Datos p_anterior;
    // Longitud del camino
    int longitud;

    /**
     * Crea los datos iniciales (sin datos anteriores)
     *
     * @param coste
     * @param p
     */
    public Datos(int coste, Punto p) {
        this(coste, p, null);
    }

    /**
     * Crea los datos a partir del número de pasos dado, del coste, del punto y de
     * los datos anteriores
     *
     * @param coste
     * @param p
     * @param anterior
     */
    public Datos(int coste, Punto p, Datos anterior) {
        this.coste = coste;
        this.p = p;
        p_anterior = anterior;
        longitud = anterior != null ? anterior.longitud + 1 : 1;
    }

    public Datos(int coste, Punto p, int longitud, Datos anterior) {
        this.coste = coste;
        this.p = p;
        this.longitud = anterior.longitud + longitud;
        p_anterior = anterior;
    }

    @Override
    public boolean equals(Object obj) {
        // Solamente comparamos que los puntos sean iguales
        return obj instanceof Datos ? ((Datos) obj).p.equals(p) : false;
    }

    @Override
    public int hashCode() {
        return p.hashCode();
    }

    @Override
    public String toString() {
        return this.p.toString();
    }

}
```

Figura 21(b) Subclase Datos.

3.2.11 HPAstar

En esta clase, se encuentran los métodos relacionados con el algoritmo HPA*. Cada fase del algoritmo está programada en métodos diferenciados, para facilitar la ejecución paso a paso. Para la creación de los arcos internos, realiza una llamada a la clase Dijkstra, y para la fase de refinamiento, a la clase Astar. Contiene variables internas para la gestión de sus estadísticas (iteraciones, memoria…) en cada fase y del algoritmo en sí, como puede ser la lista de clusters.

3.2.12 Dijkstra

En esta clase, se aplica el algoritmo de Dijkstra a un par de puntos, dado una subárea del mapa. Al igual que ocurre con Astar, utiliza una subclase interna Datos, con los mismos parámetros y métodos similares.

3.2.13 PanelArcos

Esta clase se utiliza para la representación visual del grafo abstracto. Para ello, emplea las clases y métodos relacionados con Graph y Graph2D.

3.3 Detección de errores

Como metodología para la depuración y validación del código de nuestro programa, hemos utilizado la clase Test, para, en primer lugar, obtener un fichero con resultados experimentales.

En cada línea de este archivo, se almacenan, en este orden, el par de puntos (inicial y final) que definen el problema, seguidos de los valores obtenidos por separado: longitud de la solución en ambos algoritmos, tiempo de ejecución en A* y en cada fase de HPA*, cantidad de nodos expandidos en cada algoritmo y porcentaje de error de la solución obtenida en HPA*.

En caso de obtenerse una solución para un único algoritmo, o de que los resultados sean excesivamente dispares, se ejecutan pruebas concretas para la pareja de puntos correspondientes en las que, finalmente, se muestra el mapa completo con la solución del algoritmo que necesitemos comprobar. Tras esto, se evalúa el código implementado para el algoritmo en el que se detecten los errores y se corrige.

4

Resultados experimentales

4.1 Datos empleados

Para conocer la bondad de HPA* con respecto a A*, realizamos una comparativa entre el tiempo de ejecución, los nodos expandidos y la longitud de la solución de un algoritmo con respecto al otro, además de comparar el tiempo empleado entre las distintas fases de HPA*, repitiendo así algunos de los experimentos del documento *Near Optimal Hierarchical Path-Finding* [1].

En concreto, ejecutamos unas doscientas pruebas sobre el mapa *AR0300SR* de *Baldurs Gate II* [16]. Esto nos ha permitido, aparte de realizar las comparativas mencionadas, detectar errores en el código y corregirlos.

Utilizando la clase Test, se interpreta el fichero *AR0300SR.map* del repositorio de mapas [15] para poder trasladarlo a nuestro programa y realizar las pruebas. Una vez completadas, se guardan los resultados obtenidos en un archivo que se traslada a MatLab, para crear a partir de ellos las imágenes de gráficas comparativas.

Cabe destacar, además, que para los tiempos del algoritmo HPA* se tienen en cuenta los valores obtenidos a partir de la fase de introducción de los puntos inicial y final, obviando la de preprocesamiento por ejecutarse una sola vez.

4.2 Comparativa entre A* y HPA*

En todas las comparativas, se tiene en cuenta la longitud de la solución óptima, es decir, la obtenida con A*. Este valor corresponde con la cantidad de nodos existentes en el camino desde el punto inicial hasta el final, y no tiene por qué corresponder con el coste total de la solución, ya que cualquier movimiento, incluidos aquellos en diagonal, supone sumar solamente una unidad en la longitud.

Para obtener los valores del tiempo de ejecución empleado, se realiza una llamada a *System.nanoTime()*, antes y después de la ejecución de cada algoritmo (para HPA*, se ejecuta por separado en cada una de sus fases, para obtener sus respectivos tiempos y poder evaluarlos individualmente, o se suman en caso de que se necesite estudiar globalmente el tiempo requerido por el algoritmo). Con esta llamada, se devuelve el valor del tiempo actual de la Máquina Virtual de Java en nanosegundos, por lo que cada vez que se obtienen, aparte de restar el valor final con el inicial, se divide entre 10^9 para pasarlo a segundos.

La Figura 22 muestra la comparativa entre el tiempo requerido por A*, en azul, y por HPA*, en naranja. En el eje Y de la gráfica, se nos muestra la longitud de la solución óptima, mientras que en el eje X, el tiempo de CPU requerido, en segundos. Esto permite, de manera visual, apreciar la relación existente entre ambos parámetros. Podemos observar que el tiempo empleado por A* crece de una manera cada vez más pronunciada a medida que aumenta la longitud de la solución, llegando a alcanzar casi 2.5 segundos en nuestras pruebas, mientras que en el total de HPA* el tiempo apenas se incrementa, tratándose en todo momento de milésimas.

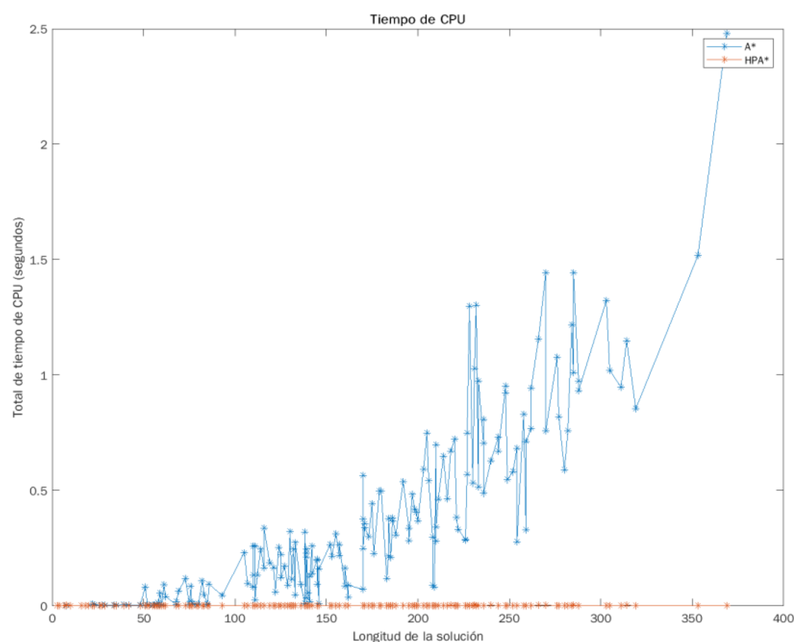


Figura 22 Comparativa de tiempo requerido por A* y HPA*.

Para obtener la cantidad de nodos expandidos, se tiene en cuenta el total de iteraciones que se ha necesitado hasta llegar a la solución.

La Figura 23 muestra, con los mismos colores, una comparativa entre la cantidad de nodos expandidos por cada uno de estos dos algoritmos. Nuevamente, en el eje Y se nos muestra la longitud de la solución óptima, mientras que en el eje X se nos muestra el total de iteraciones o nodos expandidos. Esta es una unidad que se utiliza porque representa una relación entre el tiempo y la memoria empleados. En este caso, podemos apreciar que el comportamiento es muy similar al que hemos visto en la gráfica anterior, creciendo el número de nodos explorados en mucho menor medida en HPA*, con respecto a A*.

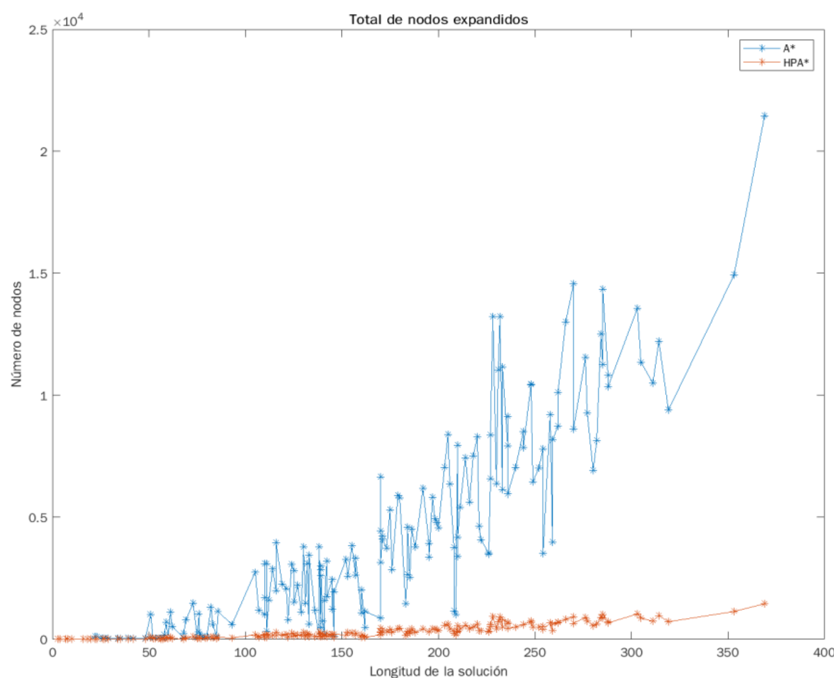


Figura 23 Comparativa de nodos expandidos por A* y HPA*.

