Trabajo de Fin de Grado

Búsquedas gráficas en mapas de juegos

Grado en Ingeniería Informática 2015/2016

Mª Isabel Peña Martín

Escuela Politécnica Superior

Tutor: Daniel Borrajo Millán

Agradecimientos

A mis padres y mis tíos, por todo su apoyo en estos años, en lo académico y lo personal, y sin los que no habría podido llegar hasta aquí.

A Álvaro por conseguir que cada día dé lo mejor de mí misma, y nunca deje de esforzarme. Por creer siempre (o casi siempre) que soy capaz de lograrlo todo. Porque todos han aguantado con paciencia mis momentos de nervios.

Muchas gracias a todos mis amigos, en especial a Lucía, Cristina y Sunny por mis constantes negativas a salir de la biblioteca. A Paloma y a Jon por la ayuda prestada. A Agus y mis "Tigres", con los que he aprendido que con esfuerzo y constancia podemos alcanzar nuestros sueños.

Agradezco a la Universidad Carlos III y a todos mis profesores los medios aportados durante mis años de carrera para conseguir el conocimiento que me ha ayudado en este trabajo. Sobre todo a mi tutor, Daniel, por resolver todas mis dudas y responder siempre lo antes posible.

Resumen

En este documento va a presentarse un trabajo sobre el uso de algoritmos de búsqueda en mapas de dos dimensiones en videojuegos como trabajo de fin de grado. Muchos videojuegos suelen hacer uso de mapas para el movimiento de los personajes. Para calcular el camino a recorrer en el mapa de estos personajes es básico poder buscar caminos en esos mapas de forma eficiente, lo más rápido posible.

Se va a diseñar una solución que divide los mapas en regiones y genera un grafo con las conexiones entre ellas. Con este grafo se hace una primera búsqueda entre las regiones por las que pasará el camino y después una búsqueda del camino por los puntos del mapa que pase por estas regiones.

Se explicará el estado del problema y se analizarán posibles soluciones presentando la elegida así como las pruebas realizadas y las conclusiones sobre los resultados. También se incluye un apartado con la planificación del proyecto, el tiempo invertido en el mismo y un presupuesto si el trabajo tuviera un carácter comercial. El trabajo cierra con un resumen en inglés del documento, los anexos, referencias y bibliografía.

Contenido

Ilustraciones

Tablas

1. Introducción

El uso de los ordenadores ha permitido resolver en periodos cortos de tiempo problemas que de otro modo requerirían muchos recursos humanos y temporales, llegando algunos problemas a no poder ser resueltos hasta la invención de las computadoras. En computación, dos objetivos fundamentales son encontrar buenos algoritmos que resuelvan un problema en poco tiempo y encuentre buenas soluciones.

Se denomina heurística al conocimiento parcial sobre un dominio que permite guiar la búsqueda de la solución a problemas. Las heurísticas se aplican en problemas donde no es fácil encontrar la solución, pero su aplicación no garantiza que se encuentre siempre. Algunas soluciones heurísticas pasan por la relajación de las restricciones de un problema obteniendo aproximaciones a la solución. La búsqueda heurística se aplica al problema de obtener caminos de un estado inicial a uno final. Mediante el uso de una función heurística, que estima la distancia a la solución desde un estado intermedio y aporta información sobre el mejor estado por el que continuar para alcanzar la solución, la búsqueda se hace óptima mediante una función que sea admisible.

Muchos videojuegos contienen mapas sobre los que se desarrolla la acción de los personajes. Estos mapas se almacenan en memoria y se muestran al jugador procesados y detallados mientras que el ordenador trabaja con modelos más simples que representan el mapa. Los mapas se recorren tanto por jugadores como por NPC's. El movimiento de los NPC's está controlado por la computadora y por tanto su recorrido debe calcularse en el tiempo de ejecución del propio juego.

Los caminos que se recorrerán en el juego pueden estar calculados, haciendo que el personaje siempre realice el mismo recorrido o pueden calcularse cada vez dependiendo de si se quiere alcanzar un objetivo distinto cada vez o frente a escenarios cambiantes. Esto aumenta la carga de cómputo del ordenador y es mucho mayor cuanto mayor sea el mapa, y en este caso también deberá tenerse en cuenta la memoria ocupada por el mapa. Es importante el uso de algoritmos eficientes para reducir esta carga de trabajo y además conseguir soluciones óptimas para aumentar el nivel de inteligencia artificial de los personajes.

Algunas soluciones pasan por la aplicación de algoritmos de búsqueda informada que hacen uso de información previa sobre el mapa cargado en memoria. Es importante mantener un equilibrio entre memoria utilizada y la mejora en el cómputo. Habrá datos que puedan calcularse durante el juego sin comprometer la fluidez del juego y nos permitirá tener más memoria libre para otros datos del mismo modo que habrá cálculos que sean muy pesados durante el desarrollo de una partida y sea mejor tenerlos calculados en memoria.

Para este trabajo se ha aplicado el método de búsqueda heurística para la búsqueda de caminos entre dos puntos de un mapa con formato cuadriculado. La idea surge de la competición anual [1] que se celebra sobre este tema. El objetivo será el de aplicar algoritmos de búsqueda para encontrar los caminos y estudiar las soluciones obtenidas. En este documento va a recogerse toda la información asociada al trabajo. Desde el análisis previo y la elección de algoritmos hasta el desarrollo de la solución, la interpretación de las soluciones y la estimación final de tiempo y coste.

2. Estado del Arte

2.1.Videojuegos

Un videojuego es un juego electrónico que se visualiza en una pantalla, se reproduce desde una consola o un computador y permite la interacción del jugador con el programa mediante mandos, teclado o movimientos. Los videojuegos surgieron a mediados de los años 50 como intento de dar un uso lúdico a las computadoras. El que se considera el primer videojuego es Tenis for Two [2] creado en 1958 por William Higginbotham, aunque existen muchos programas con fin lúdico de la época, como OXO [3]. Al inicio eran programas muy simples y que no se pensaron para su comercialización, que no llegaría hasta los años 70 con la fabricación de las primeras máquinas recreativas, como Pong de Atari [4]

Desde entonces los videojuegos han evolucionado en complejidad, historia, duración, jugabilidad y han crecido enormemente en popularidad, siendo hoy en día una de las industrias que más dinero y empleo genera en el mundo. Se han creado videojuegos para llegar a toda la sociedad desde los géneros más casuales y familiares hasta los grandes videojuegos con historias más complejas. También se ha extendido a todas las plataformas como los smartphones no siendo programas exclusivos para ordenadores o consolas fijas o portátiles.

Los videojuegos se desarrollan sobre un escenario y este puede ser simple como una imagen de fondo, un tablero o un terreno sobre el que se moverán los personajes. En estos casos existe un mapa del lugar que se muestra al jugador y que el ordenador utiliza para calcular los caminos hacia ciertos objetivos.

2.2.Mapas

Al igual que los juegos los mapas han ido creciendo en complejidad y tamaño, siendo en muchas ocasiones un elemento clave para el triunfo del juego. No se trata sólo del fondo del juego si no del escenario sobre el que se moverán los personajes. Están almacenados en memoria y supone un reto su representación y la búsqueda de caminos en ellos, sobretodo en el caso de los más grandes. Es diferente el modo en que se muestran estos mapas a los jugadores con respecto a los mapas con los que trabajará el ordenador.

El jugador verá la representación gráfica y procesada del mapa mediante formas, texturas, fondos y detalles tridimensionales. El ordenador trabaja con otras formas de representación como formas geométricas que dividen el mapa y se trabaja mediante los vértices o los lados de estas formas. Los movimientos se pueden representar mediante grafos y dependerán de la forma del mapa. Se pueden permitir movimientos horizontales y verticales además de los movimientos diagonales que permitirán a los personajes moverse en todas las direcciones.

Los mapas pueden dividirse además en regiones interconectadas para los mapas más grandes mediante puntos denominados “waypoints” que a su vez están conectados y representan un espacio de búsqueda ente las regiones. La complejidad de la búsqueda

aumenta conforme aumenta la complejidad del mapa con el tamaño, la dimensión en la que se realizan los movimientos y el número de movimientos permitidos.

2.3.Algoritmo A*

Se trata de un método de búsqueda en grafos. Fue presentado en 1968 por Peter E. Hart, Nils J. Nilsson y Bertham Raphael [5]. Este algoritmo encuentra el camino de menor coste entre un nodo origen y uno objetivo, siempre que se cumpla que existe un posible camino con un factor de ramificación finito, los costes sean homogéneos y que la función heurística sea admisible.

Algunos algoritmos de búsqueda se guían exclusivamente por la función heurística, lo que puede no indicar el camino de menor coste o la distancia más corta a la solución. A* utiliza una función de evaluación de la forma $f(n) = g(n) + h(n)$ donde $g(n)$ representa el coste del camino recorrido desde el nodo inicial al nodo n y $h(n)$ el valor de la función heurística desde n hasta el nodo final. El algoritmo mantiene dos estructuras de datos auxiliares donde se van expandiendo y calculando los costes de los nodos: una lista que llamaremos abierta en la que estarán los nodos a visitar ordenados por prioridad según el valor de la función de cada nodo; y otra lista a la que llamaremos cerrada donde estarán los nodos ya visitados. En cada paso del algoritmo se selecciona el primer nodo de la lista abierta y, si no es el nodo objetivo, se expanden sus posibles nodos hijo calculando su $f(n)$, estos nodos pasan a la lista abierta mientras que el nodo evaluado se inserta en la lista cerrada.

A* es un algoritmo de búsqueda en amplitud y es considerado completo: si hay solución, dará con ella. Para garantizar que es óptimo la función $h(n)$ debe ser admisible, esto es que represente el coste real de la solución y que nunca decrece según se avanza en el grafo.

2.4.Algoritmo Dijkstra

El algoritmo de Dijkstra se aplica para la determinación del camino más corto dado un nodo origen al resto de los nodos en un grafo con pesos en cada conexión. Su nombre se refiere a Edsger Dijkstra, quien lo describió por primera vez en 1959 [6].

La idea de este algoritmo consiste en ir explorando todos los caminos más cortos que parten del nodo origen y que llevan a todos los demás nodos; cuando se obtiene el camino más corto desde el nodo origen, al resto de nodos que componen el grafo, el algoritmo se detiene.

Para su funcionamiento el algoritmo parte de un nodo origen con una lista abierta de nodos a evaluar. Desde el nodo inicial se comprueban los nodos vecinos y se actualiza la distancia a cada uno de esos nodos. Después se escoge aquel con el menor coste y se avanza expandiendo los nodos vecinos del seleccionado, actualizando la distancia desde el nodo inicial a cada uno de estos nodos, pero sumando todos los costes hasta ese nodo. Para el siguiente paso se selecciona de nuevo el camino de menor coste. Esto se repite hasta alcanzar todos los nodos mediante el camino de coste mínimo a cada uno de ellos.

2.5.Grid-Based Path Planning Competition

Actualmente existe una competición sobre la aplicación de algoritmos de búsqueda para crear caminos en mapas. En esta competición se busca crear un algoritmo que sea rápido y eficiente, que nos permita generar caminos en la menor cantidad de memoria y tiempo posible. En su web [1] pueden conocerse las restricciones de la competición y las normas para participar.

La competición comenzó en 2012 y desde entonces un gran número de investigadores y programadores de forma individual han participado en ella. Esto ha permitido que actualmente se disponga de un gran número de estudios y aplicaciones de algoritmos de búsqueda gracias a la motivación generada por la competición. Cada 1 de diciembre comienza una nueva competición anual pero los participantes pueden unirse en cualquier momento del año.

2.6.Moving AI Lab

Este laboratorio tiene su sede en la Universidad de Denver y su director es el profesor Nathan Sturtevant. Aquí se fundó la competición del Grid-Based Path Planning y en su web [7] pueden consultarse los documentos escritos por los participantes, las distintas convocatorias y los ganadores. También hay vídeos y documentos explicativos sobre el funcionamiento de algunos algoritmos de búsqueda.

Además del estudio de búsquedas en mapas en este laboratorio se realizan investigaciones acerca del uso de la Inteligencia Artificial y aplicaciones en distintos juegos como el cubo de rubik o el desarrollo de un pinball multijugador.

3. Objetivos

El objetivo principal de este trabajo es desarrollar técnicas de búsqueda que permitan encontrar caminos en mapas cuadriculados en dos dimensiones dados un punto inicial y final. Estas técnicas deberán tener equilibrio entre el tiempo de búsqueda y la memoria necesaria.

Como se ha comentado en la introducción, para la búsqueda informada es bueno tener cierta información almacenada en memoria para realizar de manera eficiente este proceso de búsqueda. Como en el trabajo previo de Álvaro Parra, Álvaro Torralba y Carlos Linares López [8] se hará una clasificación previa de regiones de los mapas en precomputo y el cálculo de las distancias que las separan.

Los subobjetivos serán conseguir la función que clasifique un mapa en regiones y localice las conexiones entre dos regiones vecinas o waypoints. Mediante esos waypoints se trazará un grafo que represente las conexiones y así encontrar un camino entre las regiones.

Con el grafo de regiones se hará una primera búsqueda de las regiones por las que debe pasar un camino desde la región del punto inicial a la región del punto final. Una vez obtenido este camino se realizará otra búsqueda entre cada par de puntos de las conexiones entre regiones a nivel de puntos del mapa desde el inicio hasta la meta. La solución será una lista ordenada de puntos del mapa que representen el camino a seguir.

Esta búsqueda se realizará mediante un algoritmo de Dijkstra para las regiones y un algoritmo A* para el camino final. En el programa a realizar se implementarán las estructuras de datos necesarias para almacenar las regiones y las conexiones entre ellas además de los algoritmos a utilizar.

El programa deberá ser capaz de dar con la solución y encontrar el camino en un tiempo y memoria equilibrados además de funcionar para cualquier mapa del formato especificado en la competición [9].

4. Requisitos

El sistema tiene una serie de requisitos que deben cumplirse.

4.1.Requisitos funcionales

RF – 01: ejecución con mapa y búsqueda			
Descripción	El programa podrá ejecutarse con un mapa y dos puntos del mismo sobre los que buscar un camino.		
Prioridad	Alta	**Necesidad**	Alta

Tabla 1: RF - 01

RF – 02: modificar mapa			
Descripción	El usuario podrá cambiar el mapa de búsqueda		
Prioridad	Alta	**Necesidad**	Alta

Tabla 2: RF-02

RF – 03: modificar puntos			
Descripción	El usuario podrá cambiar los puntos de búsqueda		
Prioridad	Alta	**Necesidad**	Alta

Tabla 3: RF-03

RF – 04: completitud			
Descripción	Si existe un camino entre los dos puntos elegidos, el programa lo encontrará.		
Prioridad	Alta	**Necesidad**	Alta

Tabla 4: RF-04

RF – 05: búsqueda en puntos válidos			
Descripción	Si alguno de los puntos no es válido, el programa mostrará un error		
Prioridad	Media	**Necesidad**	Alta

Tabla 5: RF-05

RF – 06: punto final alcanzable			
Descripción	Si el punto final no es accesible desde el punto inicial se indicará un error.		
Prioridad	Media	**Necesidad**	Alta

Tabla 6: RF-06

RF – 07: uso de mapas de la competición			
Descripción	El programa funcionará con los mapas de la competición.		
Prioridad	Media	**Necesidad**	Baja

Tabla 7: RF-07

RF – 08: clasificación en regiones			
Descripción	El programa clasificará el mapa en regiones.		
Prioridad	Alta	Necesidad	Alta

Tabla 8: RF - 07

RF – 09: modificar número de waypoints			
Descripción	El usuario podrá elegir el número de waypoints a generar para realizar la búsqueda.		
Prioridad	Alta	Necesidad	Alta

Tabla 9: RF - 09

RF – 10: generación de waypoints			
Descripción	Los waypoints se generarán según el número elegido.		
Prioridad	Alta	Necesidad	Alta

Tabla 10: RF - 10

4.2.Requisitos no funcionales

RNF – 01: uso en cualquier sistema			
Descripción	El programa funcionará en cualquier sistema operativo.		
Prioridad	Alta	Necesidad	Alta

Tabla 11: RNF - 01

RNF – 02: tamaño del mapa			
Descripción	El programa funcionará para cualquier tamaño de mapa bidimensional		
Prioridad	Media	Necesidad	Alta

Tabla 12: RNF - 02

RNF – 03: validez de los argumentos			
Descripción	El sistema se ejecutará si los argumentos son válidos.		
Prioridad	Media	Necesidad	Alta

Tabla 13: RNF – 03

RNF – 04: mensajes de error			
Descripción	El sistema informará del tipo de error producido.		
Prioridad	Media	Necesidad	Media

Tabla 14: RNF - 03

5. Diseño del sistema

Como se ha comentado el objetivo del trabajo es el de realizar búsquedas informadas en mapas cuadriculados, por eso ha sido necesario establecer en el inicio la información que va a ser necesario almacenar para esas búsquedas. Antes de ejecutar la búsqueda se realizan las funcionalidades necesarias para leer, almacenar y organizar la información del mapa.

Toda esta información previa permite agilizar el tiempo de búsqueda. Hay que pensar que durante la ejecución de un videojuego será mucho más ágil encontrar el camino que un personaje quiere realizar si previamente está almacenado un posible camino para llegar al objetivo o tener todos los costes precalculados antes de ejecutar la búsqueda.

En esta sección se va a definir en la primera parte las clases creadas y la jerarquía entre ellas. Después se numerarán sus atributos y métodos.

5.1. Definición de clases

Se han definido una serie de clases de objetos que nos han permitido almacenar la información necesaria y establecer relaciones entre ellas para conseguir la funcionalidad deseada. El uso de instancias de estas clases por otras es importante ya que un objeto aporta más información y funcionalidad que un tipo básico.

Se han definido unas clases para realizar acciones básicas y otras que ya trabajan directamente sobre el mapa con su información y ejecutan la búsqueda.

La clase *Main* es la que se encarga de ejecutar el programa al inicio. Se tiene la clase *Reader* que permite leer el mapa y almacenarlo inicialmente sin analizar toda la información necesaria. Otra clase *Writer* escribe otro mapa en un documento de texto con las regiones halladas tras analizar el mapa.

Para almacenar la información se define la clase *GameMap* que contiene tipos básicos e instancias de *RegionMap* clase que representa cada una de las regiones del mapa, instancias de *Waypoint* que son las conexiones entre regiones y *Distance* que es la distancia entre dos regiones a través de sus waypoints y se utiliza como coste para el algoritmo de Dijsktra.

La búsqueda se ejecuta en la clase *Search* y se almacena en una instancia de tipo *Result* que contiene la información final de la búsqueda. Se ha definido una clase *Dijkstra* que hace la funcionalidad del algoritmo de Dijkstra. Una clase *Astar* que realiza la función del algoritmo de A* y requiere de objetos de las clases *Node* que representa los nodos de búsqueda, *Successor* que va generando los sucesores del nodo a analizar.

Las relaciones entre objetos que se han establecido pueden verse en la ilustración 1.

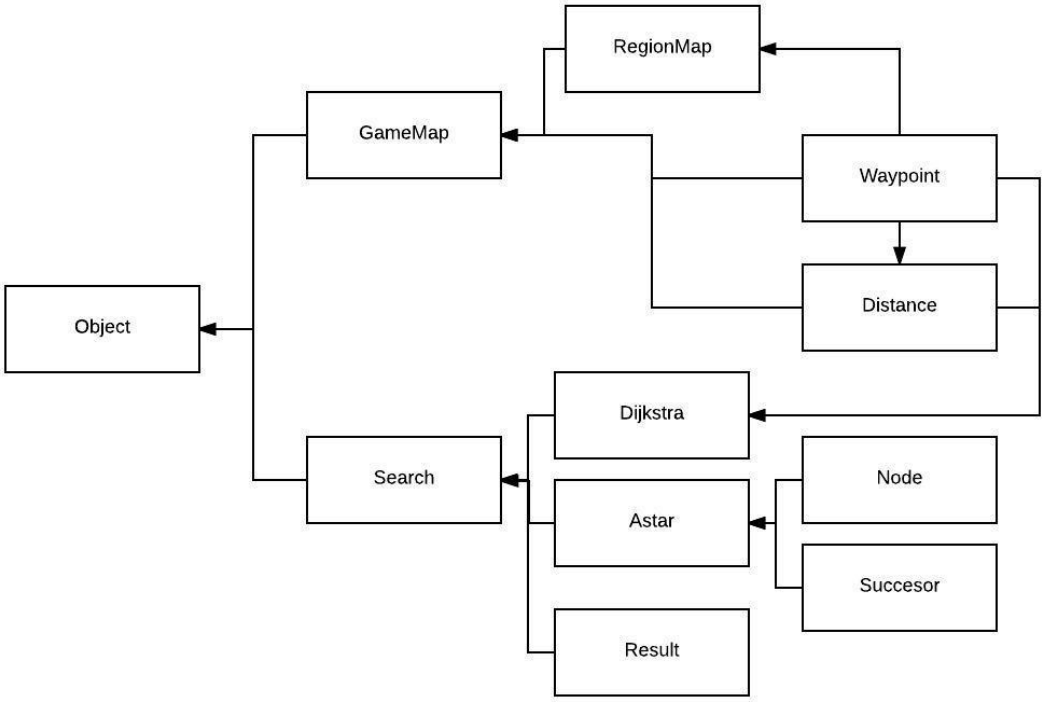


Ilustración 1: jerarquía de objetos

5.2. Arquitectura del sistema

Van a definirse con mayor detalle las clases y sus atributos y métodos.

5.2.1. Main

Esta clase es la primera en iniciarse y se encarga de ir realizando en orden las funcionalidades necesarias. Contiene los siguientes atributos:

Nombre	Tipo	Descripción
reader	Reader	Lee el mapa para su uso en el programa.
writer	Writer	Escribe las regiones del mapa en un documento de texto.
map	GameMap	Tiene toda la información y métodos relativos al mapa necesaria para hacer las búsquedas.
search	Search	Realiza la búsqueda entre los puntos elegidos.