La última prueba que compara estos dos algoritmos consiste en el cálculo del porcentaje de error en la solución obtenida por el algoritmo HPA*. En este caso, no solo se necesita la longitud de la solución óptima, es decir, la de A*, sino que también, la longitud del camino obtenido en HPA*, calculada de la misma forma. La fórmula que corresponde al porcentaje de error *e*, es la siguiente:

$$e = \frac{hl - ol}{ol} \; x \; 100$$

Donde *hl* es la longitud de la solución encontrada con HPA*, y *ol* es la longitud de la solución obtenida con A*.

En la Figura 24, podemos apreciar en color azul el porcentaje de error en el camino obtenido del algoritmo HPA*, comparando con la solución óptima, es decir, con el resultado de A*. En el eje Y aparece, nuevamente, la longitud de la solución óptima, mientras que el eje X representa el porcentaje de error. Podemos observar que para caminos muy cortos, el error obtenido puede ser muy elevado, aunque también existen casos en los cuales obtenemos un error del 0%, es decir, la longitud de la solución obtenida por ambos algoritmos de pathfinding es idéntica. Por lo general, y, sobre todo, a medida que aumentamos la longitud de la solución, el error mayoritariamente se aproxima a un 10%, obteniéndose finalmente un valor de error total medio de tan solo 7.4615%.

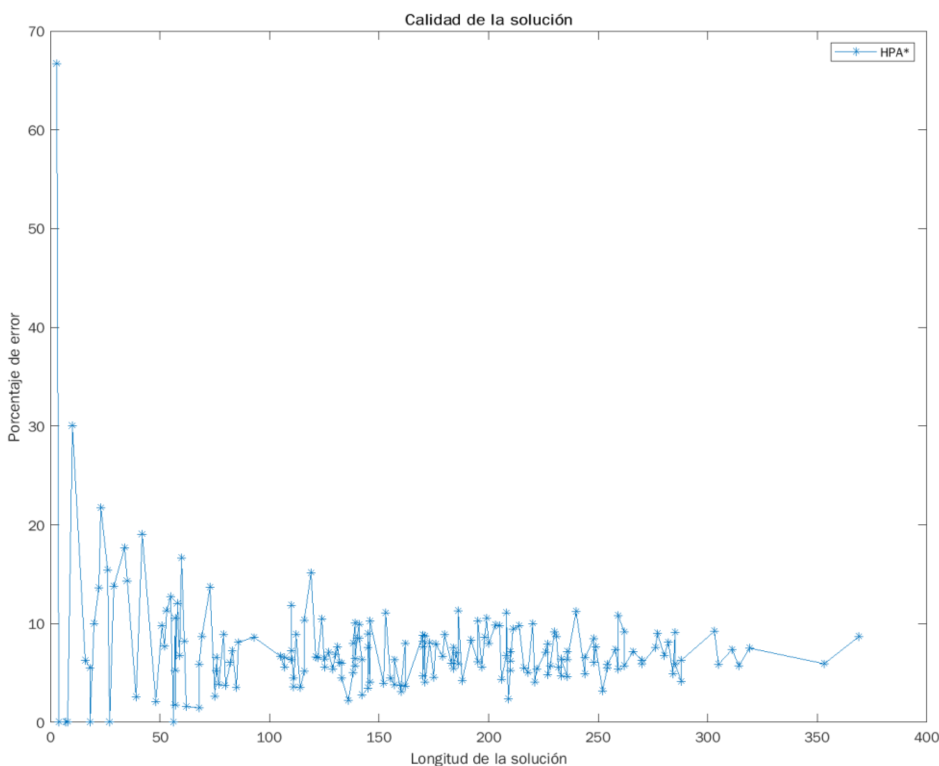


Figura 24 Porcentaje de error en la solución de HPA*.

4.3 Comparativa entre fases de HPA*

En este apartado, comparamos el tiempo de ejecución para cada fase dentro de HPA*. El valor obtenido en la fase de preprocesamiento es de 0.85675342 segundos, y no se representa gráficamente por ser constante, ya que solo se ejecuta una única vez,

y porque supera considerablemente, al tiempo empleado en las fases siguientes, cuyas sumas en intervalos no llegan a alcanzar un valor de 0.0025 segundos.

En la Figura 25, vemos una comparativa en el tiempo de ejecución obtenido en las fases de introducción de los puntos inicial y final, y en la de búsqueda en el grafo abstracto junto al refinamiento. Se han organizado estos resultados en intervalos, y podemos observar, el tiempo empleado en la búsqueda en el grafo abstracto junto al refinamiento aumenta a medida que crece la longitud de la solución. Por otro lado, observamos que el tiempo de introducción de los puntos inicial y final crece y desciende de manera aleatoria, y esto es debido a que depende del cluster en que se encuentre cada punto.

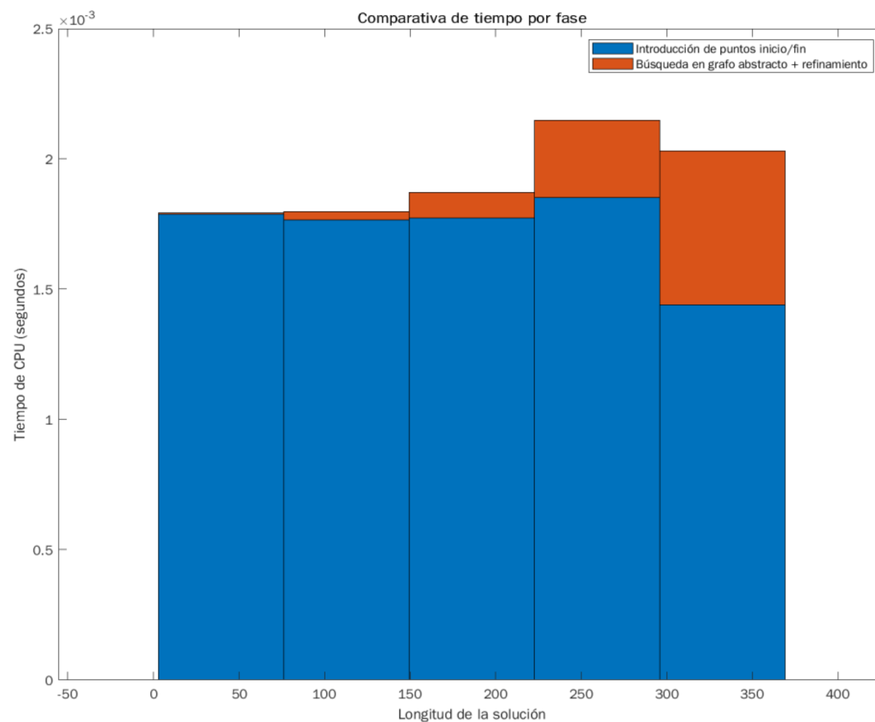


Figura 25 Comparativa de fases de HPA*.

5

Conclusiones y trabajo futuro

Para concluir con este proyecto, podemos decir que los objetivos propuestos se han completado satisfactoriamente: Se ha desarrollado en Java un simulador que permite al usuario la creación de mapas, cuya información puede ser almacenada y cargada a través de ficheros, y sobre los que más adelante pueden aplicarse los algoritmos de pathfinding A* y HPA*, cuyos comportamientos pueden verse paso a paso, además de la información de algunas estadísticas de consumo.

El desarrollo de este proyecto nos ha permitido estudiar con detenimiento y comprender mejor estos algoritmos, además de aprender acerca de programación de aplicaciones gráficas en Java.

Otra tarea realizada han sido pruebas que nos han permitido refinar el programa, y de cuyos resultados podemos deducir los beneficios del algoritmo HPA* frente a A*, sobre todo cuando los puntos de comienzo y objetivo están muy alejados: El algoritmo A* crece en cuanto a consumo de recursos de manera cuadrática, mientras que HPA* lo hace de una manera mucho menor, aunque seguramente también cuadrática, con un error medio en el resultado que no alcanza el 10% con respecto al camino óptimo, aunque mayor sobre todo si la distancia entre los puntos inicial y final es mínima. Dentro de HPA*, hemos observado que las fases de búsqueda en el grafo abstracto y refinamiento son las únicas realmente afectadas en lo que a consumo de tiempo se refiere cuando aumenta la longitud de la solución, ya que el tiempo en la fase preprocesamiento es siempre constante, al solo ejecutarse una vez, y la introducción de los puntos de inicio y de fin apenas se ve influido, ya que depende más de las características del cluster en el que se encuentren.

Una cosa que llama la atención en el algoritmo HPA* es la forma en que se crean los nodos. Podría disminuirse el error en la solución obtenida si en lugar de tener en cuenta únicamente los clusters adyacentes en dirección horizontal y vertical, se consideraran en diagonal, aunque esto aumente a priori el coste computacional. Así, por ejemplo, en vez de dar dos pasos para alcanzar un nodo que se encuentre en una de las diagonales, solo se daría uno.

De cara al futuro, podríamos comenzar a ampliar este trabajo desarrollando la fase de suavizado para HPA*. También, debería buscarse la forma de optimizar el código en las demás fases, en especial en la de preprocesamiento, ya que es la que tiene mayor coste computacional en el algoritmo. Dentro del programa, otro proceso con un gran peso computacional y que, por tanto, quizás requiere de una optimización es la creación de mapas, ya que al ejecutarse se consumen recursos considerablemente, de forma cada vez más notable a medida que aumentan las dimensiones de la matriz de botones. Además, queda pendiente adaptar los tamaños posibles de cluster a cada dimensión que se defina para el mapa y no únicamente permitir dos opciones comunes, independientemente del número de filas y columnas de la matriz.