Tabla 15: atributos de Main

Esta clase no cuenta con ningún método propio salvo el método *main* en el que se van ejecutando todos los pasos. El mapa y los puntos se reciben por argumentos.

Al iniciarse se crean las instancias de estos atributos y la primera acción es la de leer el mapa y después pasarlo al objeto *map* para generar las regiones, los waypoints, las distancias y toda la información que es necesaria para hacer la búsqueda y que se calcula previamente para agilizar el cálculo en la ejecución de la búsqueda del camino. Finalmente se ejecuta la búsqueda entre los puntos y se muestra la solución.

5.2.2. Reader

Esta clase recibe el archivo en el que está contenido el mapa inicial en el formato de la competición. El mapa se lee y se almacena la información de cada punto en forma de matriz del mismo tamaño del mapa. Contiene los siguientes atributos:

Atributo	Tipo	Descripción
x	Int	Ancho del mapa.
y	Int	Alto del mapa.
map	Byte [][]	Matriz que contendrá el mapa una vez leído.
file	FileReader	Clase de Java que nos permite cargar la información de un documento.
Br	BufferedReader	Clase de Java que nos permite ir leyendo y tratando la información cargada del documento.

Tabla 16: atributos de Reader

La clase cuenta además con los siguientes métodos:

Método	Parámetros	Descripción
Reader	map GameMap	Recibe un objeto de tipo mapa y lee su contenido almacenándolo en la matriz del mapa. Los puntos se codificarán como '1' si es un paso accesible o '0' si es un obstáculo.

Tabla 17: métodos de Reader

5.2.3. Writer

Esta clase escribe en un documento de texto el mapa con las regiones generadas una vez leído el mapa.

Contiene los siguientes atributos:

Atributo	Tipo	Descripción
fw	FileWriter	Objeto de java que permite crear un fichero.
pw	PrintWriter	Objeto de Java que permite escribir sobre un fichero.

Tabla 18: atributos de Writer

Y los siguientes métodos:

Método	Parámetros	Descripción
printZoneMap	Map GameMap	Recibe un objeto mapa e imprime en un documento de texto el mapa con las regiones generadas.

Tabla 19: métodos de Writer

5.2.4. GameMap

La clase que contiene toda la información con respecto al mapa es *GameMap* nombrada así para evitar problemas con la clase de Java *Map*. Esta clase contiene unos atributos de tipo básico e instancias de region, conexiones y distancias que aportan la información para hacer las búsquedas.

Atributo	Tipo	Descripción
map	Byte [][]	Almacena un array con el mapa básico, los puntos accesibles se codifican con 1 y los obstáculos con 0.
zone	Int [][]	Almacena en un array el mismo mapa codificando cada punto con la región a la que pertenece. Los obstáculos se representan con 0 y las regiones con un entero mayor que 0
width	Int	Almacena el ancho del mapa.
heigh	Int	Almacena el alto del mapa.
regions	List<RegionMap>	Lista de las regiones que forman el mapa.
connections	List<Waypoint>	Lista de los waypoints del mapa.
distances	List<Distance>	Lista con las distancias entre waypoints.

Tabla 20: atributos de GameMap

Sus métodos son:

Método	Parámetros	Descripción
getMap	Ninguno	Devuelve la matriz de byte que contiene el mapa.
getZone	Ninguno	Devuelve la matriz de enteros que contiene las regiones.
getWidth	Ninguno	Devuelve un entero con el ancho del mapa.
getHeigh	Ninguno	Devuelve un entero con el alto del mapa.
getRegions	Ninguno	Devuelve la lista de regiones.
getConnections	Ninguno	Devuelve la lista de waypoints.
getDistances	Ninguno	Devuelve la lista de distancias entre waypoints.
GamepMap	width Int, heigh Int, map Byte[][]	Constructor de la clase, guarda el mapa básico y genera las regiones, waypoints y distancias.
generateRegions	Ninguno	Lee el mapa básico y realiza un proceso de inundación localizando muros y obstáculos para localizar las regiones. Las regiones se van almacenando en la lista *regions*.

generateGateways	Ninguno	Una vez están localizadas las regiones se localizan las conexiones entre ellas y se crean instancias de *waypoint*. Se almacenan en la lista *connections*.
calculateDistances	Ninguno	Calcula las distancias entre los waypoints de dos regiones que están conectadas a través de otra intermedia. Se almacenan en la lista *distances*.
euclideanDistance	start Point, goal Point	Calcula la distancia euclidiana entre dos puntos y la devuelve.
region	point Point	Devuelve la región a la que pertenece el punto pasado por parámetro.

Tabla 21: métodos de GameMap

5.2.5. RegionMap

Representa cada región del mapa y contiene la información de sus conexiones. Conocer las regiones a las que se conecta cada región forma el grafo de nodos de regiones para el algoritmo de Dijkstra.

Sus atributos son:

Atributo	Tipo	Descripción
region	Int	Identificador numérico de la región.
connections	List<Waypoint>	Lista de waypoints de la región.

Tabla 22: atributos de RegionMap

Y sus métodos:

Método	Parámetros	Descripción
RegionMap	region Int	Crea una instancia de tipo región con su identificador y declara la lista de waypoints inicialmente vacía.
getRegion	Ninguno	Devuelve el número de la región.
getConnections	Ninguno	Devuelve la lista de waypoints de la región.

Tabla 23: métodos de RegionMap

5.2.6. Waypoint

Representa un punto que conecta dos regiones. Son los nodos para el algoritmo de Dijkstra.

Atributo	Tipo	Descripción
first	RegionMap	Identifica la región origen.
second	RegionMap	Identifica la región a la que pasa la conexión.
point	Point	Punto contenido en la primera que es colindante a un punto de la segunda región.

Tabla 24: atributos de Waypoint

Y sus métodos:

Método	Parámetros	Descripción
Waypoint	first RegionMap, second RegionMap, point Point	Constructor de un waypoint.
getFirst	Ninguno	Devuelve la primera región del waypoint.
getSecond	Ninguno	Devuelve la segunda región.
getPoint	Ninguno	Devuelve el punto de la conexión.

Tabla 25: métodos de Waypoint

5.2.7. Distance

Representa la distancia entre los waypoints de dos regiones. Son los grafos del algoritmo de Dijkstra.

Sus atributos son:

Atributo	Tipo	Descripción
first	Waypoint	Waypoint de la primera región.
second	Waypoint	Waypoint de la región destino, conectadas mediante una región intermedia.
distance	Double	Distancia euclidiana entre los dos waypoints.

Tabla 26: atributos de Distance

Y sus métodos:

Método	Parámetros	Descripción
Distance	first Waypoint, second Waypoint, distance Double	Constructor de un objeto Distance.
getFirst	Ninguno	Devuelve el primer waypoint.
setFirst	first Waypoint	Modifica el primer waypoint.
getSecond	Ninguno	Devuelve el segundo waypoint.
getDistance	Ninguno	Devuelve la distancia.

Tabla 27: métodos de Distance

5.2.8. Search

Controla la búsqueda ejecutando cada parte de la búsqueda. Primero se ejecutará el algoritmo de Dijkstra y una vez encontrados los waypoints por los que debe pasar el camino se buscará el camino entre cada par de waypoints con el algoritmo A*.

Atributo	Tipo	Descripción
dijkstra	Dijsktra	Instancia de la clase que ejecuta el algoritmo de Dijkstra.
astar	Astar	Instancia de la clase que ejecuta el algoritmo A*.
map	GameMap	Instancia de mapa, que contiene toda la información para hacer la búsqueda.

Tabla 28: atributos de Search

Y sus métodos:

Método	Parámetros	Descripción
Search	map GameMap	Constructor que recibe el mapa e instancia las búsquedas.
execute	start Point, goal Point	Recibe los dos puntos sobre los que se realizará la búsqueda. Si los dos puntos están en la misma región sólo se ejecuta A*. Si son de distinta región se añaden los puntos a los nodos del grafo de Dijkstra y se hace la búsqueda del camino más corto en el espacio de waypoints. Finalmente cuando tenemos el camino entre waypoints buscamos el camino entre los puntos del mapa con A* conectando cada par de waypoints.

Tabla 29: métodos de Search

5.2.9. Result

Almacena el resultado y la información de la búsqueda.

Atributo	Tipo	Descripción
time	Long	Tiempo que ha tardado en ejecutarse la búsqueda.
cost	Double	Coste de la búsqueda.
expandedNodes	Int	Número de nodos expandidos.
generatedNodes	Int	Número de nodos generados.

Tabla 30: atributos de Result

Y sus métodos:

Método	Parámetros	Descripción
Result	path List<Point>, generated Int, expanded Int, time long	Constructor del resultado de la búsqueda.
getTime	Ninguno	Devuelve el tiempo de búsqueda,
getCost	Ninguno	Devuelve el coste de la búsqueda.
getExpadedNodes	Ninguno	Devuelve el número de nodos expandidos.
getGeneratedNodes	Ninguno	Devuelve el número de nodos generados.
addResult	aux Result	Añade otro resultado, este método sirve para encadenar las búsquedas entre los waypoints.
print	filName String	Imprime el resultado por pantalla y en un documento.
printPathInMap	fileName String, map GameMap	Imprime el camino de la búsqueda en un mapa.

Tabla 31: métodos de Result

5.2.10. Dijsktra

Realiza la función de búsqueda del algoritmo de Dijsktra.

Sus atributos son:

Atributo	Tipo	Descripción
waypoints	List<Waypoint>	Lista de los waypoints del mapa que serán los nodos del grafo.
distances	List<Distance>	Lista de las distancias entre waypoints que serán las aristas del grafo.
settledNodes	Set<Waypoint>	Lista de nodos ya evaluados.
unsettledNodes	Set<Waypoint>	Lista de nodos expandidos pero no evaluados.
predecesors	Map<Waypoint, Waypoint>	Conexiones entre los nodos, indican la relación nodo padre-hijo.
distance	Map<Waypoint, Integer>	Tabla con los costes de los caminos.

Tabla 32: atributos de Dijkstra

Y sus métodos:

Método	Parámetros	Descripción
Dijkstra	map GameMap	Constructor del objeto, recibe un mapa que contiene la lista waypoints y de distancias.
execute	Waypoint source	Nodo de origen desde el que trazar las distancias.
findMinimalDistances	Waypoint node	Comprueba el camino más corto entre los nodos vecinos del actual.
getDistance	Waypoint node, Waypoint target	Devuelve la distancia entre los dos nodos indicados.
getNeighbors	Waypoint node	Devuelve los nodos vecinos del indicado.
getMinimum	Set<Waypoint> waypoints	Devuelve el nodo cercano con menor coste.
isSettled	Waypoint node	Comprueba si el nodo indicado ya ha sido visitado. Si se encuentra en la lista settledNodes devuelve true.
getShortestDistance	Waypoint destination	Devuelve el menor coste al nodo indicado.
getPath	Waypoint target	Devuelve el camino más corto desde el nodo inicial con que se ha calculado el grafo al destino.

Tabla 33: métodos de Dijkstra

5.2.11. Astar

Realiza la función de búsqueda del algoritmo de A*.

Sus atributos son:

Atributo	Tipo	Descripción
cost	Double	Coste del movimiento, en este caso se asignará coste 1.
map	GameMap	Objeto mapa con la información de puntos accesibles o con obstáculos.

Tabla 34: atributos de Astar

Y sus métodos:

Método	Parámetros	Descripción
Astar	GameMap map	Constructor del objeto, recibe el mapa para poder evaluar los puntos y se asigna al coste valor 1.
successors	Point actualState	Devuelve los nodos expandidos del punto actual, calcula todos los posibles puntos siguientes sin obstáculos en todas las direcciones del movimiento.
heuristic	Point state, Point goalState	Calcula la heurística del punto expandido, se ha elegido la distancia euclidiana desde el punto actual hasta el punto final elegido.
search	Point start, point goal	Realiza la búsqueda analizando los nodos que se van añadiendo a la lista abierta y pasándolos a

		cerrada cuando se han evaluado. En cada interacción selecciona el primer nodo de la lista de prioridad y genera los sucesores, si alcanza la solución devuelve un objeto Result.
generatePath	Node goalState	Cuando la búsqueda ha alcanzado el nodo final (la solución) se genera una lista con los nodos por los que pasa el camino hasta la solución.

Tabla 35: métodos de Astar

5.2.12. Node

Un nodo del algoritmo de A*.

Atributo	Tipo	Descripción
position	Point	Punto en el mapa.
from	Node	Es el nodo padre en el árbol de búsqueda.
h	Double	Valor de la función heurística del nodo.
g	Double	Valor del coste del movimiento.
cost	Double	Valor de la función del coste de la búsqueda hasta el nodo.

Tabla 36: atributos de Node

Y sus métodos:

Método	Parámetros	Descripción
Node	Point position, Node from, double g, double h, double cost	Constructor de un objeto nodo.
getPosition	Ninguno.	Devuelve el punto del nodo.
getFromTile	Ninguno.	Devuelve el nodo anterior en el árbol de búsqueda, si es el nodo raíz será null.
getG	Ninguno.	Devuelve el valor de g.
getH	Ninguno.	Devuelve el valor de h.
getF	Ninguno.	Devuelve la suma de g y h.
getCost	Ninguno.	Devuelve el valor del coste de búsqueda.
compareTo	Node tile	Compara el valor de la función de ambos nodos.

Tabla 37: métodos de Node

5.2.13. Sucessor

Es un sucesor de un nodo que se ha expandido antes de añadirlo a la lista de nodos.

Sus atributos son:

Atributo	Tipo	Descripción
point	Point	Punto en el mapa.
cost	Double	Coste del movimiento.

Tabla 38: atributos de Sucessor

Y sus métodos:

Método	Parámetros	Descripción
Successor	Point point, double cost	Constructor del objeto.
getPoint	Ninguno.	Devuelve el punto del nodo.
getCost	Ninguno.	Devuelve el valor del coste.

Tabla 39: métodos de Sucessor

6. Diseño de la solución

En esta sección va a explicarse cómo funciona cada módulo del sistema y las partes en las que se divide la ejecución. Inicialmente se lee un fichero que contiene el mapa y se clasifican sus puntos en regiones. Se genera un número de waypoints determinado y se traza una conexión entre estos waypoints para realizar la búsqueda en el espacio de regiones. Una vez realizado este proceso que es previo a la búsqueda se dispone de toda la información necesaria para hacer la búsqueda informada. La búsqueda consistirá en una primera búsqueda con un algoritmo de Dijkstra entre las regiones del punto inicial y final y después una búsqueda con un algoritmo A* sobre los puntos del mapa para cada par de waypoints del camino entre regiones desde el punto inicial al punto final.

6.1.1. Lectura del mapa

El fichero que contiene el mapa se pasa como argumento al programa para su lectura. Las dos primeras líneas contienen el ancho y el alto del mapa que se guardan en dos variables de tipo entero y nos sirven para conocer el tamaño de la matriz que representa los puntos del mapa. El fichero está codificado con "." o "G" para representar los huecos libre y "@", "O" para puntos no accesibles, "T" para los árboles que se consideran obstáculos y "W" para el agua, también terreno que no puede alcanzarse. Para simplificar esta codificación los puntos accesibles se almacenan como "1" y los puntos no accesibles u obstáculos como "0".

La matriz del mapa, el ancho y el alto son atributos de la clase *Reader* y tras la lectura del mapa se toman estos valores de la instancia de *Reader* y se pasan por parámetro de un objeto *GameMap* para continuar con la generación de regiones y waypoints.

El siguiente código es de la clase *Reader* y corresponde con la lectura del fichero y completar la matriz de puntos accesibles del mapa. No devuelve ningún valor ya que se toman los valores de los parámetros.

```
//Instanciar matriz de mapa
map = [x][y];
line = leer línea del fichero;
while (line != null) {
      for (j = 0; j < y; j++) {
            for (i = 0; i < x; i++) {
                  if (line.letra(i) == '.' || line.letra(i) == 'G') {
                  map[i][j] = 1;
                  //Punto libre
                  } else
                        this.map[i][j] = 0;
                        //Obstáculo
                  }
            line = siguiente linea;
            }//Fin primer for
      }//Fin Segundo for
}//Fin while
```

Puede consultarse el código complete en el anexo 2.

6.1.2. Regiones

Cuando se lee el mapa este se procesa para generar regiones en función de los muros y obstáculos que existen. Tras leer el mapa se tiene una matriz codificada con 1 y 0 para cada punto del mapa, siendo 1 los puntos caminables y 0 los obstáculos. Esta matriz, el ancho y el alto del mapa son los parámetros que se pasan al constructor de la clase *GameMap* para instanciar un objeto de esta clase.

Esta clase tiene un atributo *zone* que es una matriz de enteros con las mismas dimensiones del mapa y cada punto de la matriz corresponde con un punto del mapa y tiene el número de región a la que pertenece. Todos los obstáculos se guardan con el valor "0" y los puntos accesibles con el valor "-1" mientras se desconozca la región a la que pertenecen.

Para generar las regiones se puede realizar un proceso de inundación que vaya buscando los muros y delimite las regiones. Para hacer esto se ha adaptado el algoritmo de inundación del trabajo de Yngvi Björnsson y Kári Halldórsson [10] en el que también se busca una solución mediante la clasificación previa de regiones. Como la matriz de puntos y la de zonas se atributos de la clase el algoritmo no necesita recibir ningún parámetro ni devuelve ningún dato.

Para conocer la región de cada punto la función busca el primer punto libre que no pertenece a ninguna región y avanza hacia la derecha del mapa hasta que da con un muro. En la ilustración 2 el algoritmo comienza en el primer punto más a la izquierda libre del mapa y avanza hacia la derecha en la misma fila.

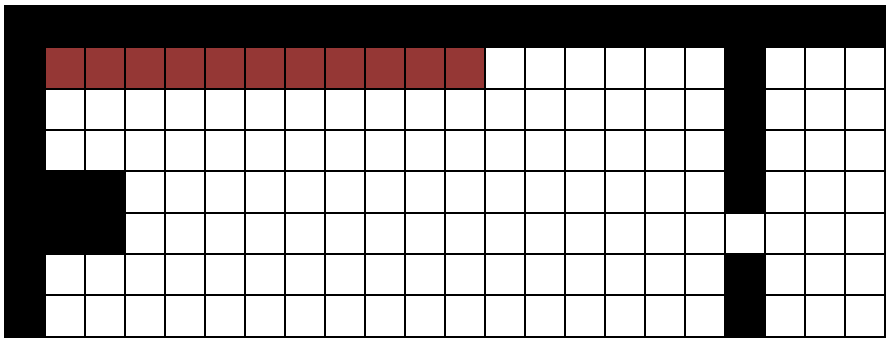


Ilustración 2: en la primera interacción se mueve hacia la derecha

En ese punto se vuelve al primer punto libre a la izquierda una línea más abajo y se vuelve a avanzar hacia la derecha. En la ilustración 3 se ha llegado a un punto que contiene un muro, se baja a la siguiente fila y continúa avanzando hacia la derecha.

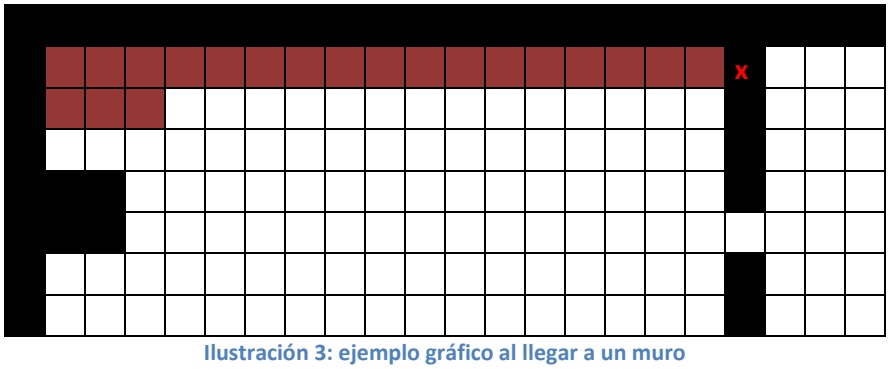


Ilustración 3: ejemplo gráfico al llegar a un muro

Si al volver a la izquierda la casilla no está libre se avanza una casilla para comprobar si es un obstáculo o un muro. Si la siguiente casilla tampoco está libre se considera un muro y se genera una región con todos los puntos antes recorridos.

Si se llega a una apertura, como en los puntos anteriores hay un muro se considera otra región y se vuelve atrás una línea más abajo como se ve en la ilustración 4, donde a la derecha hay un hueco que da a una zona amplia.

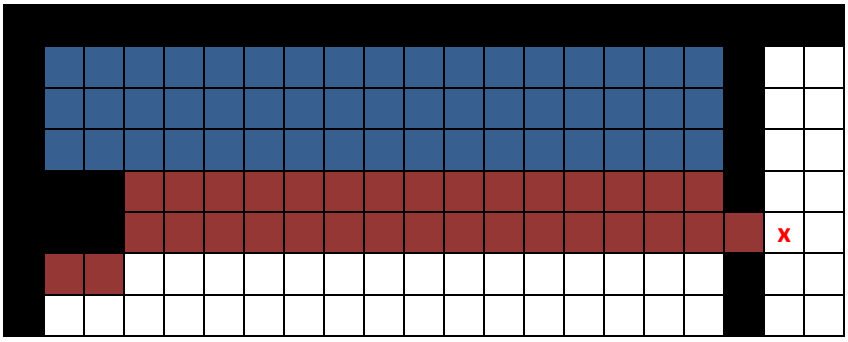


Ilustración 4: ejemplo gráfico al llegar a una abertura

Según se recorren los puntos en la matriz de zonas se asigna el valor correspondiente de la región. Este proceso se realiza hasta que se ha leído todo el mapa y no queda ninguna celda libre. Hasta que termina no se sabe cuántas regiones tiene un mapa, pues inicialmente el mapa no está clasificado en regiones.