Otra ampliación posible para el programa es la implementación de otros algoritmos de pathfinding, aprovechando la interfaz del simulador, para poder seleccionarlos y ver su ejecución paso a paso.

6

Bibliografía

[1] Botea Adi, Müller Martin y Schaeffer Jonathan (2004). *Near Optimal Hierarchical Path-Finding.* Department of Computing Science, University of Alberta Edmonton, Alberta, Canada. https://webdocs.cs.ualberta.ca/~mmueller/ps/hpastar.pdf

[2] Sutherland, Jeff, Ken Schwaber (noviembre de 2020). *SCRUM GUIDES.* https://scrumguides.org/

[3] Mandow, L. (7 de julio de 2010). *aida-uma.* Google code. https://code.google.com/archive/p/aida-uma/

[4] Oracle Corporation (24 de junio de 2020). *Java™ Platform, Standard Edition 7 API Specification.* https://docs.oracle.com/javase/7/docs/api/

[5] Torvalds, Linus Benedict (25 de mayo de 2022). *Git*. https://git-scm.com/

[6] Russel, Stuart J. y Norvig, Peter (2016). *Artificial Intelligence A Modern Approach.* (3ªed.). Editorial Pearson.

[7] Nilsson, Nils J. (2000), *Inteligencia artificial: Una nueva síntesis* (Trad. Marín Morales, Roque, Palma Méndez, José Tomás y Paniagua Aris, Enrique. Rev. Dormido Bencomo, Sebastián). Editorial McGraw-Hill.

[8] MathWorks (2022). *MATLAB Documentation*. https://es.mathworks.com/help/matlab/

[9] Tello, Javier (16 de junio de 2017). *Matriz dinámica de botones en java.* GitHub. https://github.com/DevJTello/MatrizBotones_Java

[10] Oracle Corporation (28 de febrero de 2022). *How to Use File Choosers.* The Java™ Tutorials. https://docs.oracle.com/javase/tutorial/uiswing/components/filechooser.html

[11] Oracle Corporation (28 de febrero de 2022). *How to Use Borders.* The Java™ Tutorials. https://docs.oracle.com/javase/tutorial/uiswing/components/border.html

[12] Roxas, Chuidiang (4 de febrero de 2007). *Ejemplos con Timer en java.* http://www.chuidiang.org/java/timer/timer.php

[13] GitHub, Inc. (2022). *GitHub Documentation.* https://github.com/

[14] Zhang, An, Li, Chong y Bi, Wenhao (18 de octubre de 2016). *Rectangle expansion A* pathfinding for grid maps.* ScienceDirect. https://www.sciencedirect.com/science/article/pii/S1000936116301182

[15] Stutervant, Nathan R. (verano de 2018). *Pathfinding Benchmarks.* Moving AI Lab. https://movingai.com/benchmarks/index.html

[16] Stutervant, Nathan R. (septiembre de 2000). *AR0300SR.png*. Moving AI Lab.

https://movingai.com/benchmarks/bgmaps/AR0300SR.png
[17] Programación en Castellano (s.f.) *Gráficos con Java 2D.* Programacion.net. Recuperado en marzo de 2022 de https://programacion.net/articulo/graficos_con_java_2d_111/10
[18] Naoki Takezoe (22 de agosto de 2022). *UML and ER-diagram editor for Eclipse.* GitHub. https://github.com/takezoe/amateras-modeler
[19] Chris3606 (13 de mayo de 2022). *Measuring Distance.* GitHub. https://chris3606.github.io/GoRogue/articles/grid_components/measuring-distance.html
[20] No Magic, Inc., a Dassault Systèmes company (2022). *MagicDraw Documentation.* https://docs.nomagic.com/display/MD190/MagicDraw+Documentation
[21] Tollkin (9 de febrero de 2018). *Blog Post Combat #2: Hexes vs. Squares.* Mod DB. https://www.moddb.com/news/blog-post-combat-2-hexes-vs-squares
[22] Patel, Amit (9 de enero de 2006). *Amit's Thoughts on Grids.* http://www-cs-students.stanford.edu/~amitp/game-programming/grids/
[23] Carlindo González, Carlos (24 de septiembre de 2009). *Búsqueda y funciones de evaluación heurística.* Gestiopolis. https://www.gestiopolis.com/busqueda-funciones-evaluacion-heuristica
[24] Instituto Tecnológico de Nuevo Laredo (2005). *ALGORITMO A*.* http://www.itnuevolaredo.edu.mx/takeyas/apuntes/Inteligencia%20Artificial/Apuntes/tareas_alumnos/A-Star/A-Star(2005-II-A).pdf
[25] Algoritmo de búsqueda A* (9 de enero de 2022). En *Wikipedia*. https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*
[26] Algoritmo de Dijkstra (6 de enero de 2022). En *Wikipedia*. https://es.wikipedia.org/wiki/Algoritmo_de_Dijkstra
[27] Roxas, Chuidiang (21 de diciembre de 2015). *JFileChooser.* Chuwiki. https://chuwiki.chuidiang.org/index.php?title=JFileChooser
[28] Franco García, Ángel (enero de 2000). *Las clases Color, Font y FontMetrics.* http://www.sc.ehu.es/sbweb/fisica/cursoJava/applets/grafico/color_font.htm
[29] Roxas, Chuidiang (28 de marzo de 2014). *Uso de Layouts.* Chuwiki. http://chuwiki.chuidiang.org/index.php?title=Uso_de_Layouts
[30] Oracle Corporation (28 de febrero de 2022). *A Visual Guide to Layout Managers* The Java™ Tutorials. https://docs.oracle.com/javase/tutorial/uiswing/layout/visual.html
[31] Roxas, Chuidiang (4 de febrero de 2007). *Ejemplo de uso del GridBagLayout.* http://www.chuidiang.org/java/layout/GridBagLayout/GridBagLayout.php
[32] Oracle Corporation (28 de febrero de 2022). *How to Use GroupLayout.* The Java™ Tutorials. https://docs.oracle.com/javase/tutorial/uiswing/layout/group.html
[33] Tello, Javier (15 de junio de 2017). *COMO HACER UNA MATRIZ GRÁFICA DE BOTONES EN JAVA.* PrograFácil. https://prografacilsite.wordpress.com/2017/06/15/como-crear-una-matriz-grafica-de-botones-en-java-dinamicamente/
[34] Sánchez, Carlos (8 de febrero de 2019). *Normas APA – 7ma (séptima) edición.* Normas APA (7ma edición). https://normas-apa.org/

Apéndice A

Manual de Usuario

7.1 Introducción

Este apéndice contiene el manual de usuario de la aplicación. En los siguientes apartados, se explicará cómo ejecutar el programa, y los distintos elementos y funcionalidades del mismo.

7.2 Ejecución del programa

Para el desarrollo de la aplicación, se ha utilizado el entorno Eclipse IDE. Si se desea ejecutar, deben añadirse a Eclipse todos los paquetes del programa, que puede descargarse en: https://github.com/davih22997/SimuladorHPAstar. Tras ello, se deberá, primero, seleccionar la clase *Simulador.java*, que se encuentra dentro del paquete *program*, y, después, pulsar el botón *Run Simulator*.

Por defecto, vendrá la opción de ejecutar el simulador, teniendo la clase *Simulador* instanciado un objeto de la clase *Interfaz*. Si se desea ejecutar la clase para pruebas experimentales, deberá instanciarse cambiarse el objeto de la clase *Interfaz* por uno de clase *Test*. A continuación, se explicarán las distintas partes del simulador, y al final del manual, se tratará la clase de pruebas.

7.3 Uso del simulador

El simulador depende directamente de la clase *Interfaz.java*, que cuenta con la interfaz del mismo, y cuyas partes y funcionalidades se explicarán a continuación.

7.3.1 Pantalla principal del simulador

Inicialmente, se muestra la interfaz con una preselección del algoritmo A* y de la modalidad 4-vecinos. Podemos apreciar las siguientes partes diferenciadas:

1. **Gestión de archivos.** Se sitúa en la parte superior de la interfaz. En ella, podemos ver tres botones, que corresponden, de izquierda a derecha, con el borrado de todos los datos, la apertura de archivos y el guardado de información en un fichero. A su derecha, hay una barra de texto en la que se muestra la ruta del último archivo abierto o guardado, salvo que se borren los datos.
2. **Selección y configuración del algoritmo**. Se sitúa en la parte izquierda, justo debajo de la gestión de ficheros. Corresponde con la selección del algoritmo que queremos simular y con la configuración de parámetros del mismo, los cuales difieren según el algoritmo seleccionado.
3. **Configuración del mapa**. Se sitúa en la parte derecha, justo debajo de la gestión de archivos. En ella, se seleccionan las dimensiones del mapa, y encontramos la parte de edición de mallas, la cual presenta distintos botones. Los primeros cuatro sirven para seleccionar qué hacer sobre el tablero del mapa: Consultar la coordenada del punto, y establecer el punto inicial, el final u obstáculos. Los dos siguientes, de izquierda a derecha, sirven para intercambiar el punto inicial por el final, en caso de estar definidos; y para eliminar los datos introducidos en el mapa, respectivamente.
4. **Control de la simulación**. Se encuentra justo bajo la parte de selección y configuración del algoritmo, a la izquierda de la interfaz. Difiere según el algoritmo seleccionado, pero principalmente corresponde con la parte de inicio o fin de la simulación
5. **Tablero del mapa**. Se sitúa en la parte central de la interfaz, y corresponde con el mapa sobre el que se introducen los puntos y en el que se muestra la simulación. Inicialmente está vacío hasta que se establecen las dimensiones del mapa.
6. **Registro de acciones**. Se sitúa en la parte inferior de la interfaz. En ella, se muestra por texto las distintas acciones que se van realizando.

En la Figura 26, podemos ver cómo se nos muestra inicialmente la interfaz y cada una de sus partes.

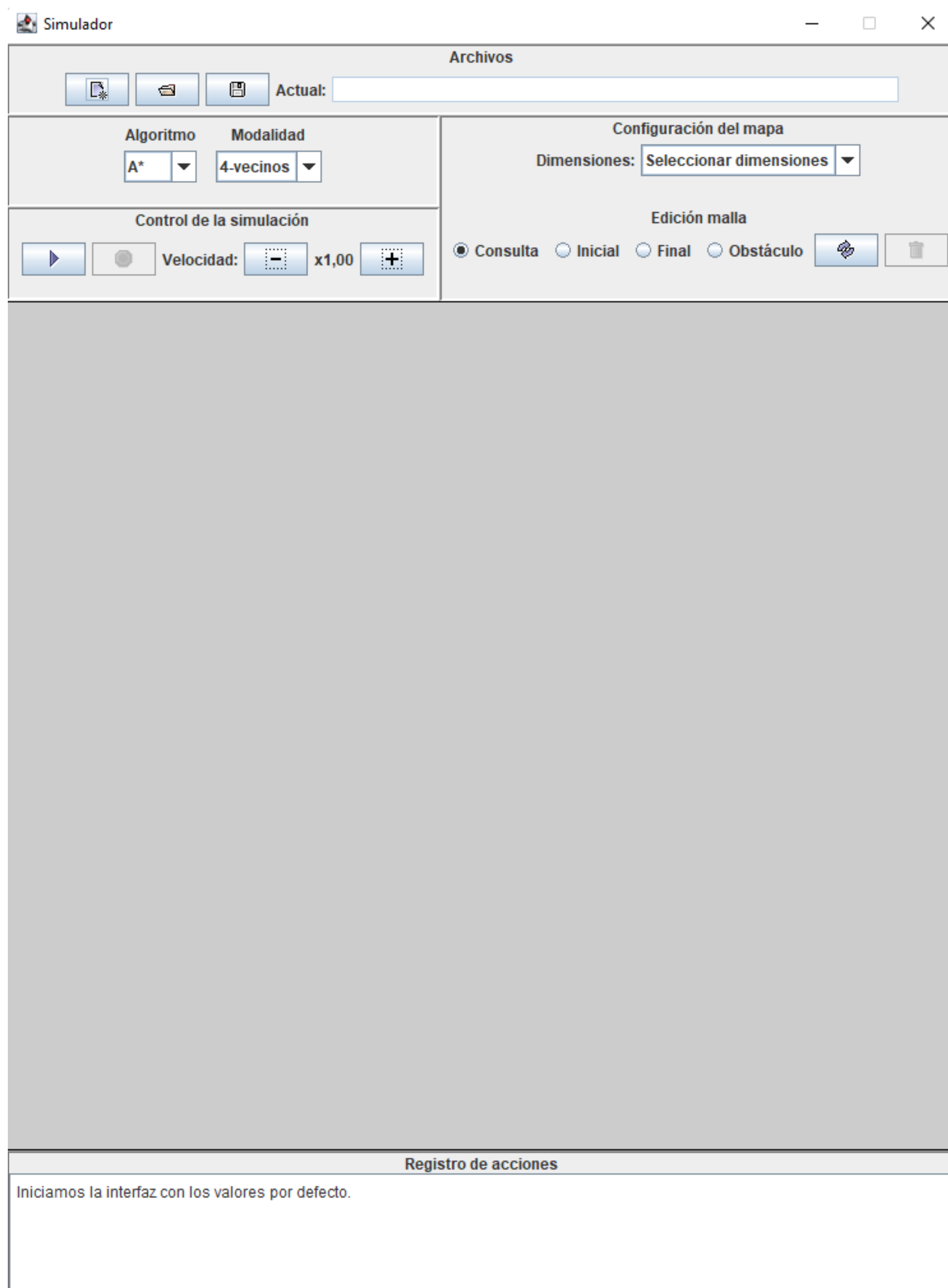


Figura 26 Pantalla principal.

7.3.2 Gestión del mapa

Esta parte influye en el mapa central. En la Figura 27, podemos ver su sección correspondiente.

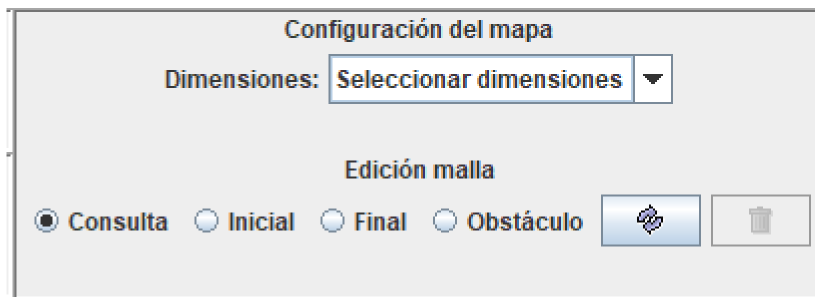


Figura 27 Parte de gestión del mapa.

Para empezar, deben seleccionarse las dimensiones del mapa, cuyas opciones disponibles se pueden ver en la Figura 28.

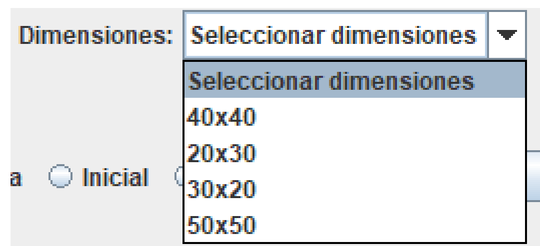


Figura 28 Dimensiones disponibles.

Una vez seleccionadas las dimensiones, podemos editar la malla. En la Figura 29, se muestra un mapa 20x30 vacío.

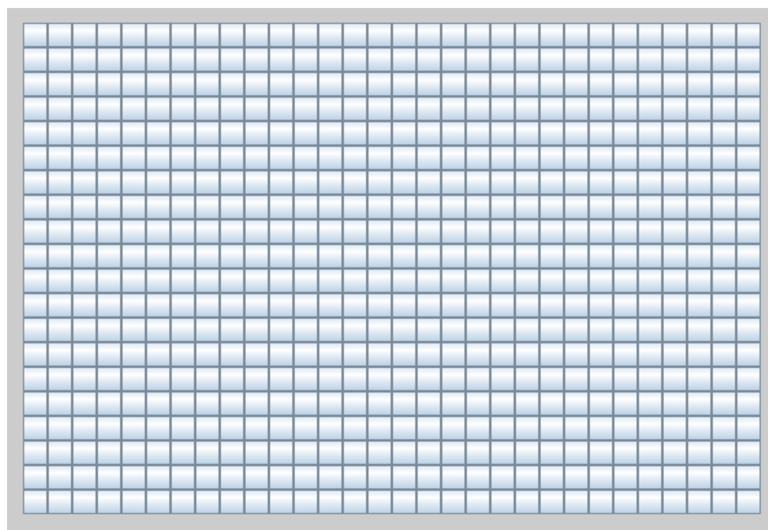


Figura 29 Mapa vacío de 20x30.

A través de los botones de edición de malla, vamos a interactuar con el mapa.

7.3.2.1 Consulta

Con esta opción, se nos muestra la coordenada del punto que se seleccione. En la Figura 30, podemos ver una consulta tras pulsar sobre el botón que corresponde con la fila 2, columna 9.

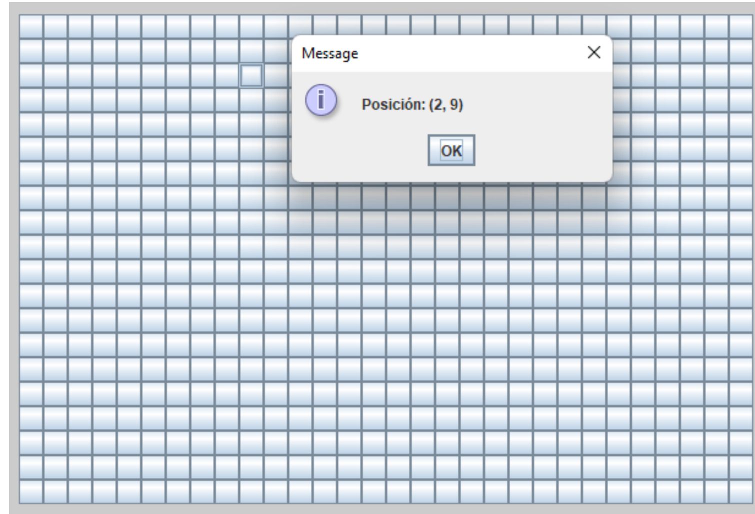


Figura 30 Consulta.

7.3.2.2 Inicial

Con esta opción, podemos seleccionar el punto inicial del problema, que quedará representado en el mapa con color verde.

7.3.2.3 Final

Esta opción se comporta de manera idéntica que la de selección del punto inicial, pero correspondiendo con el punto final, el cual quedará representado en color rojo.

7.3.2.4 Obstáculo

Con esta opción, se pueden seleccionar los obstáculos en el mapa, que quedan representados con color negro.

En la Figura 31, podemos ver el mapa con los puntos inicial y final, y algunos obstáculos definidos.