De un mapa sencillo que inicialmente tuviera la forma de la ilustración 5.

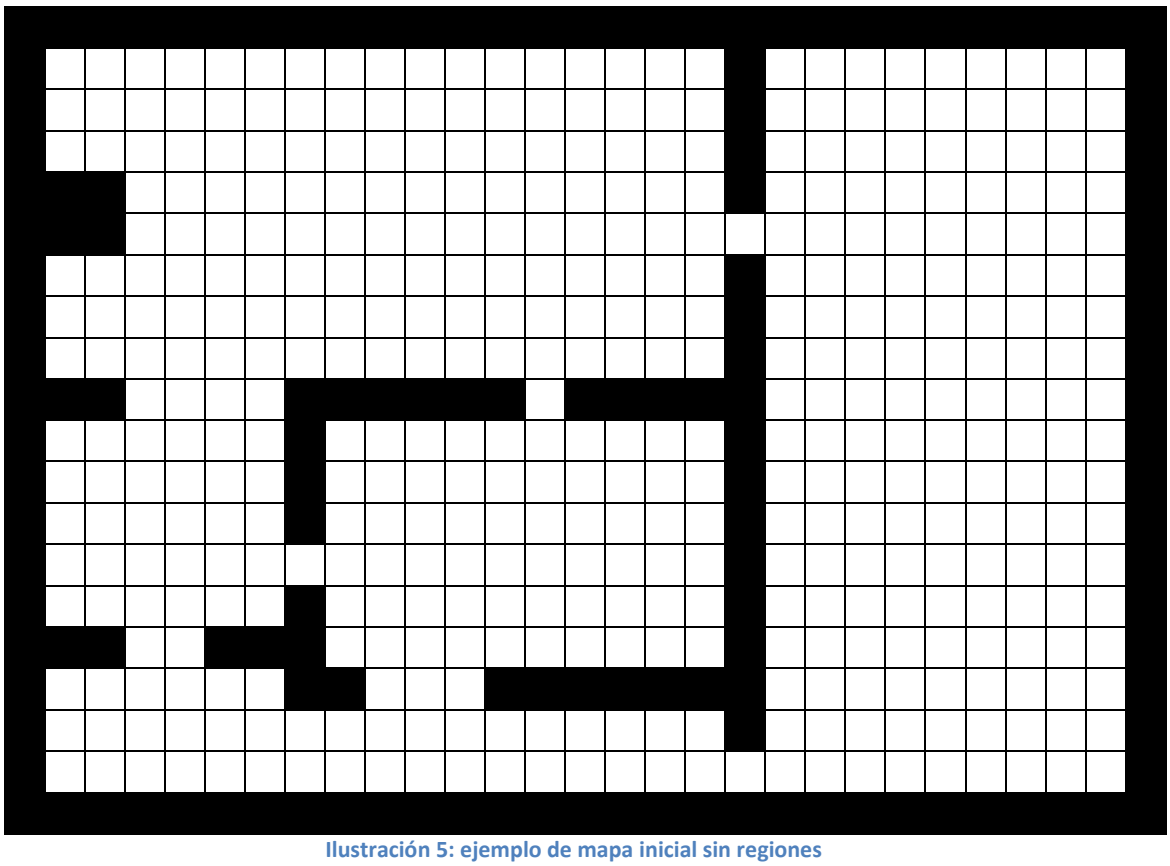


Ilustración 5: ejemplo de mapa inicial sin regiones

Se tendría la clasificación en regiones de la ilustración 6.

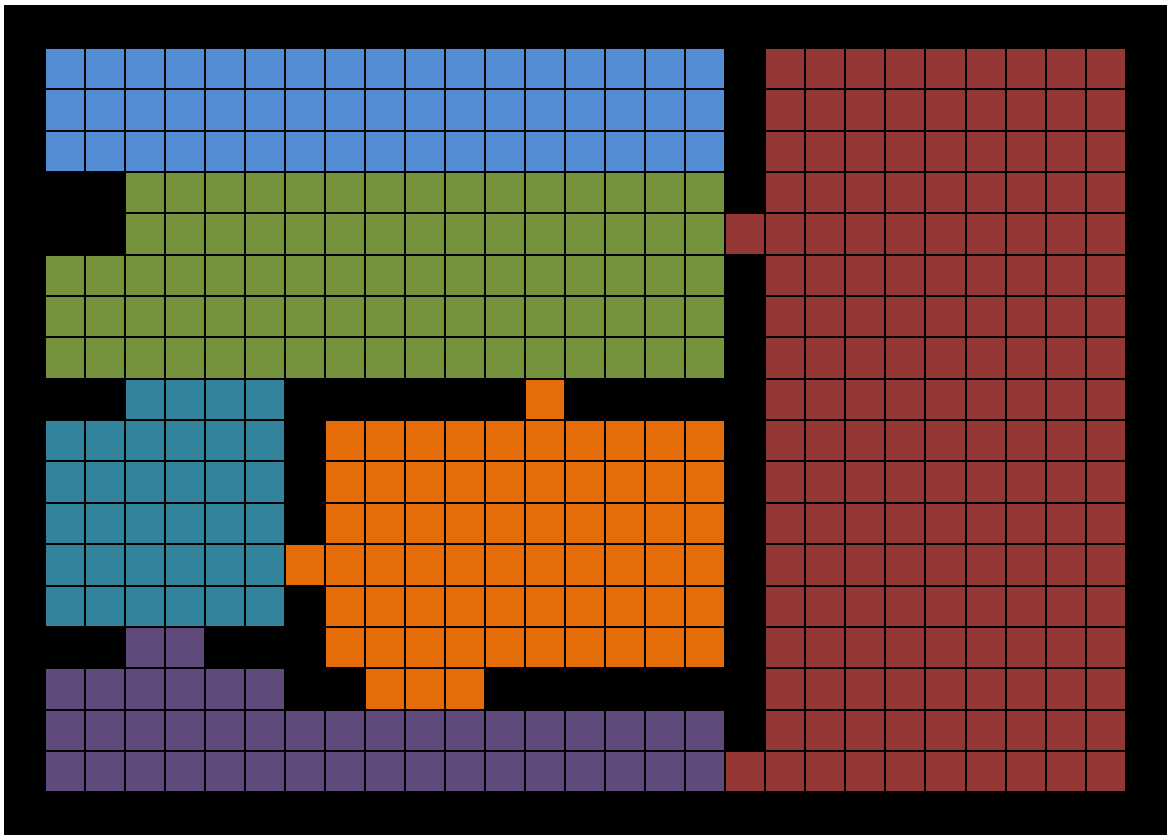


Ilustración 6: ejemplo de mapa clasificado en regiones

Cada vez que se termina de recorrer una región se crea una instancia de región que se añade a la lista de regiones del mapa. Las regiones contienen una lista con sus waypoints que inicialmente está vacía pues aún no se han generado estos. El algoritmo termina cuando se ha leído todo el mapa y no queda ninguna celda libre. El algoritmo de inundación para clasificar las regiones corresponde al siguiente pseudocódigo:

```
Punto(x,y) -> punto libre más a la izquierda
shrunkR, shrunkL = false;
actualRegion = 1;
do {
      x = xLeft;
      if (zone(x,y)==free)
            zone(x,y) = actualRegion
      while (zone(x,y) == free  && zone(x+1, y-1) !=free ) {
                  x = x + 1;
                  zone(z,y) = actualRegion
      }
      if (zone(x+1,y-1) == actualRegion)
            shrunkR = true;
      else if (zona(x,y - 1) != actualRegion && shrunkR) {
            while (zone(x,y) == actualRegion) {
                  zone(x,y) = free;
                  x = x - 1;
            }
            break;
      }//fin else if
      x = xLeft;
      y = y + 1;
```

```
      if (y >= heigh) //fuera del mapa
            break;
      while (zone(x,y) != free && zone(x, y -1) == actualRegion){
                  x = x + 1;
      }}//fin while
      while (zone(x,y) != free&& zone(x-1, y) == free) {
                  x = x - 1;
      }
      if (x >= width)
            break; //Fuera del mapa
      xLeft = x;
      if (zone(x-1, y-1) == actualRegion) {
            shrunkL = true;
      } else if (zone(x, y-1) != actualRegion && shrunkL) {
            break;
      }
      } while (true);
      actualRegion++;
} while (puntos libres);
```

Puede consultarse el código completo en el anexo 3.

Tras la generación de regiones se genera un documento de texto en la carpeta del programa llamado *RegionMap.txt* donde puede verse el mapa clasificado en regiones.

6.1.3. Waypoints

Los waypoints representan la conexión entre las regiones y su dirección va desde una región de origen a una región final que será vecina de esta. Mediante la función que va a explicarse en esta sección se recorre la matriz de regiones hasta encontrar puntos en los que hay dos regiones vecinas, es decir que sus celdas coincidan con dos valores distintos de región.

Para encontrar los waypoints existen dos casos:

- **Waypoints en sentido horizontal**: cuando una región queda encima de la otra región vecina y hay una abertura entre ellas. La matriz se recorre hacia la derecha hasta encontrar un muro o hasta que la región vecina de abajo cambia a otra región. El valor de la coordenada X será mayor y la Y será igual. En la región vecina Y tendrá una unidad más. El pseudocódigo que localiza un waypoint de este tipo es el siguiente:

```
if (zone(x+1, y) != actualRegion && zone(x+1,y) != obstáculo) {
   startRegion = actualRegion; //primera región
   endRegion = zone(x + 1,y); //segunda región
   firstX = x; //Primer punto del waypoint
   firstY = y;
   while (zone(x+1,y+1) == endRegion && zone(x,y+1 == startRegion){
         y = y +1;
   }
   lastX = x;//Punto del otro extremo
   lastY = y;}
}
```

- **Waypoints en sentido vertical:** cuando una región queda a la derecha de otra región. La matriz se recorre hacia abajo hasta encontrar un muro o una región distinta a la

vecina actual. El valor de Y aumenta y X es igual. En la región vecina X tiene una unidad más. El pseudocódigo que identifica este tipo de waypoints:

```
if (zone(x,y+1) != actualRegion && zone(x,y+1) != obstáculo) {
   startRegion = actualRegion; //primera región
   endRegion = zone(x,y+1); //segunda región
   firstX = x; //Primer punto del waypoint
   firstY = y;
   while (zone(x+1,y+1) == endRegion && zone(x+1,y == startRegion){
         y = y +1;
   }
   lastX = x; //Coordenada X del extremo
   lastY = y;}//Coordenada Y del primer extremo
}
```

Cada waypoint pertenece a una región y por cada waypoint de una región hay un waypoint de la región vecina.

El algoritmo para generar los waypoints no recibe ningún parámetro ya que la matriz de regiones es un atributo de la clase *GameMap* y los waypoints se generan en esta misma clase. Cada vez que se crea un waypoint se añade a la lista de waypoints del mapa y a la región a la que pertenecen. Cuando se ha recorrido todo el mapa finaliza el algoritmo y no es necesario devolver ningún valor.

Un ejemplo de qué punto sería un waypoint puede verse en la ilustración 7.

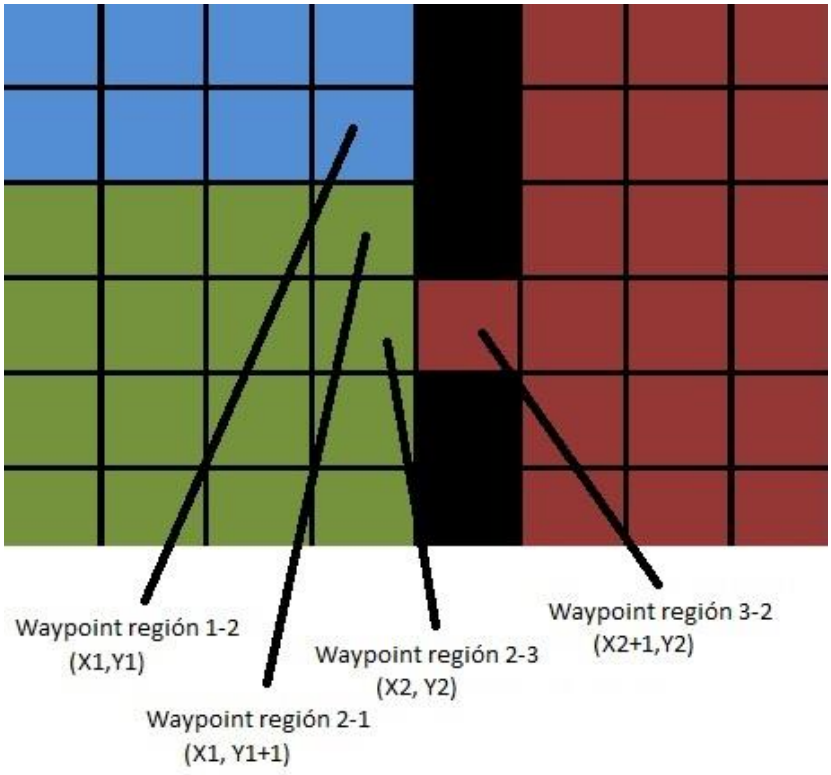


Ilustración 7: esquema de waypoints

A partir de esta definición se han diseñado distintas soluciones que corresponden con el número de waypoints entre cada región que dependerá de la anchura de la abertura. Si la abertura sólo ocupa 1 punto, solo habrá un waypoint por región en esa conexión.

6.1.3.1. Un waypoint en el punto central

El único waypoint que aparecerá entre cada región será el punto central de la abertura. Se muestra un ejemplo de los puntos seleccionados en la ilustración 8.

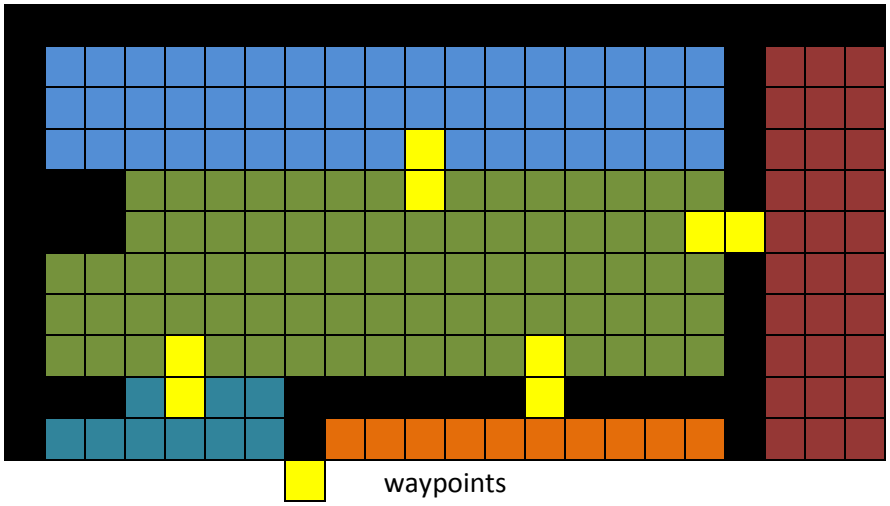


Ilustración 8: ejemplo de solución de 1 waypoint

El pseudocódigo que corresponde con esta solución:

```
// Para waypoint punto central
if(lastY != firstY){
      X = ((lastX + firstX) / 2);
      Y = ((lastY + firstY) / 2);

//Declarar waypoint en region actual
      aux = Waypoint(actualRegion,endRegion,Point(X,Y))
//Añadir conexión a la region y al mapa
      actualRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
//Crear conexión de la segunda región y añadir
      aux = Waypoint(endRegion, actualRegion, Point(X+1, Y));
      endRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
}else{ //Fin if
//Si la abertura es un solo punto
      //Crear waypoint y añadir
      aux = Waypoint(actualRegion, endRegion, Point(firstX, firstY));
      actualRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);

      //Crear waypoint de la segunda región y añadir
      aux = Waypoint(endRegion), actualRegion,Point(firstX+1,firstY));
      endRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
}//Fin else
```

6.1.3.2. Un waypoint en cada extremo

Hay dos waypoints por conexión, en los extremos. Se muestra un ejemplo en la ilustración 9.

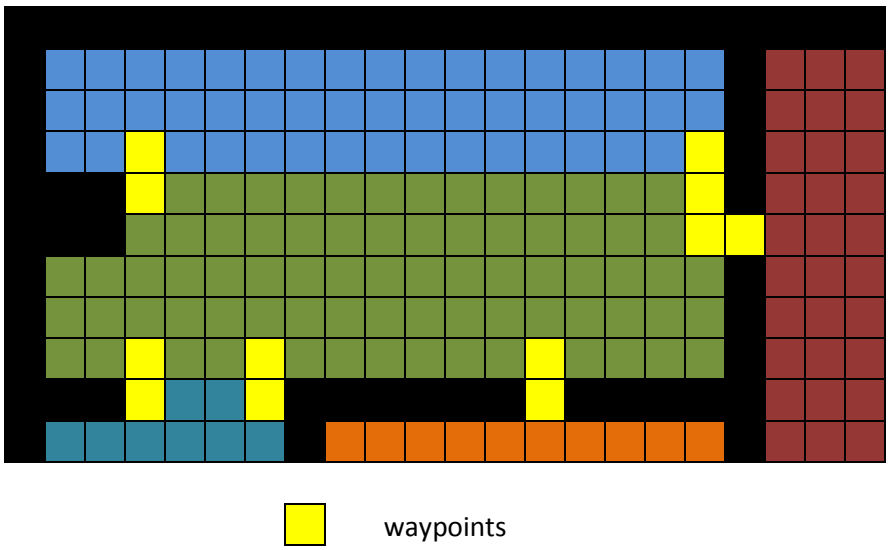


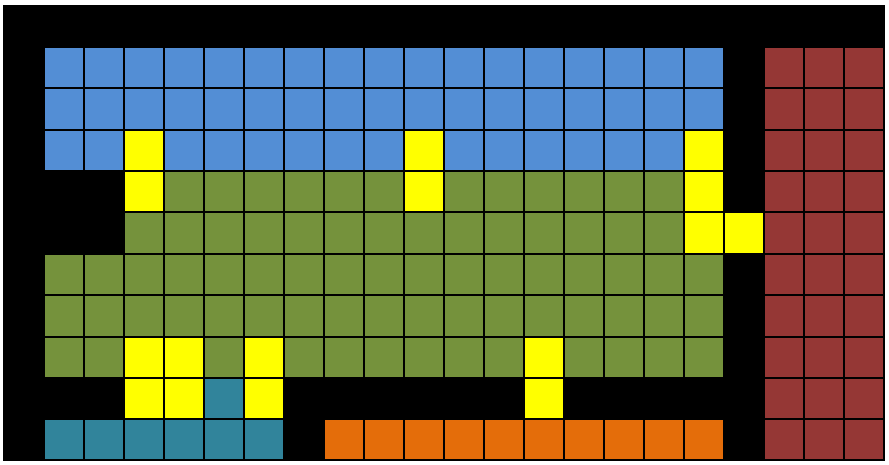
Ilustración 9: ejemplo de solución con waypoints en los extremos

Que corresponde con el pseudocódigo:

```
// Para dos waypoints
//Crear waypoint de la primera región y añadir
aux =  Waypoint(actualRegion, endRegion, Point(firstX, firstY));
actualRegion.añadirWaypoint(aux);
conexiones.añadir(aux);

//Crear waypoint de la segunda región y añadir
aux = Waypoint(endRegion, actualRegion),Point(firstX + 1, firstY));
endRegion.añadirWaypoint(aux);
conexiones.añadir(aux);

//Si la apertura es de más de un punto, guardar el otro extremo
if (lastY != firstY){
//Waypoint de la primera región
      aux = Waypoint(actualRegion, endRegion, Point(lastX, lastY));
      actualRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);

//Waypoint de la segunda región
      aux = Waypoint(endRegion, actualRegion, Point(lastX + 1,lastY));
      endRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
 }//Fin if
```

6.1.3.3. Un waypoint en cada extremo y el punto medio

Además de los extremos se añade la conexión del punto medio de la primera solución. Se corresponde con la unión de los espacios de waypoints de las dos soluciones anteriores. Se muestra un ejemplo en la ilustración 10.



waypoints

Ilustración 10: ejemplo de solución con waypoints en extremos más punto medio

El pseudocódigo de esta solución sería:

```
// Para tres waypoints
//Waypoint de la primera región
aux = Waypoint(actualRegion, endRegion, Point(firstX, firstY));
actualRegion.añadirWaypoint(aux);
conexiones.añadir(aux);

//Waypoint de la segunda región
aux = Waypoint(endRegio, actualRegion, Point(firstX + 1, firstY));
endRegion.añadirWaypoint(aux);
conexiones.añadir(aux);

//Si la longitud es mayor que 1, se guarda el otro extremo
if (lastX != firstX) {
      //Punto de la primera región
      aux = Waypoint(actualRegion,endRegion, Point(lastX, lastY))
      actualRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);

      //Punto de la segunda región
      aux = Waypoint(endRegion, actualRegion, Point(lastX + 1,lastY));
      endRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);

//Si existe punto medio
if (lastY > firstY + 2) {
      X = ((lastX + firstX) / 2);
      Y = ((lastY + firstY) / 2);
      //Punto de la primera región
      aux = Waypoint(actualRegion, endRegion, Point(X,Y));
      actualRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);

      //Punto de la segunda región
      aux = Waypoint(endRegion, actualRegion, Point(X + 1,Y));
      endRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
}//Fin if
```

6.1.3.4. Un waypoint por cada extremo y cada 5 puntos

Se crean los waypoints de los extremos y un waypoint cada 5 puntos entre los extremos. En la ilustración 11 se muestra un ejemplo de los waypoints con esta solución.