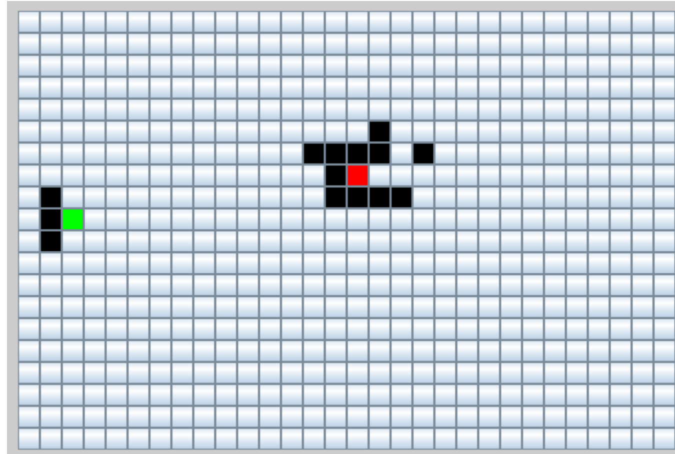


Figura 31 Mapa con puntos definidos.

7.3.2.5 Otras opciones

Las otras opciones restantes son la de intercambio de puntos inicial y final, y la de borrado del contenido del mapa. En la Figura 32(a) se ve el icono de intercambio, y en la Figura 32(b), el de borrado.

Figura 32(a) Botón de intercambio.

Figura 32(b) Botón de borrado.

7.3.3 Gestión de ficheros

Una vez conocida la forma de configurar el mapa, queda aprender a gestionar los ficheros. A continuación la Figura 33, se muestra la parte correspondiente en la interfaz.

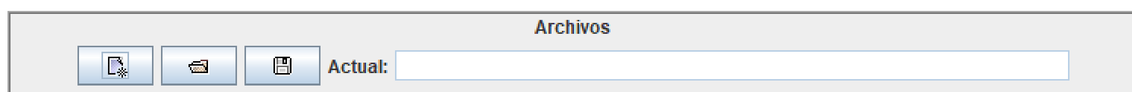


Figura 33 Parte de gestión de archivos

7.3.3.1 Formato del fichero

Los ficheros de este simulador se guardan en formato .txt por defecto, pero se puede establecer otro formato escribiendo otra terminación en su nombre.

A continuación, podemos ver en la Figura 34 la estructura con la que se definen sus datos.

```
Mapa:
Dimensiones: FILAS x COLUMNAS
Punto inicial: NULL | PUNTO
Punto final: NULL | PUNTO
Obstáculos: {} | {LISTA DE PUNTOS}
```

Figura 34 Formato de guardado de fichero.

No se puede cambiar el orden de los parámetros. Es obligatorio definir el tamaño del mapa. Si no se definen puntos, se escribe *NULL*, y si no se definen obstáculos, se escribe una lista vacía entre corchetes.

7.3.3.2 Guardar fichero

Tras definir como mínimo las dimensiones del mapa, puede guardarse un fichero con los datos del mismo. En la Figura 35, aparece el botón de guardar.

Figura 35 Botón de guardar.

Una vez seleccionada esta opción, se abre el menú para establecer la ruta y el nombre del archivo. En la Figura 36, puede verse este menú.

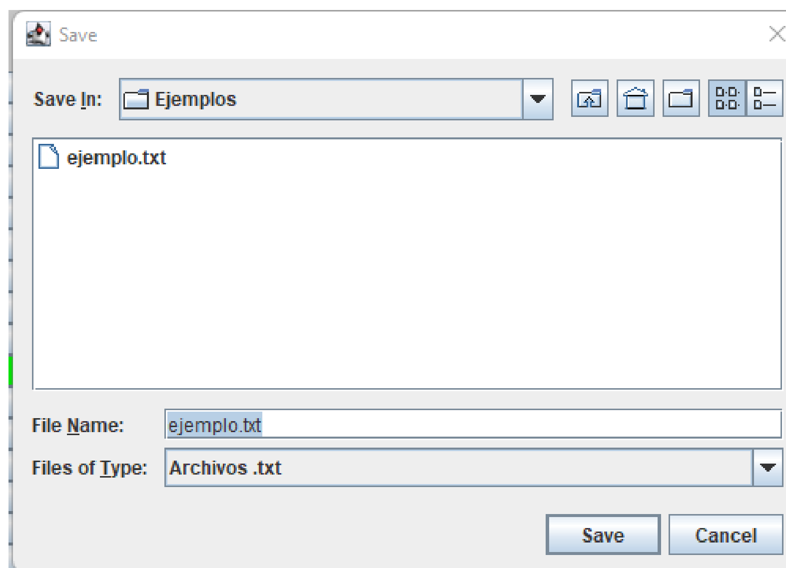


Figura 36 Menú de guardado.

Si se selecciona guardar sobre un archivo existente, aparece un pop-up de confirmación.

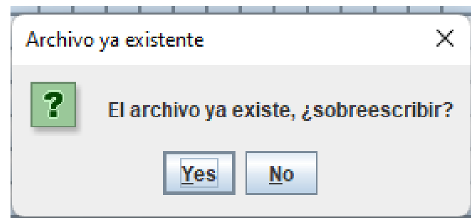


Figura 37 Pop-up de sobreescribir.

Una vez guardado el fichero, aparece su ruta en la línea de texto en la que se muestra el archivo actual.

7.3.3.3 Abrir fichero

Una vez se han almacenado los datos de un mapa en un fichero, este puede abrirse. Tras abrirse, aparece su ruta en la línea de texto en la que se muestra el archivo actual. En la Figura 38, podemos ver el botón de apertura de archivos.

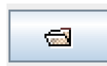


Figura 38 Botón de abrir.

7.3.3.4 Nuevo

Con esta opción, se elimina el texto de la ruta del fichero actual, y se establece la opción de *Seleccionar dimensiones* en las dimensiones del mapa, borrándolo. La Figura 38 muestra el botón correspondiente.

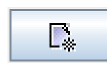


Figura 39 Botón de Nuevo.

7.3.4 Selección de algoritmo

Por defecto, viene seleccionado el algoritmo A*, aunque también puede seleccionarse el algoritmo HPA*. A continuación, va a explicarse qué ofrece cada uno de ellos.

7.3.4.1 Selección del algoritmo A*

Seleccionar A* como algoritmo permite acceder a su simulación. En la Figura 40, se aprecia la parte de la interfaz que se muestra de forma exclusiva tras la selección de A*

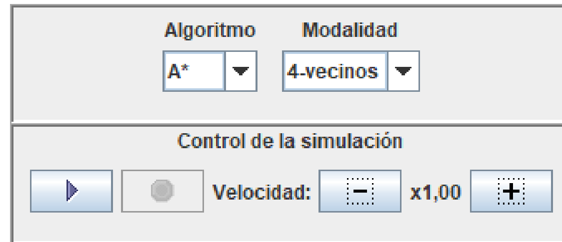


Figura 40 Parte mostrada con al opción de A*.

7.3.4.1.1 Modalidad

Con la opción del algoritmo A*, se ofrece un par de modalidades: 4-vecinos, que considera el movimiento en horizontal y en vertical, aplicando como heurístico la distancia de Manhattan a la hora de hallar al solución en A*; y 8-vecinos, que acepta también el movimiento en diagonal, y con el que se aplica la distancia octil. En la Figura 41(a), se muestra el camino obtenido con 4-vecinos usando como mapa el ejemplo de la Figura 31. En la Figura 41(b), se muestra el resultado obtenido con 8-vecinos usando el mismo mapa.

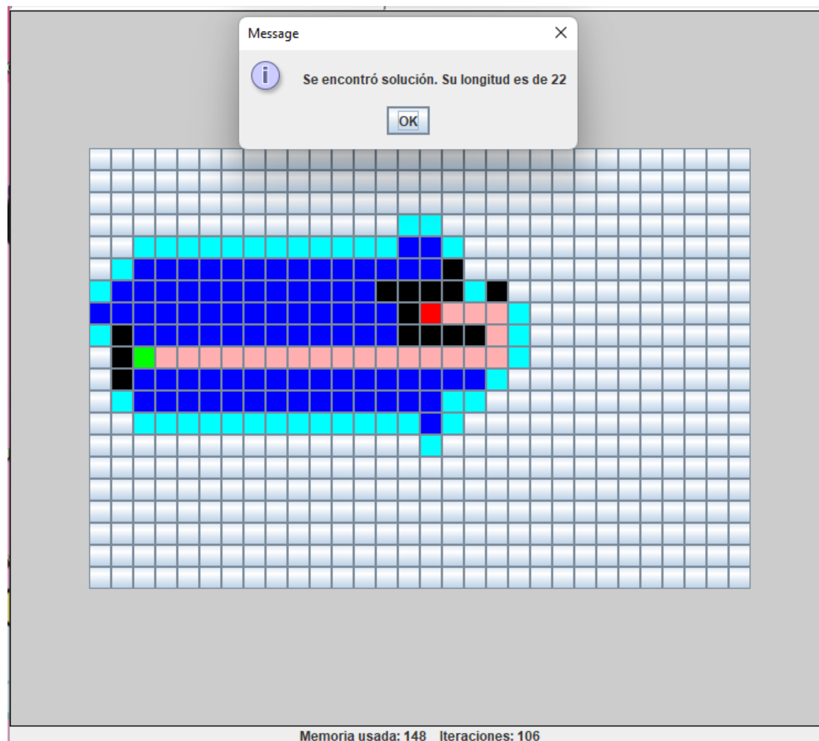


Figura 41(a) Solución con 4-vecinos.

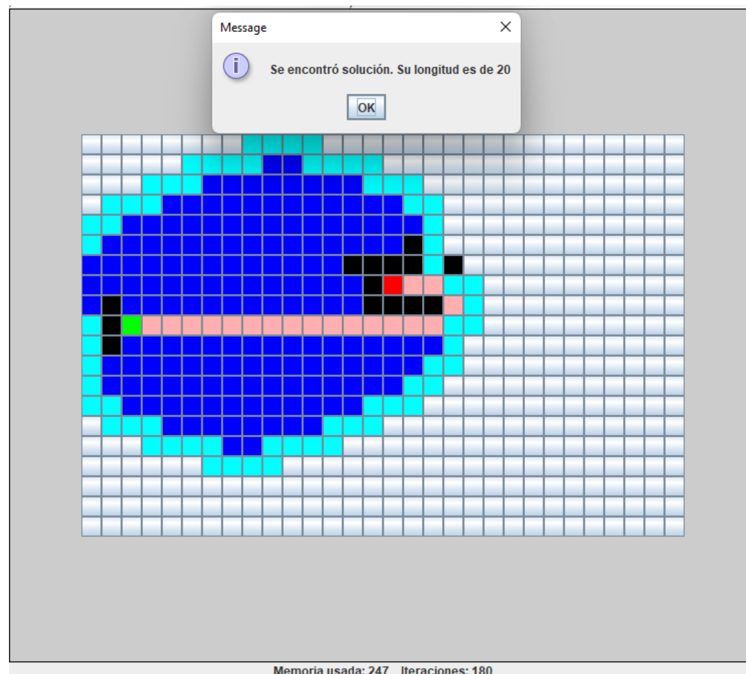


Figura 41(b) Solución con 8-vecinos.

En color salmón, se representa el camino resultante, en azul oscuro, los nodos cerrados, sin contar los de la solución; y en celeste, los nodos abiertos. Vemos que al aplicar distintos heurísticos, y permitir 8-vecinos un mayor número de movimientos

distintos, cambia la solución, y cantidad de memoria empleada e iteraciones para alcanzarla. El cómo mostrar esta simulación se explicará a continuación.

7.3.4.1.2 Control de la simulación

Las soluciones vistas en la Figura 41(a) y en la Figura 41(b) no habrían podido obtenerse sin controlar la simulación. En el apartado de control de simulación se permite lo siguiente:

1. **Iniciar la simulación.** Su selección es la más importante, ya que sin ella no se podría ver la simulación del algoritmo. En la Figura 42(a), se muestra el botón correspondiente.
2. **Pausar la simulación.** Permite congelar la simulación en el estado en el que se encuentre en ese momento, ya que la detiene de manera temporal. En la Figura 42(b), se muestra el botón correspondiente.
3. **Detener la simulación.** Finaliza la simulación, devolviendo el mapa a su estado antes de iniciarla. En la Figura 42(c), se muestra el botón correspondiente.
4. **Controlar la velocidad.** Permite mostrar con mayor o menor velocidad la simulación. Viene por defecto con valor *x1,*00. Si se pulsa el botón con el símbolo *+*, se duplica el valor, y si se pulsa aquel con el símbolo -, se divide entre 2. El máximo valor es de *x8,*00, y el mínimo, de *x0,*12. En la Figura 43, se muestra cómo se muestra este apartado por defecto.

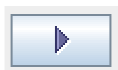


Figura 42(a) Botón de iniciar.

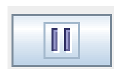


Figura 42(b) Botón de pausar.

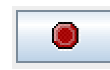


Figura 42(c) Botón de detener.

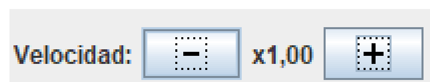


Figura 43 Control de velocidad por defecto.

7.3.4.2 Selección del algoritmo HPA*

Permite ejecutar la simulación del algoritmo HPA* paso a paso. En la Figura 44, se muestra la parte de la interfaz que aparece tras seleccionar este algoritmo, y que difiere de la del algoritmo A*.

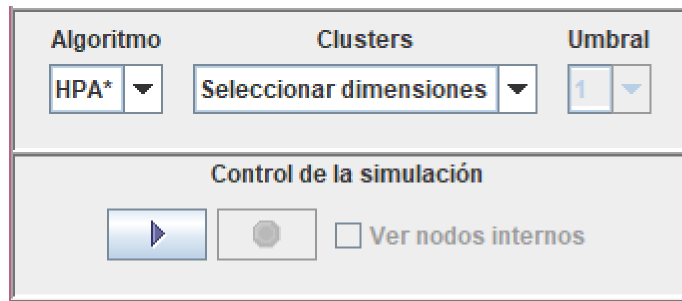


Figura 44 Parte mostrada con la opción de HPA*.

7.3.4.2.1 Selección de parámetros

Los parámetros que pueden seleccionarse son:

1. **Dimensiones de los clusters**. Define el tamaño (filas x columnas) de los clusters o secciones en las que se divide el mapa en la fase de preprocesamiento de HPA*. Las únicas opciones posibles en el simulador son 10x10 y 5x5, que pueden observarse en la Figura 45.
2. **Umbral**. Define el valor del umbral a aplicar a la hora de crear los nodos en la fase de preprocesamiento de HPA*. Esta opción permanece bloqueada hasta seleccionarse el tamaño de cluster. Su valor por defecto es 1, y su máximo corresponde a la mínima dimensión del cluster.

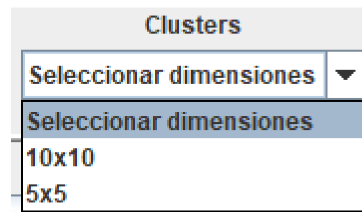


Figura 45 Dimensiones de clusters posibles.

7.3.4.2.2 Control de la simulación

Esta parte contiene los mismos botones que la opción del algoritmo A*, sin el control de la velocidad. Funciona de la misma forma para ambos algoritmos, pero con la diferencia de que inicialmente entre cada fase de HPA* solo pueden pulsarse los botones de iniciar y detener la simulación, hasta llegar a la fase de refinamiento, donde también puede pausarse. Los iconos pueden verse nuevamente en las Figura 42(a), Figura 42(b) y Figura 42(c).

7.3.4.2.3 Visualizar nodos internos

Su activación solo es posible tras la creación de los nodos y arcos. Sirve para mostrar la lista de nodos internos de un cluster tras pulsar cualquiera de sus puntos. En la Figura 46(a) puede verse cómo se muestra la opción, y en la Figura 46(b), una lista con los nodos de uno de los clusters del mapa de ejemplo de la Figura 31 y sus correspondientes arcos internos, a partir de unas dimensiones de cluster de 10x10 y de un umbral de 6.

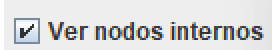
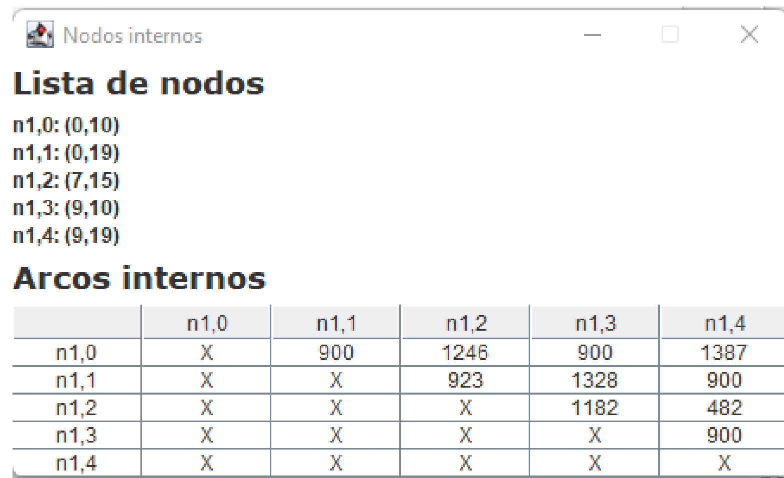


Figura 46(a) Opción de *Ver nodos internos*.



Nodos internos

Lista de nodos

n1,0: (0,10)
n1,1: (0,19)
n1,2: (7,15)
n1,3: (9,10)
n1,4: (9,19)

Arcos internos

	n1,0	n1,1	n1,2	n1,3	n1,4
n1,0	X	900	1246	900	1387
n1,1	X	X	923	1328	900
n1,2	X	X	X	1182	482
n1,3	X	X	X	X	900
n1,4	X	X	X	X	X

Figura 46(b) Lista de nodos y arcos.

7.3.4.2.4 Simulación por pasos

Debido a que HPA* cuenta con distintas fases, se explicará la forma en que se muestran en la simulación, que cuenta con 4 pasos, que se van ejecutando a medida que se pulsa el botón de iniciar la simulación. En la Figura 47, se muestra la definición completa del problema en la interfaz.

- **Primer paso**: Creación de los clusters en el mapa. En la Figura 48, podemos ver el resultado de este primer paso.
- **Segundo paso:** Creación del grafo abstracto. Incluye la introducción de los puntos inicial y final en el mismo, y activa la opción de *Ver nodos internos*. Los nodos se representan con un color más grisáceo. Podemos ver el resultado en la Figura 49.

- **Tercer paso:** Representación del grafo abstracto, que puede verse para nuestro ejemplo en la Figura 50.
- **Cuarto paso:** Corresponde con las fases de búsqueda en el grafo abstracto y refinamiento de HPA*. En este paso, se aplica A* para encontrar la solución en el grafo abstracto mostrando paso a paso cómo selecciona los nodos, además de la memoria usada y las iteraciones. En caso de hallarlo, nos muestra por pantalla la longitud del camino entre los puntos inicial y final. Podemos ver el resultado en la Figura 51.