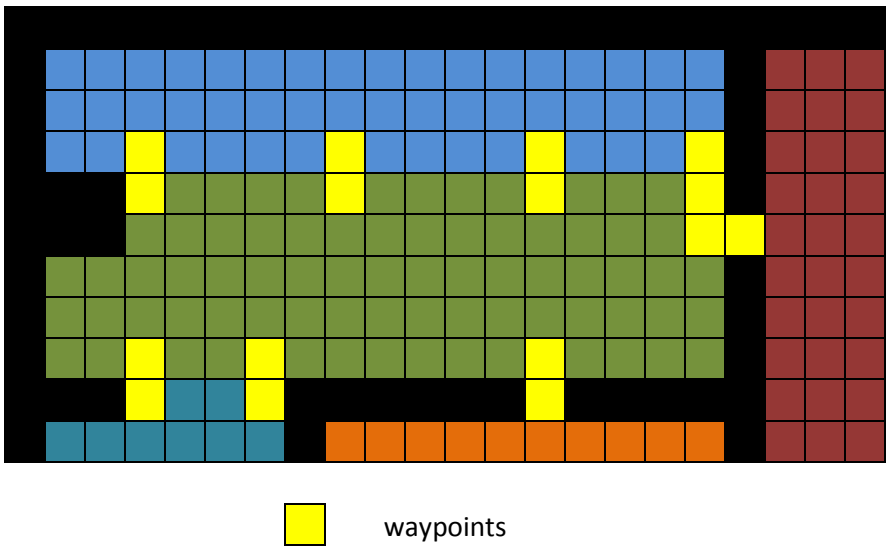


Ilustración 11: ejemplo de solución con un waypoint en los extremos y cada 5 puntos

Que corresponde con el pseudocódigo:

```
// cada 5 puntos
//Punto de la primera región
aux = Waypoint(actualRegion,endRegion,Point(firstX, firstY));
actualRegion.añadirWaypoint(aux);
conexiones.añadir(aux);

//Punto de la segunda región
aux = Waypoint(endRegion, actualRegion, Point(firstX + 1, firstY));
endRegion.añadirWaypoint(aux);
conexiones.añadir(aux);

//Si longitud mayor que 1, añadir el otro extremo
if (lastY != firstY) {
      //Punto de la primera región
      aux = Waypoint(actualRegion, endRegion, Point(lastX, lastY));
      actualRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
      //Punto de la segunda región
      aux = Waypoint(endRegion, actualRegion, Point(lastX + 1,lastY));
      endRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
}
//Añadir waypoints cada 5 puntos
if (lastY > firstY + 5) {
      j = firstY + 5;
      while (j < lastY) {
            Y = j;
            //Punto de la primera región
            aux = Waypoint(actualRegion,endRegion, Point(firstX, Y));
            actualRegion.añadirWaypoint(aux);
            conexiones.añadir(aux);
            //Punto de la segunda región
            aux = Waypoint(endRegion,actualRegion,Point(firstX + 1,Y));
```

```
            endRegion.añadirWaypoint(aux);
            conexiones.añadir(aux);
            j = j + 5;
      }
}// Fin if
```

6.1.3.5. Un waypoint por cada extremo y cada 3 puntos

Se crean los waypoints de los extremos y un waypoint cada 3 puntos entre los extremos. Igual que en el caso anterior en algún caso por la longitud de la abertura se creará el waypoint del punto medio. En la ilustración 12 se ve un ejemplo de la solución.

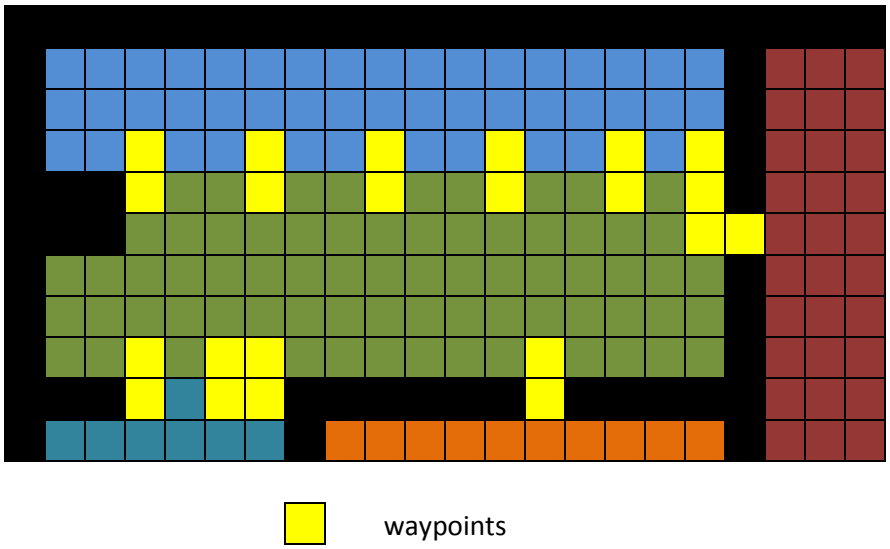


Ilustración 12: ejemplo de solución con un waypoint en los extremos y cada 3 puntos

Que corresponde con el pseudocódigo:

```
// cada 5 puntos
//Punto de la primera región
aux = Waypoint(actualRegion,endRegion,Point(firstX, firstY));
actualRegion.añadirWaypoint(aux);
conexiones.añadir(aux);

//Punto de la segunda región
aux = Waypoint(endRegion, actualRegion, Point(firstX + 1, firstY));
endRegion.añadirWaypoint(aux);
conexiones.añadir(aux);

//Si longitud mayor que 1, añadir el otro extremo
if (lastY != firstY) {
      //Punto de la primera región
      aux = Waypoint(actualRegion, endRegion, Point(lastX, lastY));
      actualRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
      //Punto de la segunda región
      aux = Waypoint(endRegion, actualRegion, Point(lastX + 1,lastY));
      endRegion.añadirWaypoint(aux);
      conexiones.añadir(aux);
}
//Añadir waypoints cada 3 puntos
if (lastY > firstY + 3) {
      j = firstY + 3;
      while (j < lastY) {
```

```
                Y = j;
                //Punto de la primera región
                aux = Waypoint(actualRegion,endRegion, Point(firstX, Y));
                actualRegion.añadirWaypoint(aux);
                conexiones.añadir(aux);
                //Punto de la segunda región
                aux = Waypoint(endRegion,actualRegion,Point(firstX + 1,Y));
                endRegion.añadirWaypoint(aux);
                conexiones.añadir(aux);
                j = j + 3;
        }
}// Fin if
```

En el algoritmo de la solución se generan los waypoints correspondientes según la solución elegida. El modo de búsqueda se recibe como parámetro mediante un número del 1 al 5 que corresponde con las soluciones antes definidas:

- 1: waypoint en el punto central.
- 2: waypoints en los extremos.
- 3: waypoint en el punto central y en los extremos
- 4: waypoint en los extremos y cada 5 puntos.
- 5: waypoint en los extremos y cada 3 puntos.

En el anexo 4 puede consultarse el código completo de la generación de waypoints.

6.1.4. Distancias

Para el algoritmo de Dijkstra se tendrá un grafo que representa las conexiones entre las regiones a través de sus waypoints. Estas distancias son la distancia euclidiana entre los waypoints de cada región con los waypoints de su región vecina, excepto los waypoints que conectan con la región de la que se parte.

Gráficamente, las distancias en un waypoint se trazarían como en la ilustración 13.

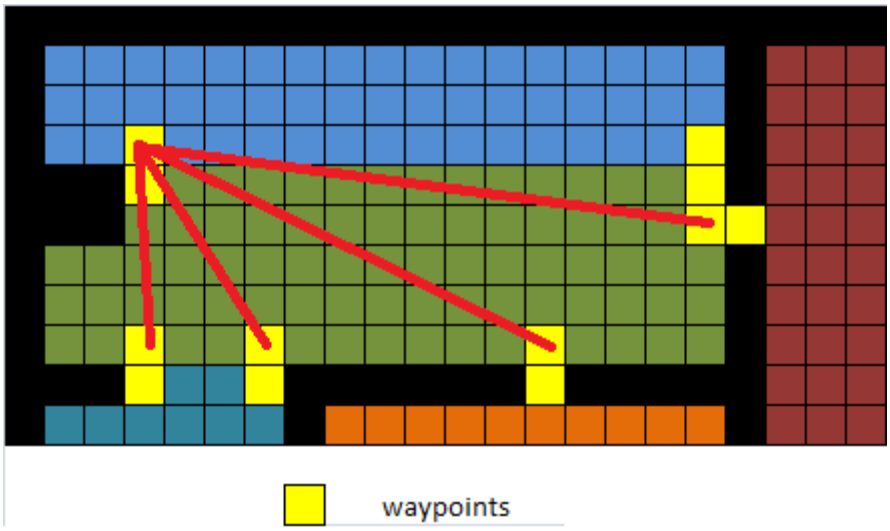


Ilustración 13: representación gráfica de las distancias

En la ilustración 13, el waypoint de la región 1 se conecta con los waypoints de la región 2 sin incluir el waypoint que conecta la región 2 con la 1. Existe una misma distancia en sentido inverso de las conexiones de la región 2 con las de la 1.

Si se trazan las conexiones del otro waypoint de la región 1 se forma un grafo de la como en la ilustración 14.

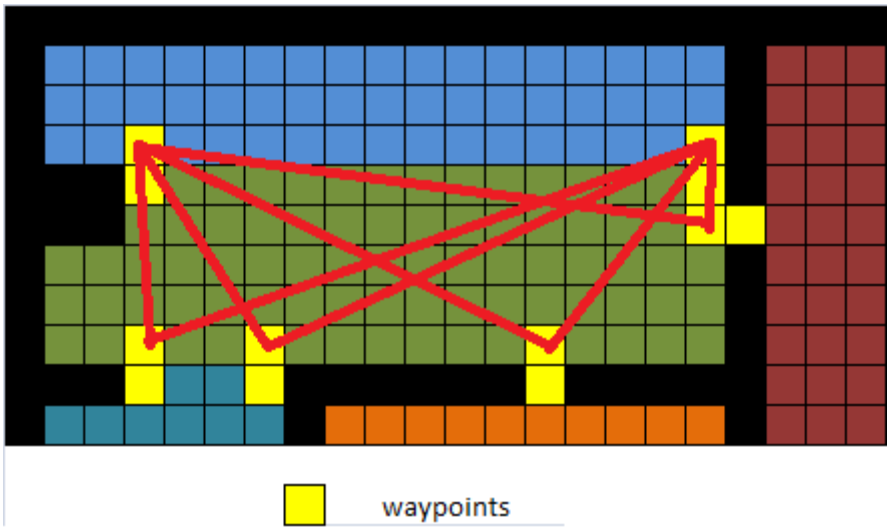


Ilustración 14: distancias del segundo waypoint de la región 1

Y también trazando las distancias de la región 5 con la región 2 quedando como en la ilustración 15. No se añaden el resto de conexiones con todas las regiones para mantener la claridad de la imagen.

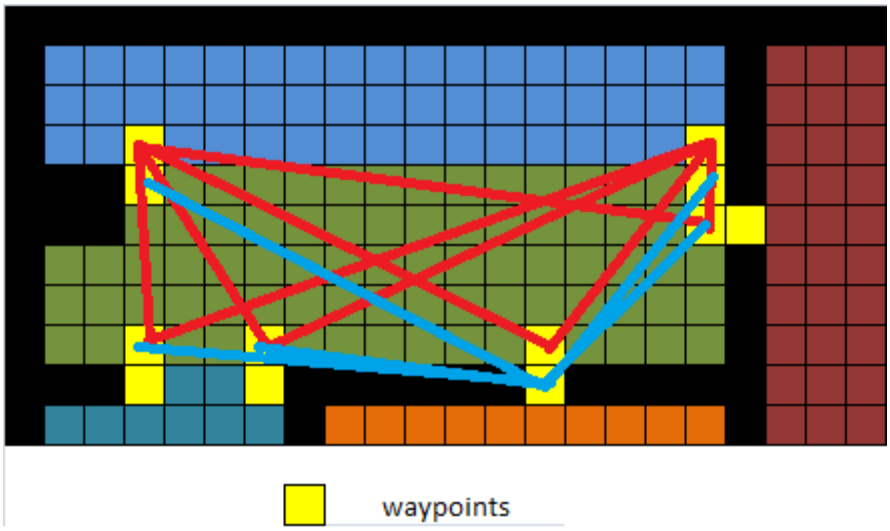


Ilustración 15: distancias de las regiones 1 y 5 con la región 2

El pseudocódigo para la generación de distancias es el siguiente:

```
i = 0;
while (i < número de regiones) {
      for (Waypoint auxF = region i waypoints) {
            for (Waypoint auxS = conexiones region vecina de i)) {
                  if (auxF.primeraRegion == auxS.segundaRegion)
                        continue;
                  distancias.añadir(Distance(auxF, auxS,
                        EuclideanDistance(auxF.point,auxF.point);
            }
      }
      i = i +1;}}
```

La función *euclideanDistance* calcula la distancia euclidiana entre los dos puntos pasados por parámetro. Puede verse el código completo en el anexo 5.

6.1.5. Búsqueda

Para realizar la búsqueda se hará uso de los dos algoritmos comentados en conjunto. Según los puntos escogidos pueden darse distintos casos que se resuelven de manera distinta.

6.1.5.1. Alguno de los puntos no es válido

Antes de comenzar la búsqueda se comprueba si alguno de los puntos corresponde a un obstáculo o muro. Si esto sucede, como no puede encontrarse un camino hasta un punto no accesible se genera un error y no se realiza la búsqueda.

6.1.5.2. Puntos en la misma región

Si los dos puntos se encuentran en la misma región no es necesario buscar en el espacio de regiones y directamente se llama al algoritmo de A* sobre estos dos puntos. Para conocer la región a la que pertenece un punto basta con consultar sus coordenadas en la matriz *zone* del mapa, incluida en la clase *GameMap*.

6.1.5.3. Puntos en regiones distintas

Si los dos puntos están en regiones diferentes será necesario hacer primero una búsqueda entre la región del punto inicial y la región del punto final. Esta búsqueda se realiza mediante el algoritmo de Dijkstra utilizando las conexiones entre waypoints antes definidas. El peso viene dado por la distancia euclidiana entre cada par de waypoints conectados mediante las distancias explicadas en la sección anterior.

Para poder realizar la búsqueda directamente entre el nodo inicial y final se añaden estos puntos al grafo de waypoints conectándolos con los waypoints de la región a la que pertenecen y calculando la distancia euclidiana a esos waypoints como se muestra en la ilustración 16. La dirección del punto final va desde las conexiones de su región a ese nodo.

El algoritmo de Dijkstra recibe un objeto mapa y el grafo de waypoints está reprsentado mediante la lista de distancias. El algoritmo hace un recorrido del grafo desde el nodo inicial y añade las distancias mínimas al resto de nodos. Este es un paso previo a la búsqueda.

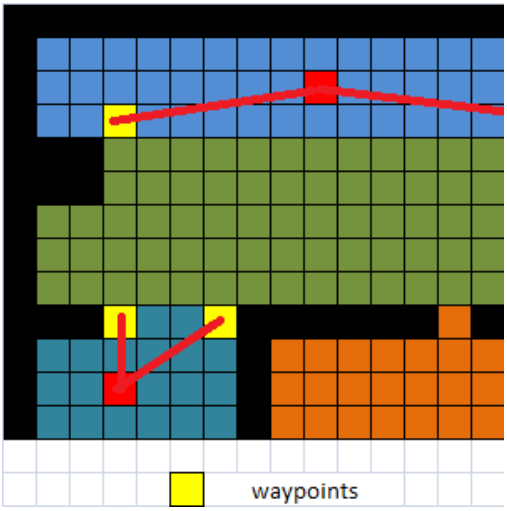


Ilustración 16: paso inicial de búsqueda con Dijkstra

El pseudocódigo del algoritmo de Dijkstra sería el siguiente:

```
//recorrido de distancias
Dijkstra (Waypoint origen) {

settledNodes = List <Waypoint>; //lista de nodos añadidos
unsettledNodes = List <Waypoint>();//lista de nodos a comprobar
distance = List <Waypoint, Integer>();//tabla de distancias
previousNodes = List <Waypoint, Waypoint>(); //Lista de nodos previos
distance.put(source, 0);
unsettledNodes.añadir(origen);
while (unsettledNodes no vacía) {
      Waypoint node = mínimo (unsettledNodes);
      settledNodes.añadir(node);
      unsettledNodes.quitar(node);
      findMinimalDistances(node);
}
}

//elegir el nodo con distancia más corta
findMinimalDistances(Waypoint node) {

List<Waypoint> adjacentNodes = vecinos(node);
for (Waypoint target : adjacentNodes) {
      //Si la distancia es menor desde target
      if (distancia(node) > distancia(node)
            + distancia(node, target)) {
                  distance.añadir(target,distancia(node)
                              + distancia(node, target));
                  //Cambiar nodo previo
                  previousNodes.añadir(target, node);
                  //Añadir nuevo nodo para evaluarlo
                  unsettledNodes.añadir(target);
            }
      }
}
```

Cuando se completa el recorrido del grafo otra función recibe el nodo final y devuelve una lista con los waypoints por los que debe pasar el camino entre regiones desde el nodo

inicial. Este proceso forma parte del proceso de búsqueda El código de la función corresponde al siguiente pseudocódigo:

```
List<Waypoint> getPath(Waypoint meta) {

//Lista de nodos del camino
LinkedList<Waypoint> path = List<Waypoint>();
Waypoint paso = meta;
// comprobar si hay un camino entre los dos nodos
if (previousNodes.get(paso) == null) {
      return null;
}
path.añadir(paso);
while (hay siguiente paso) {
      paso = nodo anterior;
      path.añadir(paso);
}
      return path;
}
```

En el anexo 6 puede consultarse el código completo del algoritmo.

Una vez realizado este paso se tiene la lista de nodos que indican los waypoints por los que debe pasar la búsqueda pero esto no representa el camino final.

Para encontrar el camino a través de los puntos del mapa se utiliza el algoritmo de A* para conectar cada par de waypoints vecinos hasta alcanzar el nodo meta. El algoritmo recibe en cada ejecución un punto inicial y final y busca entre los puntos del mapa los nodos que se expanden hasta alcanzar la solución. En este caso se trata de los puntos caminables entre cada waypoint o entre el nodo inicial y final. El algoritmo que realiza la búsqueda de A* es el siguiente:

```
Result search(Point start, Point end) {
expandedNodes = 0;
generatedNodes = 1;
Queue <Node> openList; // Lista abierta
List<Node> closeList ;// Lista cerrada
List <Point, Double> costList; //Costes
//Añadir estado inicial
openList.añadir(Node(start, 0, heuristica(start, end)));
costList.put(start, 0.0);
while (openList no vacía) {
      Node actualState = openList.quitarPrimero;
      //Si se alcanza la solución
      if (actualState == end) {
            path = obtenerRuta(actualState);
            return Result((path, generatedNodes,expandedNodes, coste);
      }
      closeList.add(actualState);
      expandedNodes++;
      //Generar sucesores
      List successors = obtenerSucesores(actualState)
      for (Successor successor: successores) {
      //Añadirlo a la lista si no está repetido comprobando su G
      if (!closeList.contiene(successor)){
            newG = actualState.G + successor.cost;
            if (!costList.contiene(successor.point) ||
            costList.get(successor.point > newG) {
```

```
                        costList.put(successor.point, newG);
                        for (Iterator it=openList.iterator; it.Next;){
                              if(it.next.Position == successor.Point)
                                    //Quitar si la G es mayor
                                    it.quitar;}
                              node = Node (successor.point,actualState,newg,
                                    heuristic(successor.point, end),
                                    successor.cost);
                              openList.añadir(node);
                              generatedNodes++;
                              }
                        }
                  }
            }
//Si no se ha encontrado solución
return new Result(null, generatedNodes, expandedNodes, 0);
}
```

Puede verse el código completo con la función de sucesores y heurística en el anexo 7.

6.1.5.4. Puntos no alcanzables

Si los puntos son válidos se realiza el proceso de búsqueda con los pasos antes definidos pero puede darse el caso de que el punto final no sea alcanzable desde el punto inicial elegido. En este caso se devuelve una lista de nodos vacía y se genera un mensaje de error.

6.1.6. Ejemplo de ejecución

Para la ejecución de una búsqueda el programa recibirá por argumentos la ruta del fichero en formato *nombre.map*, las coordenadas x e y del mapa separadas por espacios de la forma: x1 y1 x2 y2, y el tipo de búsqueda mediante un número del 1 al 5 para indicar el número de waypoints deseados como se explica en la sección de creación de waypoints.

El primer paso es la lectura del mapa, para lo que se crea una instancia *Reader* que recibe la ruta del fichero. Abre el fichero si lo encuentra y lee cada línea codificando los caracteres con 1 y 0. Se rellena la matriz de tamaño del mapa.

Cuando se ha leído el mapa se toman los atributos de *Reader* de ancho, alto y la matriz del mapa para instanciar un objeto *GameMap*. En este objeto se ejecutan la clasificación de regiones, la generación de waypoints y de distancias. Cuando se terminan los tres procesos el objeto *GameMap* tiene los atributos de la lista de regiones, waypoints y distancias.

Para la búsqueda se instancia un objeto *Search* que recibe el objeto *GameMap* y los dos puntos de búsqueda, que se añaden a la lista de waypoints y de distancia. Se instancia un objto *Dijkstra* que recibe el mapa para acceder a los atributos de la lista de waypoints y de distancias. Antes de la búsqueda se recorre el grafo de waypoints para obtener las distancias mínimas a los nodos. Después comienza el proceso de búsqueda, se devuelve la ruta al nodo final mediante una lista de waypoints. Para encontrar el camino a la meta, se instancia un objeto *Astar* que recibe cada par de nodos desde el punto inicial al final y devuelve un objeto con el resultado que incluye la lista de puntos y su coste. En las siguientes imágenes se puede comprobar cómo ha sido el resultado de estas búsquedas con el mapa del ejemplo.