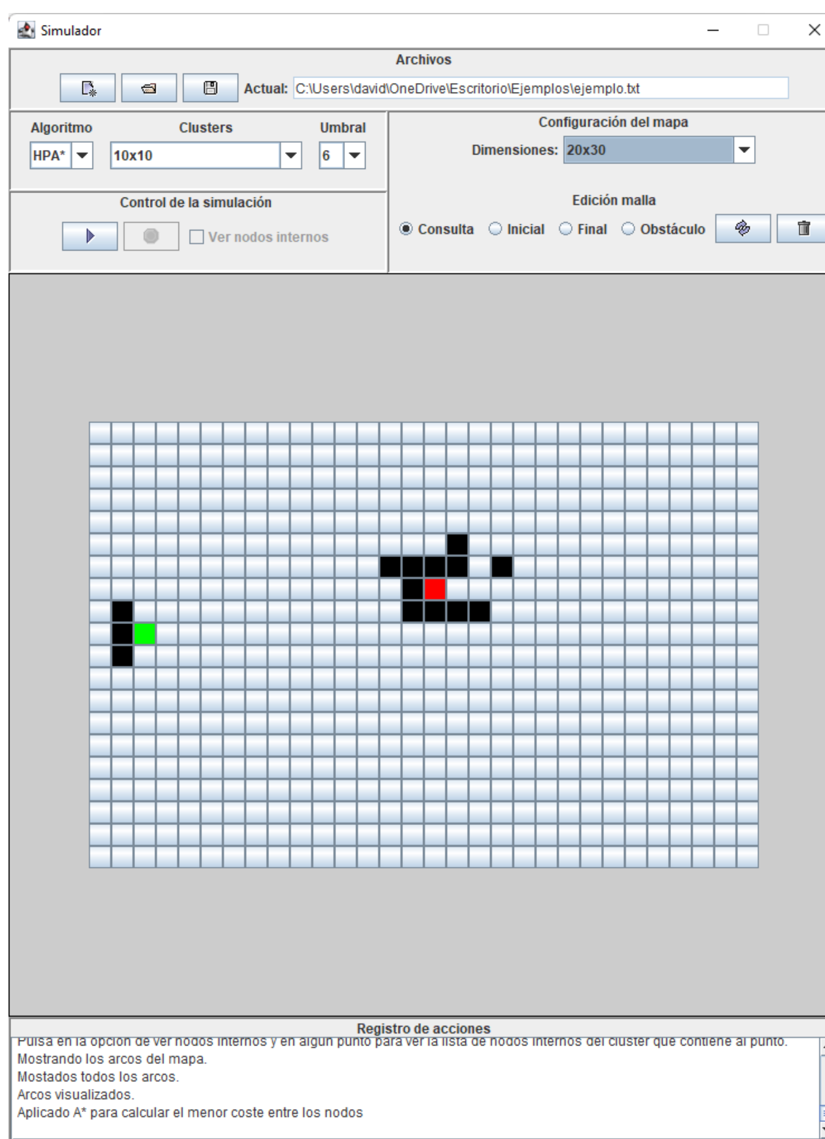


Figura 47 Definición del problema para la simulación de HPA*.

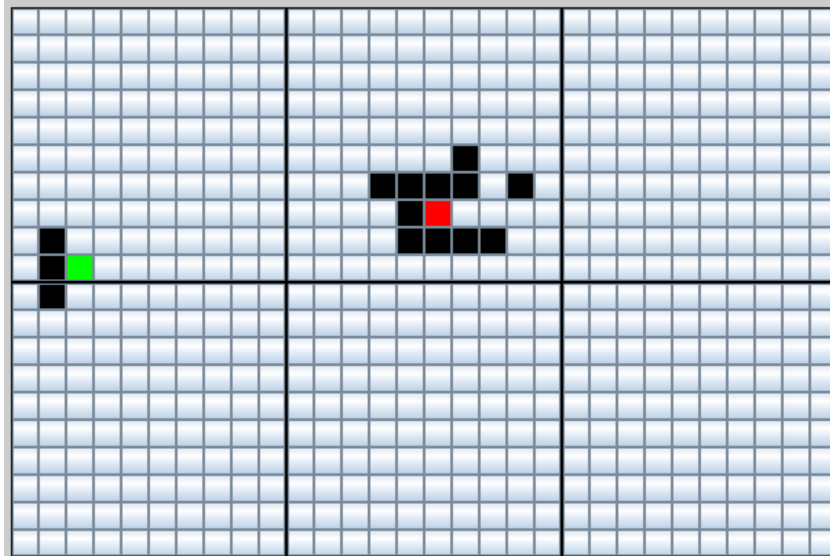


Figura 48 Primer paso en la simulación de HPA*.

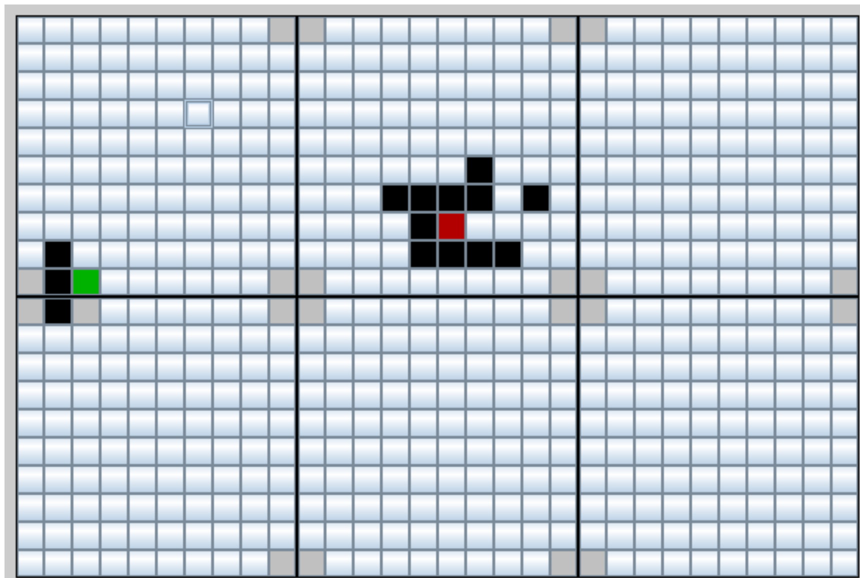


Figura 49 Segundo paso en la simulación de HPA*.

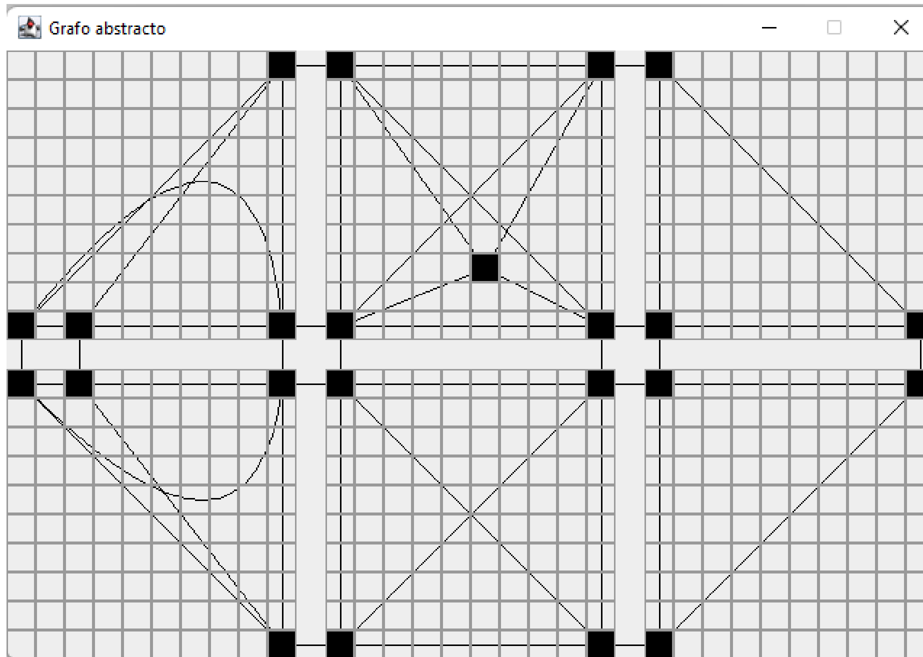


Figura 50 Tercer paso en la simulación de HPA*.

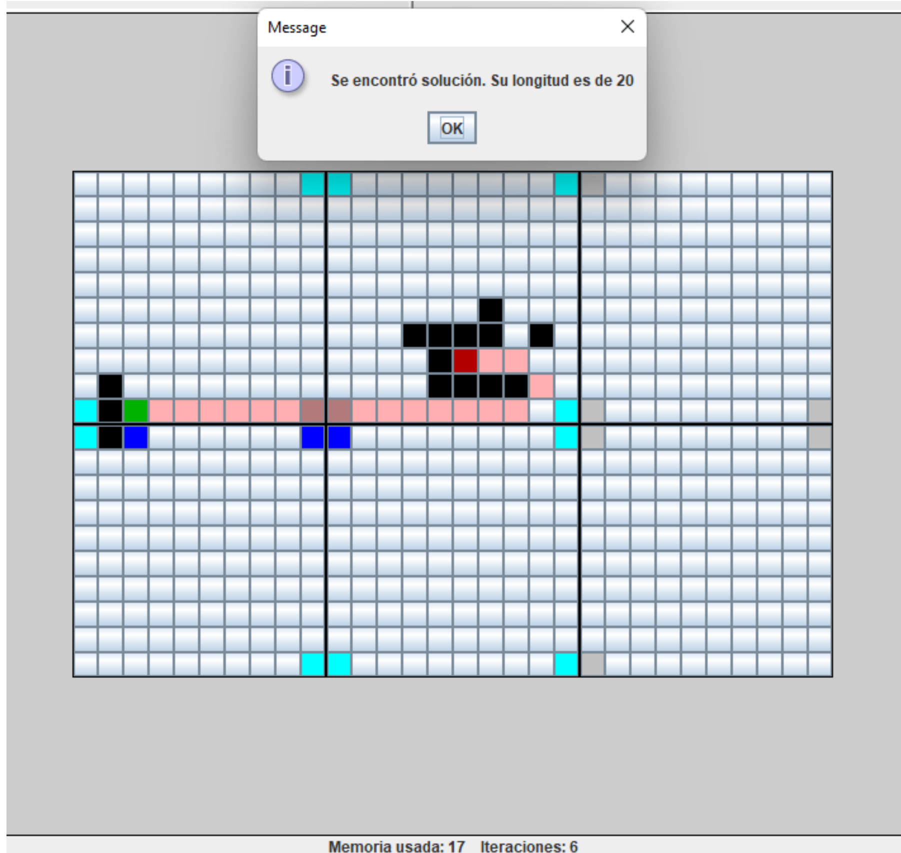


Figura 51 Cuarto paso en la simulación de HPA*.