En la ilustración 17 se muestra la búsqueda entre los puntos azul y verde usando un waypoint en el punto central.

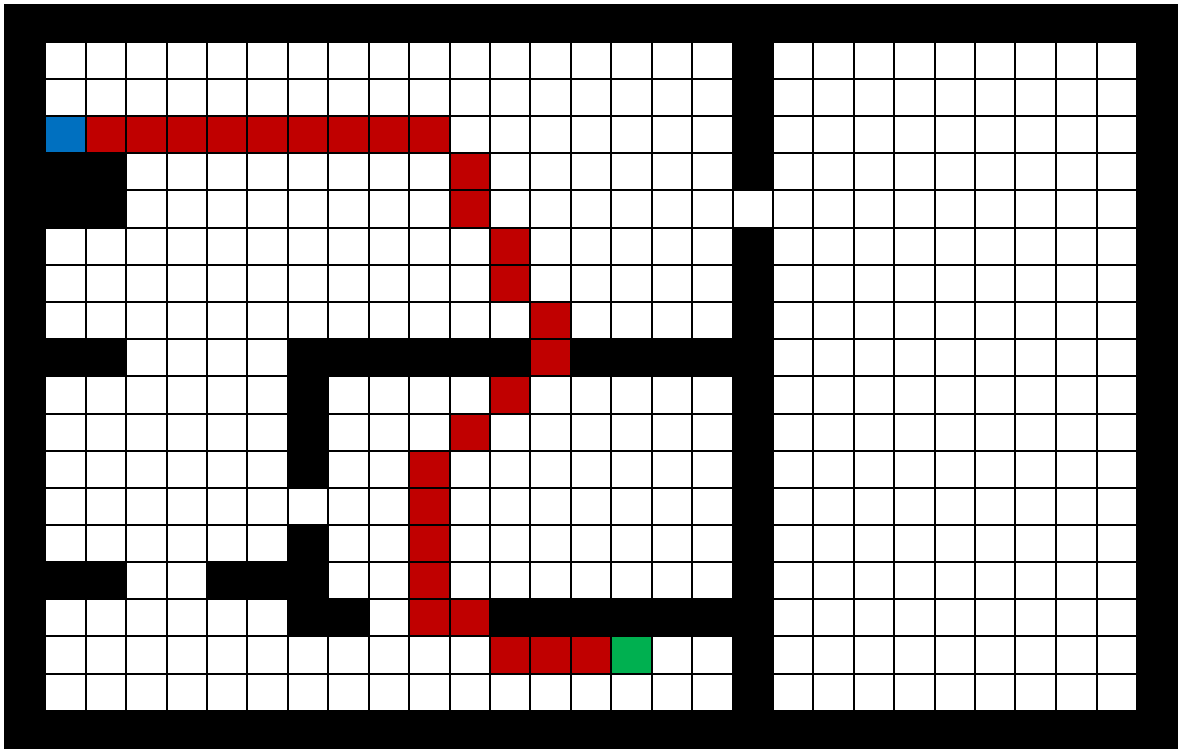


Ilustración 17: ejemplo con 1 waypoint

En la ilustración 18 se muestra un ejemplo de cómo ha sido la búsqueda en los mismos puntos del ejemplo anterior, pero con los waypoints de los extremos.

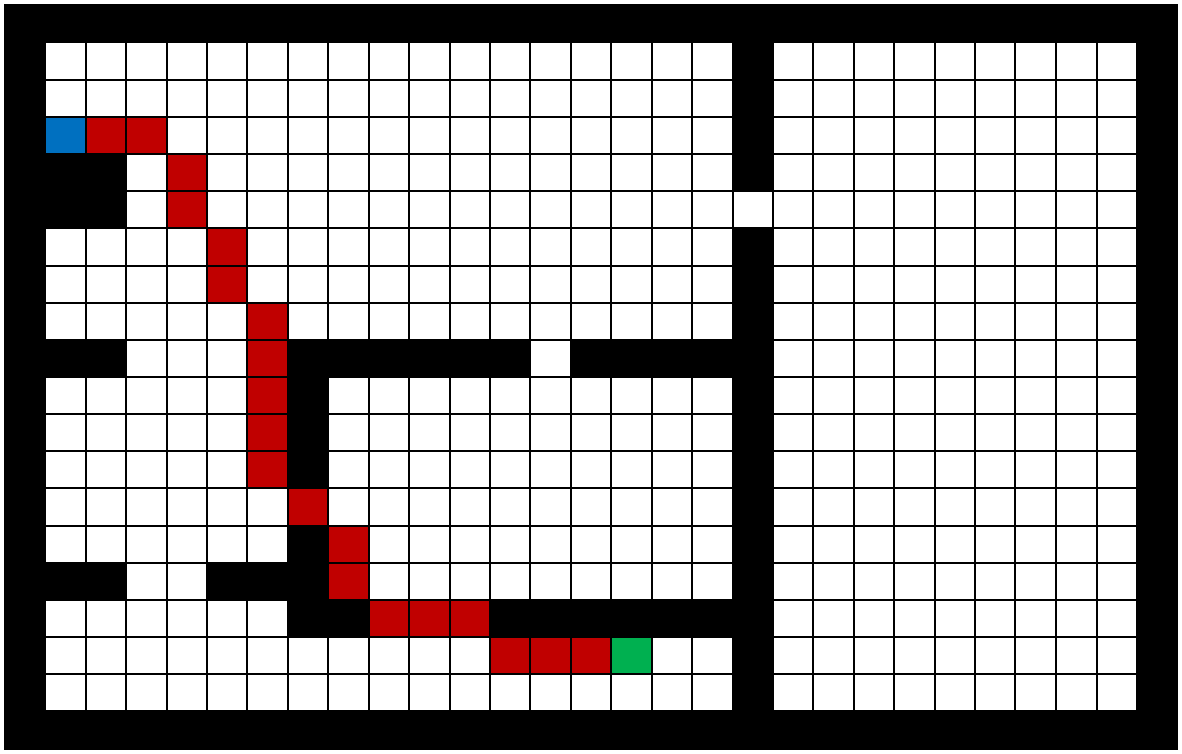


Ilustración 18: ejemplo de búsqueda con los waypoints de los extremos

En la ilustración 19 puede verse un ejemplo de búsqueda con los waypoints de los extremos y en el punto central. Se comprueba que varía con respecto al camino anterior.

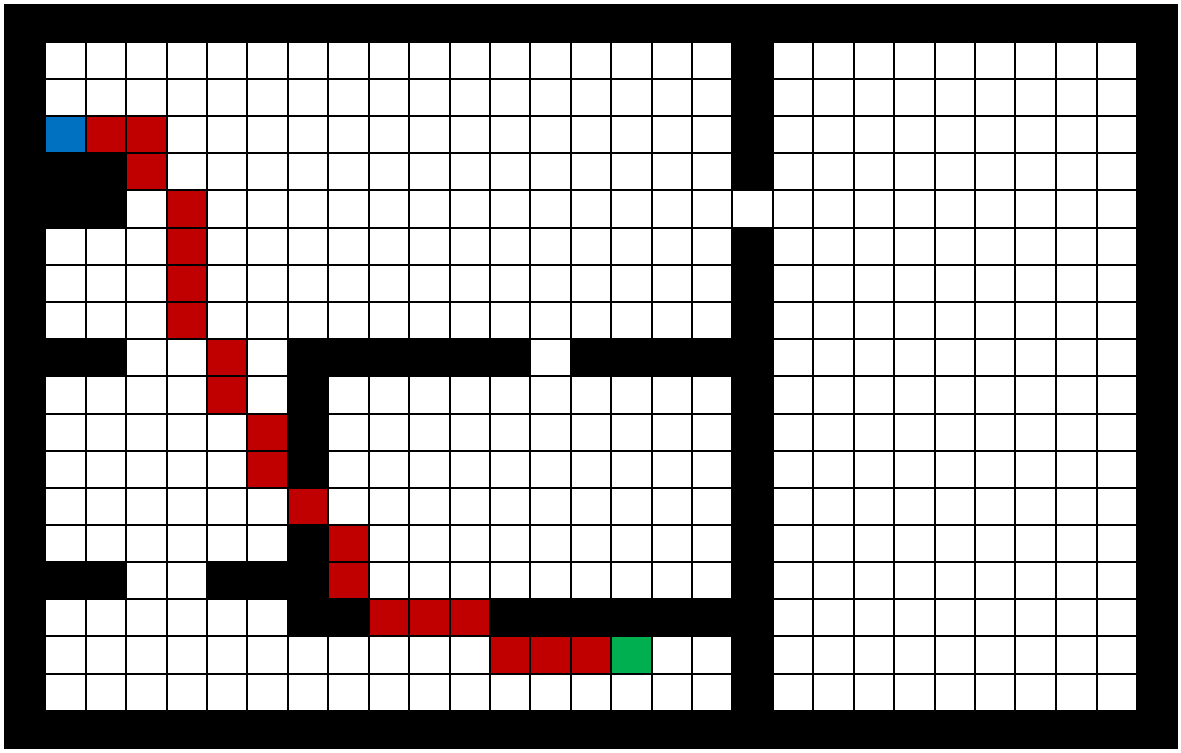


Ilustración 19: ejemplo de búsqueda con waypoints de los extremos y el punto central

En la ilustración 20 se observa el resultado de una búsqueda con los waypoints de los extremos y cada 5 puntos. Es el mismo camino de la ilustración 18 con los waypoints de los extremos, debido a que el mapa es sencillo y pocas entradas tienen un ancho mayor que 5.

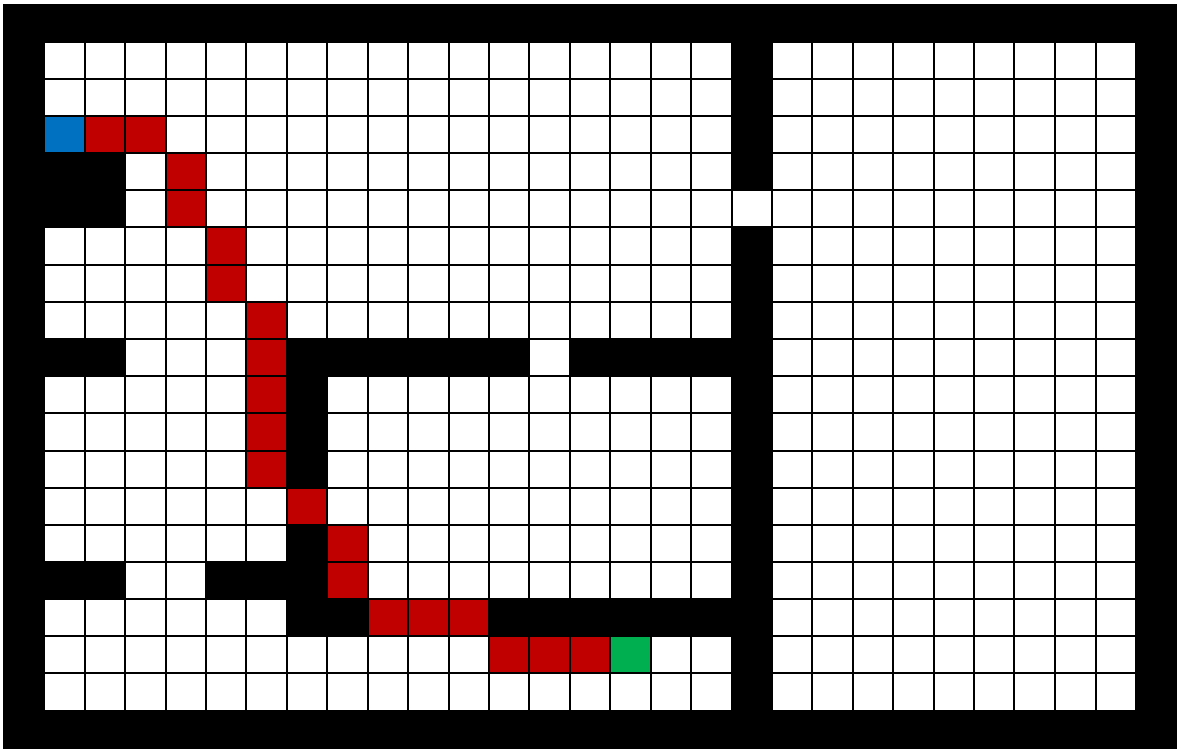


Ilustración 20: ejemplo de búsqueda con los waypoints de los extremos y cada 5 puntos

En la ilustración 21 se muestra el resultado de una búsqueda con los waypoints de los extremos y cada 3 puntos. Igual que en caso anterior, este mapa es demasiado sencillo para ver la diferencia de la búsqueda con waypoints cada 5 y 3 puntos.

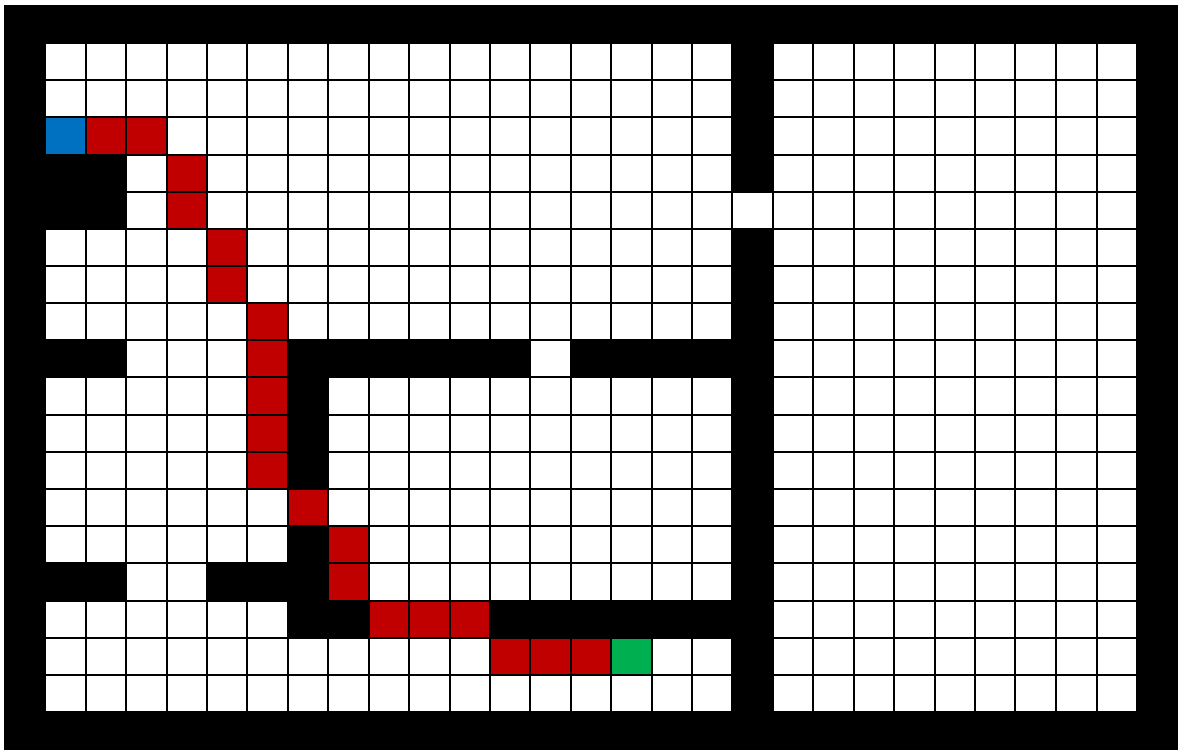


Ilustración 21: ejemplo de búsqueda con los waypoints de los extremos y cada 3 puntos

En la ilustración 22 se muestra la búsqueda con el algoritmo de A*, que es el camino óptimo.

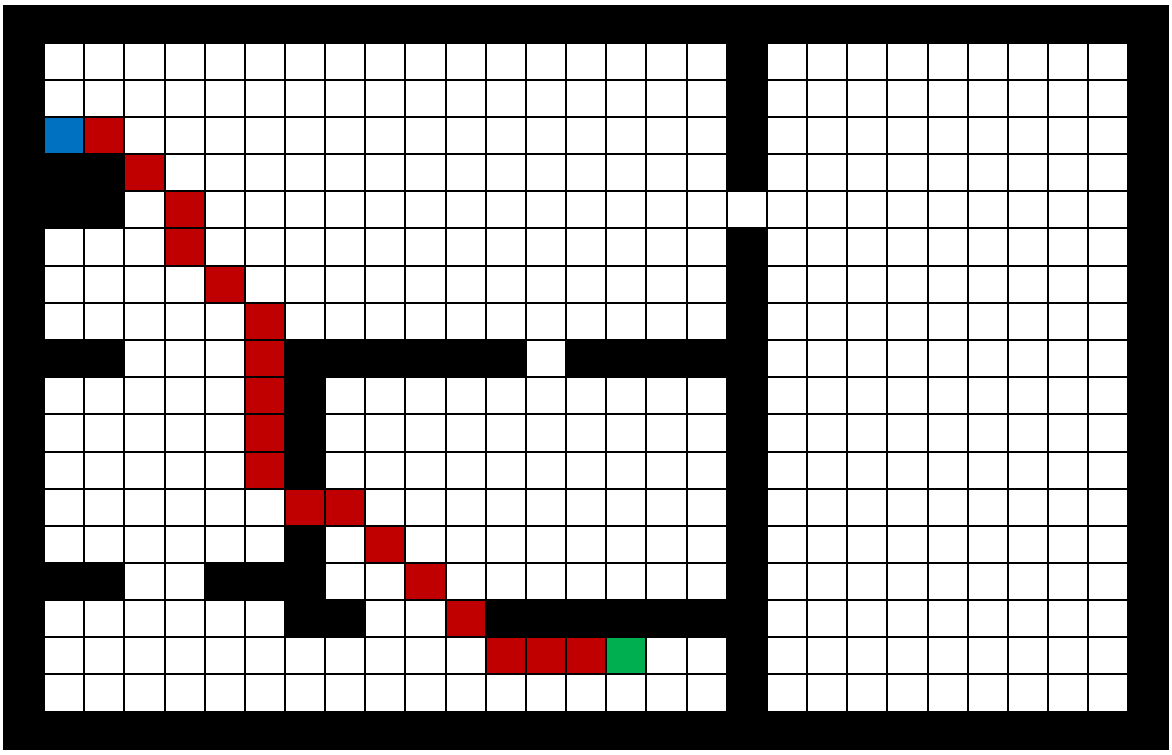


Ilustración 22: ejemplo de búsqueda con A*

7. Resultados

Para realizar las pruebas se han escogido 3 mapas: uno con formas cuadradas, otro irregular y otro de tamaño considerable para ver las variaciones en el tiempo de búsqueda.

Las pruebas de búsqueda se han realizado con el distinto número de waypoints definido en el apartado de la solución. Para comparar la funcionalidad de la solución se compara con el coste de un algoritmo de A* en diferentes longitudes.

7.1.Mapas

Los mapas para las pruebas se han escogido acorde a sus características para comprobar el funcionamiento de la generación de regiones y waypoints para después hacer pruebas con los distintos números de waypoints en mapas con distintas formas y números de regiones y waypoints distintos.

El primer mapa a probar consiste en una zona de habitaciones cuadradas y grandes aperturas. Los muros forman zonas laberínticas. El segundo pertenece al juego Dragon Age Origins y se ha escogido por tener formas más irregulares. El último mapa es del juego Baldurs Gate y tiene zonas estrechas que forman pasillos con grandes salas más amplias.

Estos mapas pueden consultarse en el anexo 9.

En esta sección se recogen datos sobre el procesado de los mapas. El número de regiones y waypoints generados y el tiempo que ha tardado en hacerse este proceso.

El número de regiones por cada mapa se recoge en la tabla 40.

	Mapa		
	Mazmorra (1)	Dragon Age Origins (2)	Baldurs Gate (3)
Regiones	76	274	79

Tabla 40: número de regiones por mapa

El tiempo medido en milisegundos que ha tardado en procesarse y el número de waypoints generados según la búsqueda se recoge en la tabla 41.

	Mazmorra (1)		Dragon Age Origins (2)		Baldurs Gate (3)	
	Tiempo	Waypoints	Tiempo	Waypoints	Tiempo	Waypoints
1 waypoint	335	144	218	1020	262	222
2 waypoints	283	288	217	1712	285	380
3 waypoints	339	392	229	1840	270	418
Cada 5 puntos	290	1088	219	2406	298	916
Cada 3 puntos	326	1632	223	3046	281	1316

Tabla 41: tiempo de procesado del mapa y número de waypoints

El aumento del número de waypoints puede verse mejor en la ilustración 23.

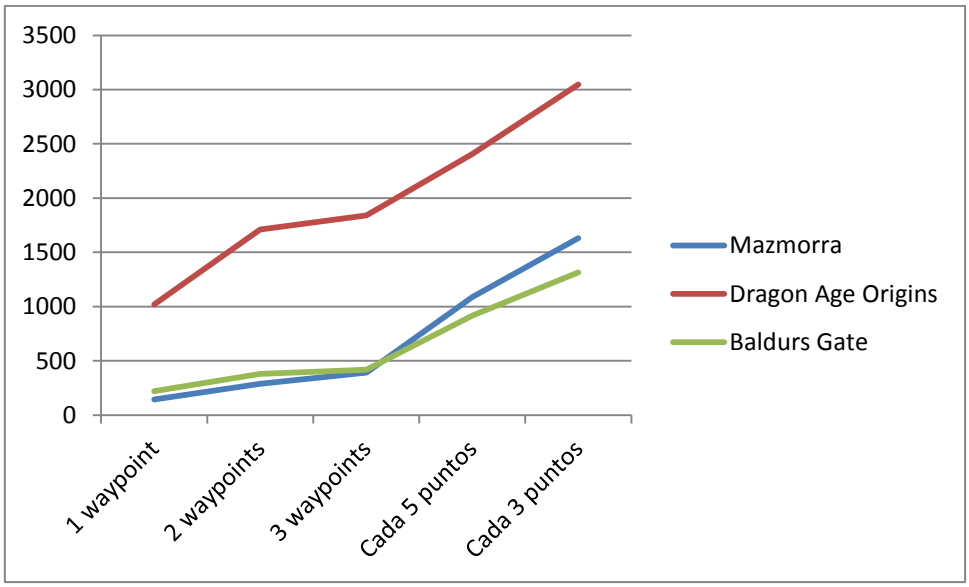


Ilustración 23: aumento del número de waypoints según el tipo de búsqueda

Se puede decir que el aumento de waypoints es lineal y que todos los mapas siguen una distribución semejante cuando se aumentan los waypoints de cada uno.

7.2.Pruebas

En esta sección se recogen los datos del coste del camino y el tiempo en milisegundos que ha tardado en encontrarse la solución según el tipo de búsqueda en cada mapa. Se han escogido en cada mapa 3 caminos de longitud cada vez mayor para comparar los tiempos de búsqueda.

Para las pruebas del diseño que se ha definido va a contarse el tiempo total en milisegundos pero también se va a diferenciar qué tiempo corresponde a la búsqueda con Dijkstra y con A*. El coste de las soluciones se verá en una tabla resumen comparándolo con el coste obtenido con un algoritmo de A* directamente sobre el mapa.

Las primeras tablas que se muestran a continuación corresponden con los tiempos de búsqueda para los diferentes números de waypoints. A continuación se comparan los costes de estas soluciones y se analiza cómo ha sido ese tiempo de búsqueda.

Al final de esta sección se hará una comparación de los datos recogidos para definir las conclusiones sobre la solución.

7.2.1. Con un waypoint en el punto central

		Coste	Tiempo Dijkstra	Tiempo A*	Tiempo total
Corto	1	372	7	44	51
	2	251	56	9	65
	3	423	21	24	45
Medio	1	1475	7	40	47
	2	652	64	23	87
	3	874	25	22	47
Largo	1	6033	7	40	47
	2	1192	58	29	87
	3	2893	22	40	62

Tabla 42: resultados con 1 waypoint

En la tabla 42 se recogen los tiempos en milisegundos obtenidos por la búsqueda con 1 waypoint. El tiempo total y lo que ha tardado en hacerse la búsqueda con Dijkstra y con A*.

7.2.2. Con un waypoint en cada extremo del paso

		Coste	Tiempo Dijkstra	Tiempo A*	Tiempo total
Corto	1	372	24	44	68
	2	245	161	10	171
	3	363	41	18	59
Medio	1	1321	26	52	78
	2	627	145	18	163
	3	799	40	21	61
Largo	1	5465	23	199	222
	2	1171	158	31	189
	3	2784	38	40	78

Tabla 43: rewsultados con 2 waypoints

En la tabla 43 se recoge el tiempo de búsqueda con los waypoints en los extremos. El tiempo de búsqueda de Dijkstra aumenta con respecto al caso anterior.

7.2.3. Con tres waypoints

		Coste	Tiempo Dijkstra	Tiempo A*	Tiempo ms total
Corto	1	372	48	58	106
	2	241	180	11	191
	3	348	50	19	69
Medio	1	1321	41	57	98
	2	624	177	16	193
	3	799	39	23	62
Largo	1	5465	47	237	284
	2	1165	187	29	216
	3	2784	37	44	81

Tabla 44: resultados con 3 waypoints

En la tabla 44 están los tiempos de búsqueda con 3 waypoints (central y extremos), debido a que hay más waypoints Dijkstra tarda más pero el tiempo de A* se mantiene parecido salvo algunos casos como el mapa 7.

7.2.4. Con un waypoint en los extremos y cada 5 puntos

		Coste	Tiempo Dijkstra	Tiempo A*	Tiempo total
Corto	1	372	106	42	165
	2	239	435	8	443
	3	348	18	140	140
Medio	1	1321	108	48	156
	2	612	421	14	435
	3	799	129	24	153
Largo	1	5465	82	150	232
	2	1159	429	26	455
	3	2778	132	40	172

Tabla 45: resultados con waypoints en los extremos y cada 3 puntos

En la tabla 45 se sigue viendo el aumento del tiempo de búsqueda con Dijkstra pero sigue manteniéndose el tiempo de búsqueda de A*. Aquí el aumento del tiempo de Dijkstra es más claro ya que se generan muchos más waypoints que en los casos anteriores.

7.2.5. Con un waypoint en los extremos y cada 3 puntos

		Coste	Tiempo Dijkstra	Tiempo A*	Tiempo total
Corto	1	372	223	42	265
	2	239	1019	12	1031
	3	348	334	19	353
Medio	1	1321	231	51	282
	2	609	1002	17	1019
	3	799	364	22	75
Largo	1	5465	210	148	358
	2	1157	1026	30	1056
	3	2778	321	42	363

Tabla 46: resultados con un waypoint en los extremos y cada 5 puntos

En la tabla 46 se muestra el tiempo de búsqueda con un waypoint cada 3 puntos, que es el caso en el que más waypoints se guardan. El tiempo aumenta considerablemente pero se ha conseguido que la búsqueda con A* no aumente su tiempo de ejecución demasiado. Para comprobar esta mejora se comparará con el tiempo que se tardaría en usar A* directamente sobre todo el mapa.

7.2.6. Resumen costes

En la tabla 47 puede comprobarse la calidad de las soluciones una vez realizadas todas las pruebas comparadas con el coste de un algoritmo de A*.

		1 waypoint	2 waypoints	3 waypoints	Cada 5 puntos	Cada 3 puntos	A*
Corto	1	372	372	372	372	372	372
	2	251	245	241	239	239	233
	3	423	363	348	348	348	348
Medio	1	1475	1321	1321	1321	1321	1312
	2	652	627	624	612	609	570
	3	874	799	799	799	799	799
Largo	1	6033	5465	5465	5465	5465	5447
	2	1192	1171	1165	1159	1157	1103
	3	2893	2784	2784	2778	2778	2762

Tabla 47: resumen de costes

Como se puede ver en muchos casos las soluciones son sub-óptimas y en otras se ha logrado aproximar bastante e incluso a conseguir un camino con el mismo coste que con A*.

7.2.7. Resumen de tiempos de búsqueda

Se han tomado los tiempos medios en milisegundos de cuánto han tardado las búsquedas. Se tiene en cuenta el tiempo total de búsqueda pero también se han tomado los tiempos de búsqueda de Dijkstra y de A* que hacen ese total. En la tabla 48 se recoge el tiempo de búsqueda total de cada búsqueda:

		1 waypoint	2 waypoints	3 waypoints	Cada 5 puntos	Cada 3 puntos	A*
Corto	1	51	68	106	165	265	32
	2	65	171	191	443	1031	6
	3	45	59	69	140	353	2
Medio	1	47	78	98	156	282	164
	2	87	163	193	435	1019	59
	3	47	61	62	153	75	86
Largo	1	47	222	284	232	358	1153
	2	87	189	216	455	1056	209
	3	62	78	81	172	363	296

Tabla 48: tiempo de búsqueda de cada prueba

Y en la ilustración 24 se puede comparar cómo ha sido la evolución de los tiempos de cada tipo de búsqueda relacionado con el aumento del coste.

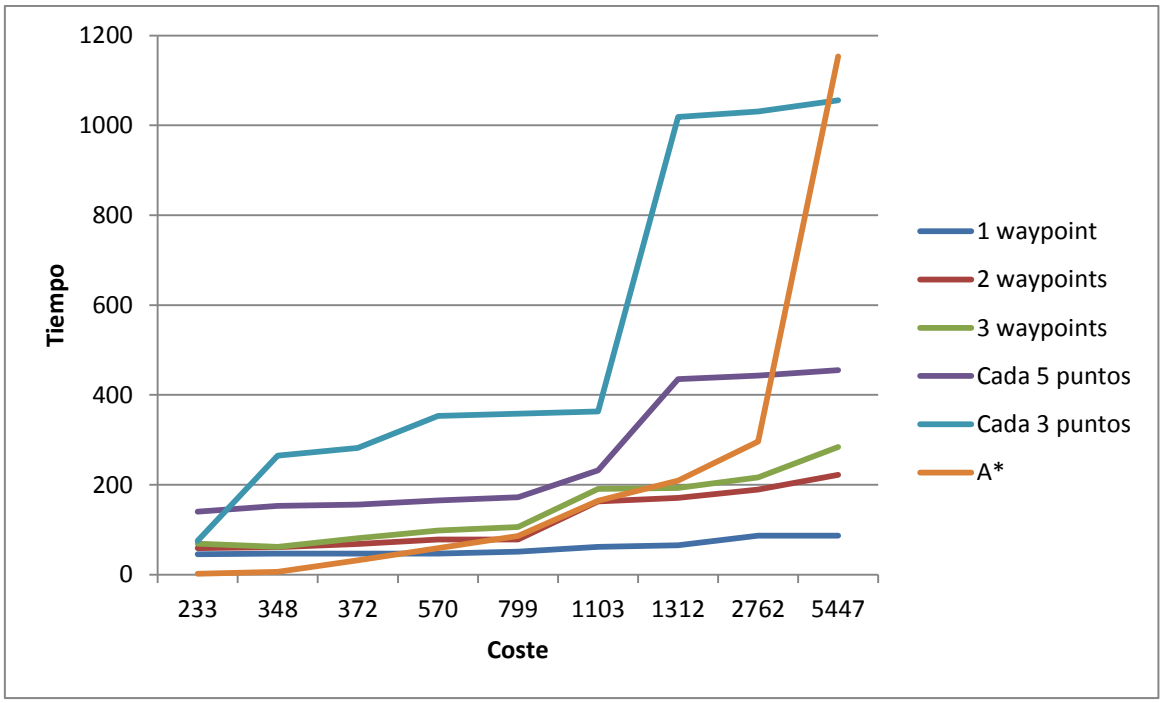


Ilustración 24: evolución del tiempo total de búsqueda con respecto al coste de la solución

El algoritmo de A* aumenta de manera exponencial mientras que con el uso de waypoints el crecimiento es más suave y es de aproximación lineal. Con las soluciones de un waypoint cada 5 y cada 3 se ve un pico en el punto donde A* comienza a aumentar pero después se estabiliza.

7.2.7.1. Tiempos de búsqueda de Dijkstra según el número de waypoints

Se puede comparar cómo ha evolucionado el tiempo de búsqueda del algoritmo de Dijkstra en cada mapa según el número de waypoints generados. En la tabla 49 se recogen estos tiempos.

	Mazmorra (1)		Dragon Age Origins (2)		Baldurs Gate (3)	
	Tiempo	Waypoints	Tiempo	Waypoints	Tiempo	Waypoints
1 waypoint	7	144	60	1020	23	222
2 waypoints	24	288	154	1712	40	380
3 waypoints	45	392	181	1840	42	418
Cada 5 puntos	99	1088	428	2406	125	916
Cada 3 puntos	221	1632	1015	3046	340	1316

Tabla 49: resumen de tiempos de búsqueda de Dijkstra

Puede verse la evolución del tiempo que tarda el algoritmo de Dijkstra en encontrar el camino más corto según el número total de waypoints en la ilustración 25.

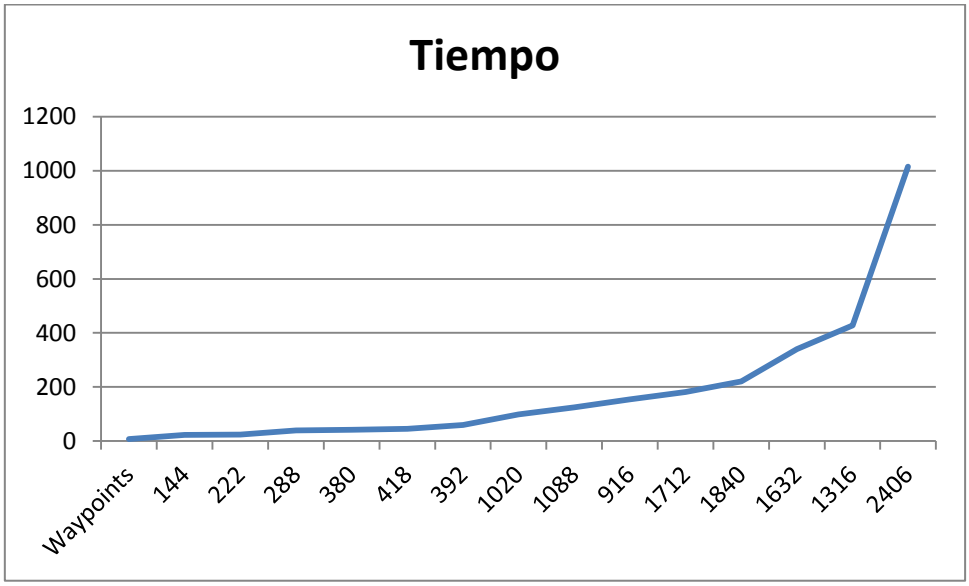


Ilustración 25: tiempo de búsqueda de Dijkstra según el número de waypoints

Se ve que aumenta de manera exponencial.

7.2.7.2. Mejora del tiempo de A* tras el uso de Dijkstra

En algunos casos se ha visto que usar Dijkstra primero ha hecho que la búsqueda haya tardado más en total, pero podemos ver cómo ha mejorado la búsqueda del camino final con A* tras el uso de Dijkstra, ya que se ha reducido el espacio de búsqueda. La tabla 50 muestra todos los tiempos del algoritmo de A* tras el proceso de Dijkstra y de A* sólo.

		1 waypoint	2 waypoints	3 waypoints	Cada 5 puntos	Cada 3 puntos	A*
Corto	1	44	44	58	42	42	32
	2	9	10	11	8	12	6
	3	24	18	19	140	19	2
Medio	1	40	52	57	48	51	164
	2	23	18	16	14	17	59
	3	22	21	23	24	22	86
Largo	1	40	199	237	150	148	1153
	2	29	31	29	26	30	209
	3	40	40	44	40	42	296

Tabla 50: tiempos de búsqueda de A*

Y en la ilustración 26 puede comprobarse como ha sido la evolución en el tiempo.

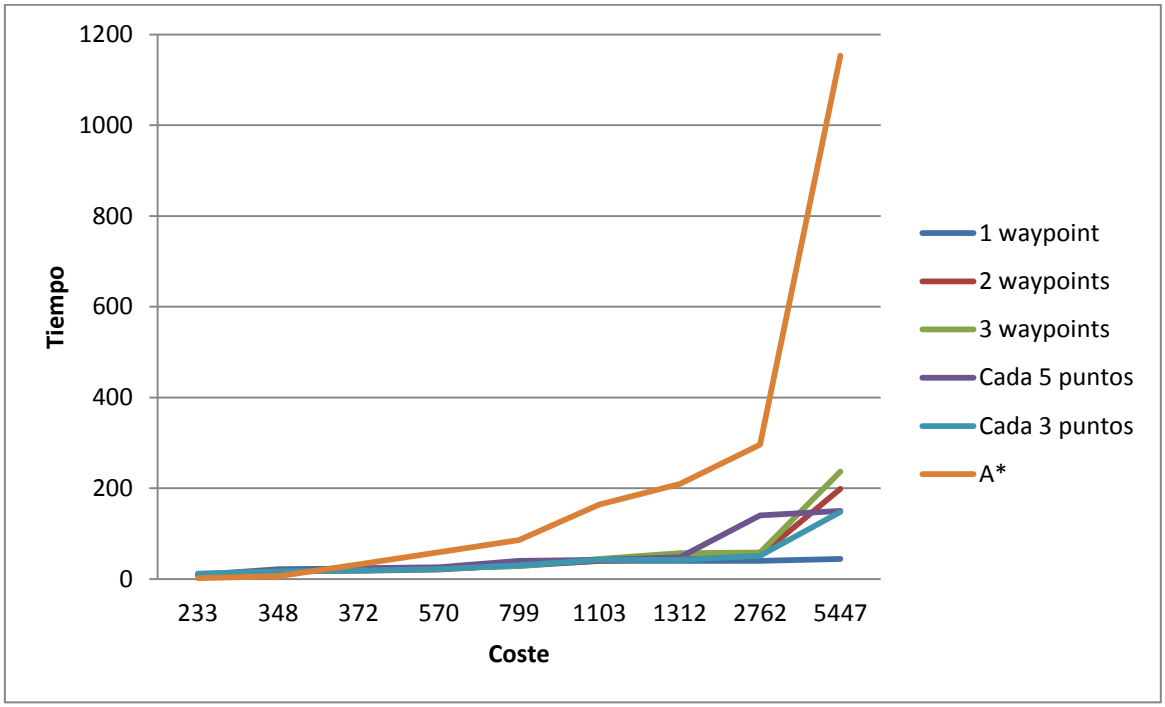


Ilustración 26: tiempo de búsqueda de A* (tras búsqueda con Dijkstra) con respecto al coste de la solución

Se puede decir que mientras el algoritmo de A* usado directamente sobre el mapa aumenta su tiempo de ejecución de manera exponencial con la longitud del camino, mientras que tras el preproceso mediante Dijkstra se consiguen tiempos con un aumento más gradual.

7.2.8. Comentarios sobre los resultados

Se puede decir que el tiempo que tarda en procesarse el mapa depende del tamaño del mapa y aumenta ligeramente cuando tienen que localizarse más waypoints como puede verse en la tabla 41 aunque son valores parecidos.

Las soluciones obtenidas mejoran cuando aumenta el número de waypoints definidos en el mapa. Esto se debe a que se añaden posibles caminos que están más cercanos a la solución. Los peores costes se dan siempre cuando se usa la búsqueda mediante un solo waypoint y van mejorando cuando su número aumenta. El problema de aumentar demasiado el número de waypoints es que aunque se aumenta la calidad de la solución se aumenta también el tiempo para procesar ese número tan alto de puntos.

Parece que la mejor opción está en generar entre 3 waypoints y un waypoint cada 5 puntos además de los extremos.

Como se ha visto, el uso de Dijkstra ayuda a mejorar el tiempo que tarda A* en dar con la solución.

8. Gestión del proyecto

8.1.Planificación inicial

Al comienzo del proyecto se realizó una planificación sobre el tiempo estimado que ocuparía el proyecto entero y cada fase del mismo. El inicio del proyecto estaba marcado el 1 de febrero y se realizó una media de 4 horas de trabajo de lunes a viernes cuando no había coincidencia de trabajo con exámenes o prácticas de las asignaturas, siendo estos días una media de dos horas de trabajo. El final del proyecto se marcó para el 5 de junio.

Para la realización del proyecto se han establecido diferentes fases. Primero se ha realizado un estudio del problema, definición de objetivos y consultas sobre el tema del trabajo. La segunda fase ha sido el desarrollo del programa en Java comenzando por la recogida de información del mapa y programando al final los algoritmos de búsqueda. Una vez terminado el sistema se han realizado las pruebas con las que se ha comprobado su funcionamiento y han podido realizarse las comparaciones y estimaciones acerca de la validez de la solución. Por último se ha realizado la documentación del sistema y de todo el trabajo llevado a cabo.

Durante la fase de estudio se han consultado otros trabajos acerca del tema elegido y se ha realizado el análisis de la información que iba a ser necesario recoger para su uso con los algoritmos escogidos.

En la fase de desarrollo primero fue necesario crear la clase que nos permite leer el mapa y luego toda la funcionalidad necesaria para obtener de este la información que en la fase de estudio se había establecido como necesaria. Una vez comprobado que el programa obtenía toda la información se ha programado la parte de búsqueda comenzando por el algoritmo de Dijkstra y después el A*, para al final unir sus esfuerzos.

La fase de experimentación se divide en elegir los mapas que van a evaluarse y los tipos de pruebas que se iban a realizar, que en este caso ha sido variando el número de waypoints que se disponían para las regiones y la distancia entre los puntos inicial y final evaluando los costes y tiempo tardado en encontrar las soluciones.

La documentación del proyecto recogerá la información sobre el trabajo realizado con el análisis de la información y los algoritmos de búsqueda escogidos, el diseño del sistema, las pruebas realizadas y sus resultados. La fase de documentación se solapa con la de experimentación ya que comienza antes de tener los resultados y no finaliza hasta que se tienen todos los resultados recogidos.

La tabla 51 recoge las fechas y los tiempos iniciales de las fases.

	Actividad	Inicio	Duración	Fin
Estudio previo	Análisis de trabajos previos	01/02/2016	5	05/02/2016
	Selección de algoritmos	03/02/2016	3	08/02/2016
	Análisis de información	08/02/2016	3	11/02/2016
Desarrollo del sistema	Lectura del mapa	11/02/2016	7	19/02/2016
	Procesado de información	22/02/2016	15	11/03/2016
	Algoritmos	14/03/2016	24	14/04/2016
Pruebas	Selección de mapas	18/04/2016	3	20/04/2016
	Pruebas	21/04/2016	5	27/04/2016
Documentación	Escritura del documento	01/05/2016	23	31/05/2016

Tabla 51: planificación inicial

En el diagrama de Gant de la ilustración 27 se muestra la planificación inicial comentada.