7.4 Uso de la clase para pruebas experimentales

Una vez tratado el simulador, falta explicar la clase diseñada especificamente para obtener resultados experimentales *Test.java.* En ella, por defecto, se utiliza el mapa *AR0300SR* [16], que se llama desde la clase *Direccion*. El mapa puede sustituirse por otro, siempre y cuando sea una matriz cuadrada cuyas dimensiones sean divisibles entre diez, debido a que los clusters presentan unas dimensiones de 10x10 de manera predeterminada. Pueden cambiarse a 5x5 (y, por tanto, se dispone de la opción de utilizar mapas de dimensiones divisibles entre cinco), modificando la variable privada *dCluster* en la clase *Test.java*, y asignándole el valor de la constante *HPAstar.CLUSTER5x5,* en vez de *HPAstar.CLUSTER_10x10*.

Cabe destacar que la clase Test cuenta con dos modalidades: la de realización de pruebas aleatorias, para generar datos sobre un fichero, y la de pruebas específicas, que sirven para la comprobación del correcto funcionamiento de un algoritmo dados dos puntos específicos, y que se explicarán a continuación.

7.4.1. Realización de pruebas aleatorias

En esta modalidad, se generan *NPRUEBAS* pares de puntos inicial y final distintos, pero aleatorios, para probarlos con los algoritmos A* y HPA* y generar datos que se retornan sobre un fichero.

Para realizar estas pruebas, simplemente debe instanciarse a la clase *Test* sin parámetros, o teniendo como parámetros de entrada la variable estática *Test.MODO_GRAFICAS* y un par de objetos de clase *Punto* cualesquiera, incluyendo *null*. Si se desea modificar el número de pruebas que se realizan, simplemente debe modificarse la variable estática *NPRUEBAS* dentro de la clase *Test*.

Una vez finalizadas las pruebas, los resultados se vuelcan sobre el archivo *datos.txt*. En caso de no encontrarse solución para un par de puntos, se devolverá el valor *NaN* en todos sus resultados dentro del archivo, para no tenerlos en cuenta. A partir de este fichero, pueden imprimirse gráficas en MATLAB usando el script *graficas.m*, que se encuentra en el paquete *matlab*.

7.4.2. Realización de pruebas específicas

En esta modalidad, se muestra el resultado obtenido de aplicar uno de los dos algoritmos sobre el mapa. Debido a que estas pruebas están diseñadas para ver el camino solución y al reducido tamaño de las casillas, no se muestran pintados los bordes de los clusters en HPA*.

Para realizar una prueba, debe instanciarse a la clase *Test*, teniendo como parámetros *Test.MODO_ERROR_Astar* o *Test.MODO_ERROR_HPAstar*, y un par de objetos de la clase Punto, cuyas coordenadas sepamos que se encuentran dentro del mapa. Una vez especificados los parámetros, se mostrará por pantalla el mapa con la solución al problema, o se indicará que no se ha encontrado. Los puntos inicial y final, se muestran en color verde y rojo, respectivamente. En color salmón, se muestra el camino, en azul claro los nodos abiertos, y en azul oscuro, los nodos cerrados.

En la Figura 51(a) y la Figura 51(b), se muestra la solución dados los puntos (35, 148) y (267, 86), como inicial y final, respectivamente, para el algoritmo A*.

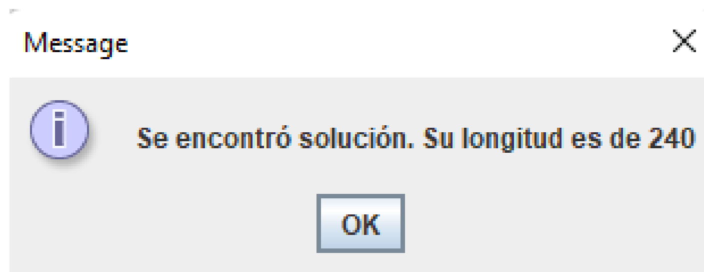


Figura 51(a) Texto del resultado en A*.

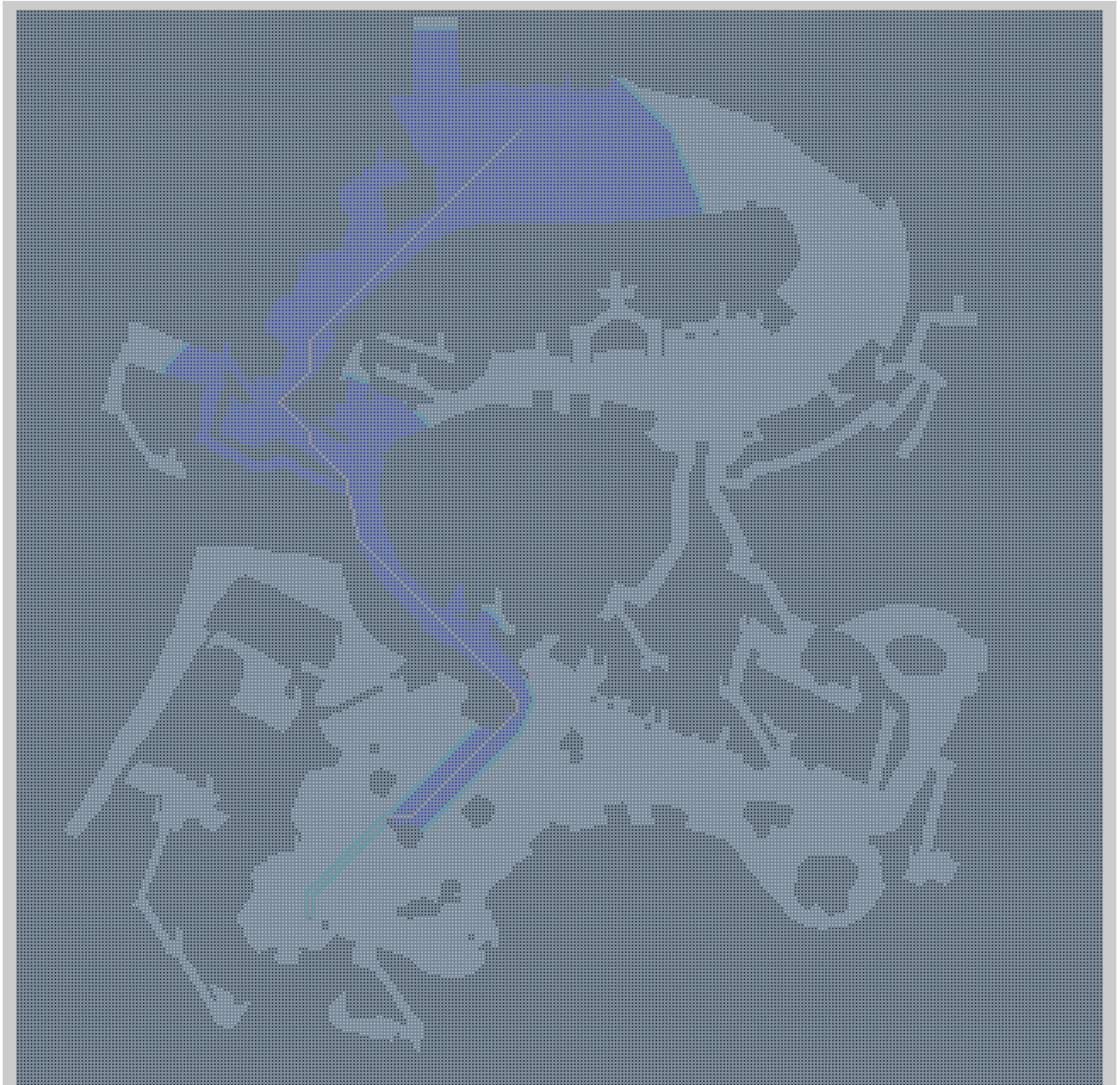


Figura 52(b) Mapa con el resultado en A*.

En la Figura 53(a) y la Figura 53(b), vemos el resultado obtenido usando los puntos anteriores para el algoritmo HPA*. La representación del camino y los nodos abiertos y cerrados viene dada con los mismos colores, pero además, se añaden en tono grisáceo los nodos de dentro de cada cluster.

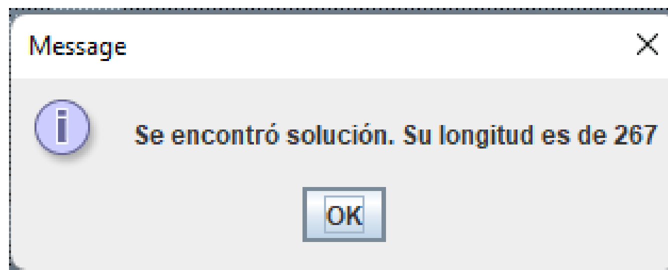


Figura 53(a) Texto del resultado en HPA*.

Figura 53(b) Mapa con el resultado de HPA*.

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