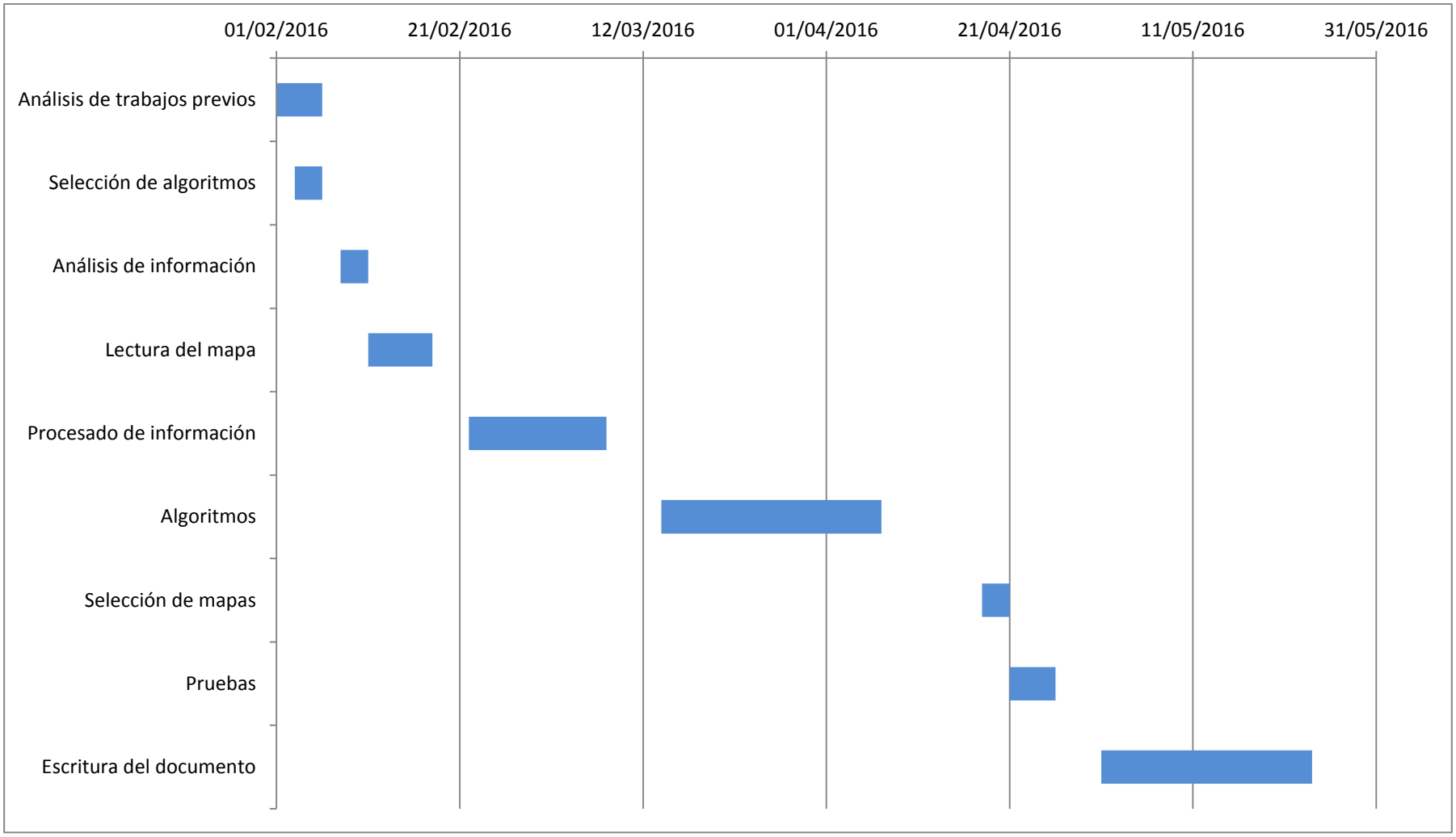


Ilustración 27: diagrama de Gant inicial

8.2.Planificación final

Debido a la realización de exámenes y prácticas durante el cuatrimestre, y a otra serie de contratiempos de carácter personal algunas fases del proyecto se fueron retrasando, alargando algunas fases y retrasando el inicio de otras. Esto no ha comprometido la decisión inicial de finalizar el proyecto para la entrega de junio de 2016.

De la planificación inicial de tener un trabajo diario constante no ha podido siempre cumplirse ya que en determinados días estas horas dedicadas al proyecto tuvieron que dedicarse a la preparación de algún examen o desarrollo de las prácticas. Se estima la duración de cada fase en días, pero no corresponde con días totales de trabajo.

La planificación final en fechas y días puede verse en la tabla 52.

	Actividad	Inicio	Duración	Fin
Estudio previo	Análisis de trabajos previos	15/02/2016	5	20/02/2016
	Selección de algoritmos	16/02/2016	3	19/02/2016
	Análisis de información	20/02/2016	5	25/02/2016
Desarrollo del sistema	Lectura del mapa	27/02/2016	12	10/03/2016
	Procesado de información	14/03/2016	32	15/04/2016
	Algoritmos	18/04/2016	32	20/05/2016
Pruebas	Selección de mapas	19/05/2016	1	20/05/2016
	Pruebas	21/05/2016	12	02/06/2016
Documentación	Escritura del documento	23/05/2016	14	06/06/2016

Tabla 52: planificación final

Y se corresponde con el diagrama de Gant de la ilustración 28.

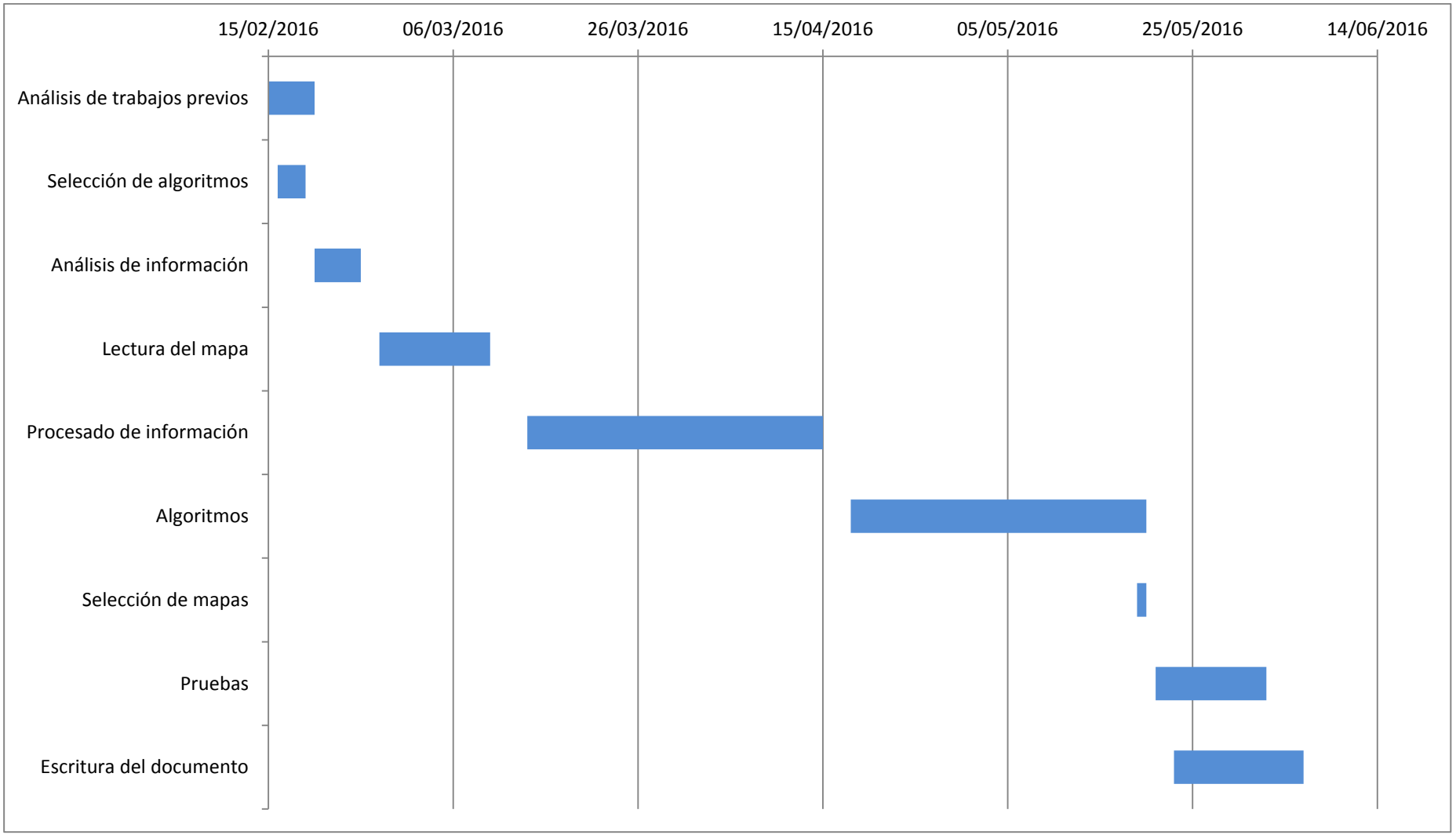


Ilustración 28: diagrama de Gant final

8.3.Presupuesto

Los resultados de este trabajo pueden tener un valor económico para empresas o personas involucradas especialmente en el desarrollo de videojuegos. Por ello se realizará un presupuesto para conocer los gastos asociados al desarrollo de este trabajo.

Para calcular este presupuesto se han tenido en cuenta los gastos generados por el personal del proyecto y el material y software utilizado. Los gastos derivados por el personal se cuentan en horas de trabajo siendo de 26'4€ la hora durante los trabajos de análisis y documentación y de 22'18 € la hora el trabajo de desarrollo. Para calcular las horas dedicadas a cada trabajo se estiman unas 4 horas de trabajo por día según el gráfico Gant final visto en la ilustración 19. En la tabla 53 se calcula el gasto total del personal.

Trabajo	Dias	Horas totales	Coste/hora	Total
Análisis	13	52	26,4	1372,8
Desarrollo	76	304	22,18	6742,72
Pruebas	13	52	22,18	1153,36
Documentación	14	56	26,4	1478,4
				10747,28

Tabla 53: gastos de personal

El total del personal asciende a 10.747'28 € a los que deben añadirse los gastos del material y el software de la tabla 54.

Material	Coste	Amortización (años)	Amortización (€/año)	Duración de proyecto (meses)	Coste en proyecto
Ordenador de sobremesa	970	5	194	4	64,67
Ordenador portátil	514	3	171,34	4	57,12
Entorno Eclipse	0	0	0	4	0
Microsoft Office	49	1	49	4	16,34
					138,11

Tabla 54: gastos de material

El gasto del material contabilizado sólo durante el tiempo de duración del proyecto ha sido de 138'11 €.

El coste total del proyecto asciende a 12.528'20 € y se resume en la tabla 55.

Trabajo	10747,28
Material	138,11
Costes indirectos (15%)	1632,81
Total	**12518,20**

Tabla 55: coste total

9. Conclusiones

9.1.Conclusiones sobre los resultados

Tras comprobar el coste de las soluciones encontradas con la combinación de los dos algoritmos escogidos y compararlo con el uso de un único algoritmo de A* puede decirse que las soluciones de este trabajo son sub-óptimas. Esto se debe al uso de los waypoints ya que en determinados casos por la propia definición se obliga a que el camino pase por estos puntos hace que se aleje del punto final y alargue unos pasos el camino. Para alcanzar la solución óptima sería necesario que los waypoints se ajustaran a los puntos por los que pasaría un algoritmo de A*.

A pesar de estos resultados con respecto al camino sí se ha logrado reducir el tiempo de búsqueda y el número de nodos expandidos, lo que permite reducir la memoria de cada cálculo además de alcanzar la solución en menor tiempo.

Se ha logrado cumplir el objetivo de encontrar soluciones con el uso de regiones, la generación de waypoints y la obtención del grafo que los conectaba para poder hacer uso del algoritmo de Dijkstra.

Por lo que ha podido verse en los resultados las mejores soluciones aparecen a partir de un conjunto de al menos 3 waypoints en los que estén incluidos los puntos extremos de las aberturas. Esto se debe a que para rodear un muro es mejor hacerlo desde su extremo más cercano que alejarse de la solución pasando por el punto medio. Sin embargo al alcanzar la solución o al pasar entre algunas regiones se genera un camino más recto pasando por la mitad de la abertura en lugar de acercarse hasta un extremo. Esto se ha visto con la solución que incluía el punto medio y los extremos, y como al guardar los extremos y un punto de cada 3 se podía perder ese punto medio, y por tanto las soluciones eran peores.

9.2.Problemas encontrados

Además de los retrasos producidos por actividades ajenas al desarrollo de este trabajo se han sucedido una serie de problemas durante el desarrollo que han retrasado algunas de las fases.

Durante la fase de desarrollo los primeros problemas encontrados se dieron con el algoritmo de generación de regiones ya que con las primeras versiones las regiones no se adaptaban al mapa de la manera deseada. Tras algunas modificaciones finalmente se consiguió que las regiones se generaran en los mapas adaptándose a las formas de los muros y obstáculos.

La primera solución para el algoritmo de Dijkstra pasaba por generar un nodo con las regiones en lugar de los waypoints del mapa y asociarle coste 1 al paso entre cada región. Esto no permitía asociar el waypoint más cercano a la solución y además tras algunas pruebas se vio que en algunos casos la solución se alejaba mucho de ser óptima.

Otro problema fue el de encontrar el mejor coste para el movimiento entre los puntos y la función heurística. La distancia euclidiana funcionaba bien para el algoritmo de A* cuando

debía encontrar el camino entre el nodo inicial y el de meta pero no funcionaba bien para encontrar el mejor camino entre las regiones. Cuando se cambió a la solución de la estructura de datos *distance* y el uso de los waypoints para el algoritmo de Dijkstra las soluciones se acercaban más al camino óptimo.

9.3.Líneas futuras

Como mejoras en vista de que la solución ha resultado bastante aproximada y además se ha logrado reducir el tiempo de búsqueda las posibles mejoras serán reducir el tiempo que tarda Dijkstra en encontrar el camino más corto y reducir la memoria utilizada para crear las soluciones.

Con esta solución hay que encontrar un equilibrio en el número de waypoints generados para no sobrecargar el algoritmo de Dijkstra. Sería interesante guardar un número aleatorio de waypoints en función de la amplitud de la abertura encontrada.

También podría extenderse el uso de sólo el algoritmo de A* en puntos que aun estando en distintas regiones estén a una distancia corta. Mediante una serie de pruebas podría determinarse el umbral de esta distancia. Las distancias entre waypoints que no son los puntos inicial y final que se añaden al grafo a la hora de hacer la búsqueda podrían mantenerse en memoria de forma que no fuera necesario hacer todos los cálculos cada vez.

10. Anexos

10.1. Resumen inglés

Path-Finding in Maps

Introduction

The use of computers has enabled us to solve problems that, otherwise, need too many human resources and long time or even the solution cannot be found. Two of the goals for computing are to find the right algorithms that solve the problems in shorter periods of time and to find the optimal solution.

The heuristic is the partial knowledge of a domain that permits to guide the search of the solution of problems. The heuristic search is applied to the problem of finding routes from an initial state to a final one. With the use of a heuristic function, that indicates the distance to the solution from an intermediate state and gives information about the best state to continue to reach the solution, the search is optimal using an admissible function.

Many videogames have maps on which the characters' actions are performed. These maps are stored in memory and are showed to the player processed and detailed but the computer works with simpler models. The maps are traversed by both players and NPC's. The movement of the NPC's is controlled by the computer and their route has to be calculated at the execution time of the game.

The routes can be pre-calculated or calculated every time the route has to change. This gives more computing load to the computer, especially in big maps. The use of efficient algorithms is important to reduce this workload and to get optimal routes in short time. One solution is the use of path finding search algorithms that have previous information about the map.

In this work heuristic search is used to find routes between two points in a grid-map. The objective is to find solutions in short time and using as little memory as possible. This document explains this work, the previous analysis, the algorithms, the solutions and its results. Finally a section explains the cost, planning and the conclusions.

Context

Videogames

A videogame is an electronic game that is shown in a screen, runs on a console or computer and allows the player to interact with the program using a remote control, a keyboard or a movement. They were invented during the 1950's trying to give some recreational use to the computers. The programs were very simple and not for a commercial use until the end of the 1970's with the invention of the arcade machine. Since then the videogames have evolved in complexity, history, duration, gameplay and have grown very much in popularity. Nowadays It is one the richest industries in the world. There are games for

the entire society with different genres and they have expanded for all the platforms like smartphones

The videogames take place on a stage which can be a simple background image, a board or a terrain in which the characters will walk on. A map of the game is showed to the player and the computer uses that to calculate the route to the objectives.

Maps

Like the games, the maps have also grown in complexity and size. They are not only the background of the game, they are also the stage for the characters' movements. They are stored in memory and the way to show them for the player is different to the maps that the computers will work with. The player will see the graphic and processed representation of the map with forms, textures, backgrounds and three-dimensional details. The computer works with geometrical forms that divide the map. The movement is represented by graphs and depends on the form of the map and the type of movement.

The maps can be divided also by interconnected regions for the biggest maps through points named "waypoints" that are connected between them and represent a search space in the regions. The complexity of searching increases with the size of the map, the dimension of the movements and the allowed moves.

A star (A*)

It is a search method in graphs. It was developed in 1968 by Peter E. Hart, Nils J. Nilsson and Bertham Raphael [5]: This algorithm finds the path with the lower cost between the origin and the goal.

A* uses the evaluation function $f(n) = g(n) + h(n)$ where $g(n)$ represents the cost of the path and $h(n)$represents the value of the heuristic function from n to the goal. The algorithm has two auxiliary data structures to analyze the nodes: an open list for the nodes to visit ordered by the value of the function and a closed list for the visited nodes. At every step, the algorithm selects the first node of the opened list and if it is not the goal node, it expands its child nodes, calculating their $f(n)$. These nodes are stored in the open list and the selected node in the closed list.

A * is complete, if there is a solution, A* will find it. It is important to use an admissible heuristic.

Dijkstra

The Dijkstra algorithm is applied to determine the shortest path from an origin node to the rest of the nodes in a weighted graph. Its name is for Edsger Dijkstra who described it in 1959 [6].

The idea of this algorithm is to explore the shortest path that starts in the origin node and goes to other nodes. When the algorithm finds the shortest path from the origin to the other nodes, it stops.

For its operation the algorithm starts from an origin node with an open list of nodes to evaluate. In each step it analyzes the adjacent nodes and chooses the one with the lowest cost. The cost of a node will be the cost of its connection adding the cost to reach this node.

Grid Based Path Planning Competition

There is an annual competition about the path-finding search in grid maps whose objective is the study and applications of finding algorithms in videogames with some restrictions of memory and time. The competition starts in 2012 and, since then, many competitor groups have entered their code. The competition starts every 1st of December, but contestants can join the competition all over the year.

Moving AI Lab

It is located in the University of Denver and is under the direction of Professor Nathan Sturtevant. There the competition of Grid-Based Path Planning was founded and its studies are about the applications of search algorithms. There are works about the use of A*, heuristics and their applications in videogames.

Objectives

The objective is to develop searching techniques to find paths in two-dimensional grid-like maps given an initial and a final point. These techniques should balance search time and needed memory. In order to perform the search process efficiently, the algorithm should store some information in memory.

Previous classification in regions for the maps is going to be made and connect them by waypoints. The distance between the waypoints will be calculated. Some sub-objectives are to develop the function of classification in regions and localize the connections. With these waypoints the program forms the graph that represents the connections between regions.

Firstly, a search in this graph of regions is going to be done with a function that returns the path of the waypoints from the origin region to the end region. With this path a search will be done in the points of the map connecting every pair of waypoints.

The solution will be an ordered list of points with the route to follow. For this search the Dijkstra algorithm is used for the search between regions and the A* algorithm for the search in the points of the map. The program will implement the needed data structures and the algorithms. The program has to be able to find the solution to the search in a balanced time and memory. It has to work with the maps of the competition [9].

Design of the system

To reach the objectives a java program will be designed and programmed to work according to some requirements. Some information will be stored after reading the map and before starting the search in order to streamline the search process. The program will read the map and will store this information.

The information needed refers to the regions, waypoints and distances. The previous information allows to expedite the search and the execution in a videogame can be more fluent. There are some classes in the system to process the map and store the information. The relations between them let the program have all the information and run. Some classes do basic actions, whilst others read the map and do the search.

The *Main* class executes the starting of the program and the order of the execution of the others classes. The *Reader* class reads the map and stores it in a basic form.

The *GameMap* class contains the basic matrix of the map with 1 and 0 for the points and the data structures to store the map information. This class classifies the map in regions and creates a matrix with the region of each point. *GameMap* also contains instances of *RegionMap* that store the number of region and its waypoints. The *Waypoint* class has the point of a waypoint and the two regions this point links. The *Distance* class contains two waypoints and the Euclidean distance between them. The *GameMap* class has some list of *Region*, *Waypoint* and *Distance* and contributes with all the necessary information for the search. The class *Writer* creates a document with the map in regions.

The search starts with the *Search* class that analyzes the points and selects the type of search. This class also checks if the points are valid. The result of the search will be stored in an *Result* object that contains the time, the cost and the nodes of the search. The *Dijkstra* class searches between the regions using Dijkstra, and the *Astar* class searches between the points using A*. *Astar* class requires instances of the class *Node* that represents a search node and *Successor* that is an expanded node from a previous one.

The relations between these classes are showed in the illustration 29.

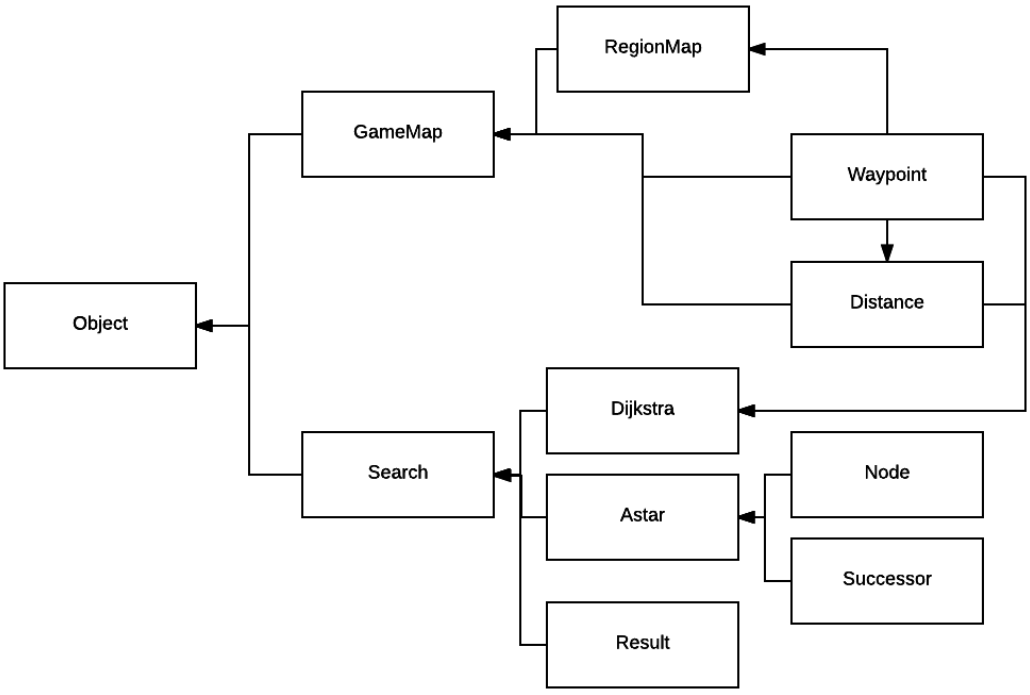


Illustration 29: object hierarchy

Design of the solution

This section explains how every module of the program works and the steps of execution. The program will classify an initial map in regions, find the connections and calculate the distances. Firstly, the search in the region space is done, finally, the search in the map's points linking the connections of the regions.

Map Reading

The file that contains the map is given to the program as an argument. The first lines are the width and the height of the map that are stored in two integer variables that will be used to declare a matrix to store the map. The file is coded with "." o "G" for the free points and "@", "O" for not accessible points, "T" for trees that are obstacles and "W" for water, that is not walkable. To simplify this, the accessible points will be coded with "1" and "0" for the not accessible ones.

The matrix with the map's points, width and height are attributes of the class and before reading the map the program takes those values from the instance of *Reader* and the parameters are given to an instance of *GameMap* to continue with the generation of regions and waypoints.

Regions

After reading the map, it has to be processed to generate the regions according to the walls and the obstacles. The instance *GameMap* needs the matrix of the map coded with "1" and "0", the width and the height of the map for the constructor.

This class has an attribute *zone* that is an integer matrix with the same dimension of the map and every grid is a point of the map and contains the number of region of the point. The obstacles have the code "0" and the accessible points are "-1" whilst their region is unknown. To generate the regions it makes a flooding process that looks for the walls and delimits the region. A function that uses the initial map and the zone map makes this process. These variables are attributes of *GameMap* so the function does not have to receive any parameter.

Firstly, the function looks for the first free point at the left, and starts the flooding process marking the points from left to right until it hits a wall. Then it returns to the left in the next line.

When it returns to the right, if the point is not free (it has an obstacle or it is yet in a region) and the next point is not free either, the algorithm stops and makes a region with the identified points. This process ends when the entire map is read and there is no free point.

Waypoints

The waypoints are the connections between the regions and their direction goes from an origin region to a final one. A function reads the matrix of the map's regions to find points with neighbor regions without a wall between them. There are two types of waypoints:

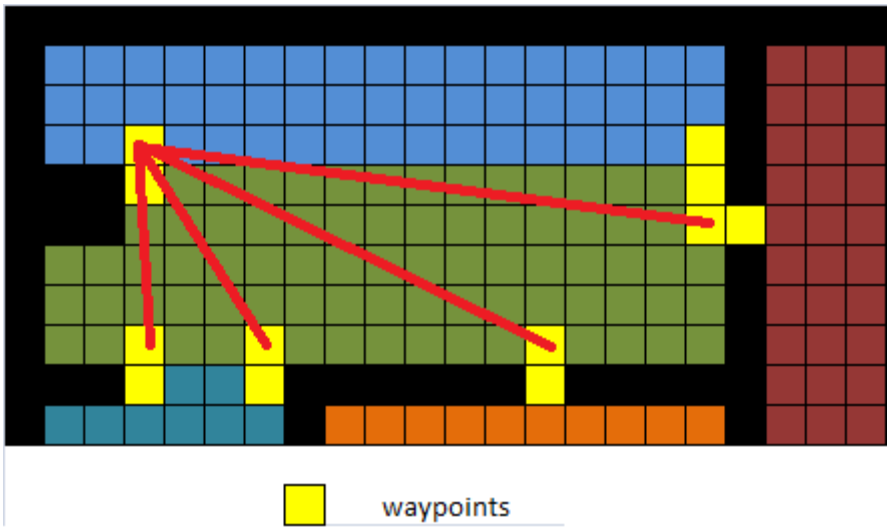
- **Horizontal waypoints:** a region is above another region, the x coordinate is the same and the y coordinate of the region below is one line more than the region above.
- **Vertical waypoints:** a region is to the right of another one, the y coordinate is the same for the two regions and the x coordinate of the right region is one column more.

The waypoints belong to each region and there is a waypoint for each region that is neighbor of the waypoint of the other region. With this definition different solutions have been designed with different number of waypoints according to the size of the opening.

- **One waypoint** at the central point.
- **Two waypoints** at the extreme sides of the opening.
- One waypoint at the **central point** and the **extreme sides**.
- Two waypoints at the **extremes sides** and a waypoint **every 5 points**.
- Two waypoints at the **extremes sides** and a waypoint **every 3 points**.

Distances

The Dijkstra algorithm uses a graph that represents the connections between the waypoints. The distances are the Euclidean distance from a region with the waypoints of the neighbor region apart from the actual region. A distance can be seen in the illustration 30, where the region 1's waypoints are connected with the region 2's waypoints.



Ilustration 30: waypoints

Search

The search is performed using the Dijkstra algorithm and A*. Different cases can be found according to the chosen points:

- **One of the points is not valid**: if one of the points is an obstacle or a wall it is not possible to find a path to this point. It generates an error message and the search does not execute.
- **Points at the same region:** in this case it is not necessary to search through the region space and directly it searches with the A* algorithm.
- **Points in different regions**: if the two points are in different regions firstly it searches between the region space with the Dijkstra algorithm and it obtains a path with the waypoints the A* search has to cross. When Dijkstra has found the solution it searches a path between every pair of connected waypoints with the A* algorithm from the origin to the goal.
- **No reachable points**: if the points are valid the search process starts but in some cases the final point is not reachable from the initial point and there is no path. The node list returns empty and it shows an error message.

Results

Three maps have been chosen to do the testing: with squared forms, an irregular one and a bigger one to see the variations in searching time. The search testing has been done with the different number of waypoints defined in the solution.

Maps

The maps used in the testing have been chosen according to their properties to verify the function of the region generation and the waypoints. These maps can be consulted in annex 9.

The number of regions in each map is collected at the table 56.

	Map		
	Mazmorra (1)	**Dragon Age Origins (2)**	**Baldurs Gate (3)**
Regions	76	274	79

Table 56: number of regions per map

And the generation time and the number of waypoints according to the type of search can be consulted in the table 55.

	Mazmorra (1)		Dragon Age Origins (2)		Baldurs Gate (3)	
	Time	Waypoints	Time	Waypoints	Time	Waypoints
1 waypoint	335	144	218	1020	262	222
2 waypoints	283	288	217	1712	285	380
3 waypoints	339	392	229	1840	270	418
Each 5 points	290	1088	219	2406	298	916
Each 3 points	326	1632	223	3046	281	1316

Table 57: processing time of maps and waypoints

The processing time increases with the size of the map and the number of waypoints to generate but it is not a strong growth.

Testing

In this section the costs of path and search times according to the type of search selected are summarized. In each map three paths of different distance have been chosen to compare the search times. These paths are of increasing length. The cost is compared with the search cost with A* algorithm, that gives the optimal solution. The table 58 shows a summary with the cost of the search for each type of search.

		1 waypoint	2 waypoints	3 waypoints	Each 5 points	Each 3 points	A*
Small	1	372	372	372	372	372	372
	2	251	245	241	239	239	233
	3	423	363	348	348	348	348
Medium	1	1475	1321	1321	1321	1321	1312
	2	652	627	624	612	609	570
	3	874	799	799	799	799	799
Large	1	6033	5465	5465	5465	5465	5447
	2	1192	1171	1165	1159	1157	1103
	3	2893	2784	2784	2778	2778	2762

Table 58: summary of costs

The costs for the defined solution are suboptimal. The search time has also been counted and the total time in milliseconds is collected in the table 59.

		1 waypoint	2 waypoints	3 waypoints	Each 5 points	Each 3 points	A*
Small	1	51	68	106	165	265	32
	2	65	171	191	443	1031	6
	3	45	59	69	140	353	2
Medium	1	47	78	98	156	282	164
	2	87	163	193	435	1019	59
	3	47	61	62	153	75	86
Large	1	47	222	284	232	358	1153
	2	87	189	216	455	1056	209
	3	62	78	81	172	363	296

Table 59: total search time

It can be seen that in the case of the largest search and the biggest maps the search time with Dijkstra and A* has been reduced with respect to the search time of A*. This is explained because the first search with Dijkstra reduces the search space of the entire map to the regions of the path. To better analyse this time, the search time of Dijkstra and the search time of A* afterwards have been counted. The Dijkstra's search time can be consulted on the table 60.

	Mazmorra (1)		Dragon Age Origins (2)		Baldurs Gate (3)	
	Tiempo	Waypoints	Tiempo	Waypoints	Tiempo	Waypoints
1 waypoint	7	144	60	1020	23	222
2 waypoints	24	288	154	1712	40	380
3 waypoints	45	392	181	1840	42	418
Cada 5 puntos	99	1088	428	2406	125	916
Cada 3 puntos	221	1632	1015	3046	340	1316

Table 60: Dijkstra's search time

This time depends on the number of waypoints, so when this number increases this time increases also. The search times of A* before using Dijkstra are summarized in the table 61.

		1 waypoint	2 waypoints	3 waypoints	Each 5 points	Each 3 points	A*
Small	1	44	44	58	42	42	32
	2	9	10	11	8	12	6
	3	24	18	19	140	19	2
Medium	1	40	52	57	48	51	164
	2	23	18	16	14	17	59
	3	22	21	23	24	22	86
Large	1	40	199	237	150	148	1153
	2	29	31	29	26	30	209
	3	40	40	44	40	42	296

Table 61: A*'s search time

This time compared with the search time of A* has improved and is shorter in the cases of the large routes.

Project

Planning

At the start of the project a planning about the estimated time of the project and every phase of it has been done. The start of the project was marked on February 1st and it has been worked during 4 hours from Monday to Friday. The end of project was marked on June 5th.

The project is divided in different phases. The first phase is the initial study of the problem, objectives definition and consults about the work. The second phase is the development of the program in java starting with the map reading, the generation of regions

and waypoints and implementation of the search algorithms at the end. When the system was finished, the testing has been done to check the proper operation of the system and to perform the comparisons and check the solution's validity. Finally, the documentation of the system and all the work done have been made.

Due to several delays because of exams and practices of other subjects the start and other phases of the project have been postponed, but the decision to finish the project in June 2016 has not committed. The final planning is summarized in the table 62.

	Activity	Start	Duration	End
Previous study	Analysis of previous works	15/02/2016	5	20/02/2016
	Algorithms selection	16/02/2016	3	19/02/2016
	Analysis of information	20/02/2016	5	25/02/2016
System Development	Map reading	27/02/2016	12	10/03/2016
	Information processing	14/03/2016	32	15/04/2016
	Algorithms	18/04/2016	32	20/05/2016
Testing	Maps selection	19/05/2016	1	20/05/2016
	Testing	21/05/2016	12	02/06/2016
Documentation	Document writing	23/05/2016	14	06/06/2016

Table 62: final planning

Costs

The results of this work can have an economic value for enterprises or people involved in videogames development. That is why a budget of the project is going to be included.

To calculate these costs, the costs of the staff, the material and software used have to been considered. The costs of the staff is counted on hours of work and it is 26'4 € for the process of analysis and documentation and 22'18 € for the process of development and testing. The staff cost is up to 10.747'28 €.

The cost of the material and software is up to 138'11 €.

The totals are collected in the table 63 and are up to 12.518'20 €.

Work	10747,28
Equipment	138,11
Indirect costs (15%)	1632,81
Total	**12518,20**

Table 63: total costs

Conclusions

About the results

After the testing it can be checked that the solutions are suboptimal. In some cases the use of waypoints forces the route to pass through these points and it moves away the route from the solution. To avoid it the waypoints had to coincide with the points of the route of A*. In spite of these results, an improvement on the search time has been achieved.

The objective of getting the solution with the use of regions, the connections of the waypoints and the graph of regions has been achieved.

The better results are given with the solution of 3 or more waypoints in which the points on the extreme sides of the opening are included. This is because it is better to cross the extreme side of a wall to surround it. On the other hand it is better to cross the opening in the middle to pass through a region, so with the middle point it obtains better results than the solutions that do not have this point.

Future lines

Reducing the search time of Dijkstra and the memory used for the solutions could be considered as improvements of the solution. To this solution a balance between the number of waypoints and the size of the opening can be found.

In some cases the initial and final point are in different regions but the distance that separates them is small, and the use of A* algorithm directly is faster than using Dijkstra and A*. With some testing, the range of the distance can be determined.

The graph of waypoints could be stored in memory, so it does not have to make all the calculations in each search.

10.2. Lectura del mapa

```
public Reader(String map) {
            try {
                  this.file = new FileReader(map);
                  br = new BufferedReader(file);
            } catch (FileNotFoundException e) {
                  e.printStackTrace();
            }
            try {
                  String height = br.readLine();
                  y = Integer.parseInt((height.substring(7))); //alto
                  String width = br.readLine();
                  x = Integer.parseInt((width.substring(6))); //ancho

                  //Instanciar matriz de mapa
                  this.map = new byte[x][y];
                  String line = br.readLine();
                  while (line != null) {
                        for (int j = 0; j < y; j++) {
                              for (int i = 0; i < x; i++) {
                                    if (line.charAt(i) == '.' ||
                                          line.charAt(i) == 'G') {
                                          this.map[i][j] = 1;
                                          //Punto libre
                                    } else
                                          this.map[i][j] = 0;
                                          //Obstáculo
                              }
                              line = br.readLine();
                        }
                  }
                  br.close();
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
```

10.3. Generación de regiones

```
public void generateRegions() {
		//Valores iniciales de la matriz
		for (int j = 0; j < this.heigh; j++) {
			for (int i = 0; i < this.width; i++) {
				if (this.map[i][j] == 1) {
					zone[i][j] = -1;
				} else
					zone[i][j] = 0;
			}
		}
		int xLeft = 0; //Punto libre más a la izquierda
		int yLeft = 0;
		boolean free = false;
		boolean shrunkR = false, shrunkL = false;
		int actualRegion = 1;
		int x = 0;
		int y = 0;
		//Buscar primer punto libre
		do {
			free = false;
			for (int j = 0; j < this.heigh; j++) {
				for (int i = 0; i < this.width; i++) {
					if (this.zone[i][j] == -1) {
						xLeft = i;
						yLeft = j;
						free = true;
						break;
					}
				}
				if (free)
					break;
			}
			shrunkR = false;
			shrunkL = false;

			if (yLeft == 0)
				yLeft = 1;
			y = yLeft;
			do {
				//Generación de regiones
				x = xLeft;
				if (this.zone[x][y] == -1)
					this.zone[x][y] = actualRegion;
				while (this.zone[x + 1][y] == -1
						&& this.zone[x + 1][y - 1] != -1) {
					x++;
					this.zone[x][y] = actualRegion;
				}
				if (this.zone[x + 1][y - 1] == actualRegion)
					shrunkR = true;
				else if (this.zone[x][y - 1] != actualRegion
					&& shrunkR) {
					while (this.zone[x][y] == actualRegion) {
						this.zone[x][y] = -1;
						x = x - 1;
					}
					break;
				}
				x = xLeft;
				y = y + 1;
```

```
                    if (y >= this.heigh) //fuera del mapa
                          break;
                    while (this.zone[x][y] != 0
                           && this.zone[x][y] != -1
                           && this.zone[x][y - 1] == actualRegion){
                          x++;
                    }
                    while (this.zone[x][y] != 0 &&
                           this.zone[x - 1][y] == -1
                           && this.zone[x - 1][y - 1] == 0) {
                          x--;
                    }

                    if (x >= this.width)
                          break; //Fuera del mapa
                    xLeft = x;

                    if (this.zone[x - 1][y - 1] == actualRegion) {
                          shrunkL = true;
                    } else if (this.zone[x][y - 1] != actualRegion
                                && shrunkL) {
                          break;
                    }

              } while (true);
              //Añadir región a la lista
              this.regions.add(new RegionMap(actualRegion));
              actualRegion++;
        } while (free);
}
```

10.4. Generación de waypoints

```
public void generateWaypoints() {

int actualRegion = 1;
int regionX = 0;
int regionY = 0;
do {
	//Buscar región actual
	boolean region = false;
	for (int j = 0; j < this.heigh; j++) {
		for (int i = 0; i < this.width; i++) {
			if (zone[i][j] == actualRegion) {
				regionX = i;
				regionY = j;
				region = true;
				break;
			}
		}
		if (region)
			break;
	}
	//Punto a evaluar
	int x = regionX;
	int y = regionY;
	//Regiones conectadas
	int startRegion;
	int endRegion;
	Waypoint aux;
	while (zone[x][y] == actualRegion) {
	// Comprobamos regiones vecinas a la derecha
	if (zone[x + 1][y] != actualRegion
		&& zone[x + 1][y] != 0) {
		startRegion = actualRegion - 1;
		endRegion = zone[x + 1][y] - 1;
		int firstX = x; //primer extremo
		int firstY = y;
		while (zone[x + 1][y + 1] == endRegion
			&& zone[x][y + 1] == startRegion) {
				y++;
		}
		int lastX = x; //Segundo extremo
		int lastY = y;
		switch (mode) {
		case 1:
		// Para waypoint punto central
		 if(lastY != firstY){
		 int X = ((lastX + firstX) / 2);
		 int Y = ((lastY + firstY) / 2);
		 aux = new Waypoint(regions.get(actualRegion),
					regions.get(endRegion),
					new Point(X, Y));
		regions.get(actualRegion).getConnections().add(aux);
		connections.add(aux);
		aux = new Waypoint(regions.get(endRegion),
			regions.get(actualRegion),
			new Point(X+1, Y));
		regions.get(endRegion).getConnections().add(aux);
		connections.add(aux);
		}else{ //Fin if
			//Si la abertura es un solo punto
			aux = new Waypoint(regions.get(actualRegion),
```

```
                regions.get(endRegion), new Point(firstX, firstY));
                regions.get(actualRegion).getConnections().add(aux);
                connections.add(aux);
                aux = new Waypoint(regions.get(endRegion),
                      regions.get(actualRegion),
                      new Point(firstX+1,firstY));
                regions.get(endRegion).getConnections().add(aux);
                connections.add(aux);
                }//Fin else
                break;
                case 2:
                // Para dos waypoints
                aux = new Waypoint(regions.get(actualRegion),
                                   regions.get(endRegion),
                                   new Point(firstX, firstY));
                regions.get(actualRegion).getConnections().add(aux);
                connections.add(aux);
                aux = new Waypoint(regions.get(endRegion),
                                   regions.get(actualRegion),
                                   new Point(firstX + 1, firstY));
                regions.get(endRegion).getConnections().add(aux);
                connections.add(aux);
                //Si la apertura es de más de un punto, guardar el otro
extremo
                if (lastY != firstY)
                      aux = new Waypoint(regions.get(actualRegion),
                                         regions.get(endRegion),
                                         new Point(lastX, lastY));
                      regions.get(actualRegion).getConnections().add(aux);
                      connections.add(aux);
                      aux = new Waypoint(regions.get(endRegion),
                                         regions.get(actualRegion),
                                         new Point(lastX + 1, lastY));
                      regions.get(endRegion - 1).getConnections().add(aux);
                      connections.add(aux);
                 }//Fin if
                break;
                case 3:
                // Para tres waypoints
                aux = new Waypoint(regions.get(actualRegion),
                                   regions.get(endRegion),
                                   new Point(firstX, firstY));
                regions.get(actualRegion).getConnections().add(aux);
                connections.add(aux);
                aux = new Waypoint(regions.get(endRegion),
                                   regions.get(actualRegion),
                                   new Point(firstX + 1, firstY));
                regions.get(endRegion).getConnections().add(aux);
                connections.add(aux);
                //Si la longitud es mayor que 1, se guarda el otro extremo
                if (lastX != firstX) {
                      aux = new Waypoint(regions.get(actualRegion),
                                         regions.get(endRegion),
                                         new Point(lastX, lastY))
                      regions.get(actualRegion).getConnections().add(aux);
                      connections.add(aux);
                      aux = new Waypoint(regions.get(endRegion),
                                               regions.get(actualRegion),
                                               new Point(lastX + 1,lastY));
                      regions.get(endRegion).getConnections().add(aux);
                      connections.add(aux);}
```

```
//Si existe punto medio
if (lastY > firstY + 2) {
      int X = ((lastX + firstX) / 2);
      int Y = ((lastY + firstY) / 2);
      aux = new Waypoint(regions.get(actualRegion),
                  regions.get(endRegion),
      regions.get(actualRegion).getConnections().add(aux);
      connections.add(aux);
      aux = new Waypoint(regions.get(endRegion),
                   regions.get(actualRegion),
                  new Point(X + 1,Y));
      regions.get(endRegion).getConnections().add(aux);
      connections.add(aux);
}//Fin if
break;
case 4:
// cada 5 puntos
aux = new Waypoint(regions.get(actualRegion),
                  regions.get(endRegion),
                  new Point(firstX, firstY));
regions.get(actualRegion).getConnections().add(aux);
connections.add(aux);
aux = new Waypoint(regions.get(endRegion),
regions.get(actualRegion),
new Point(firstX + 1, firstY));
regions.get(endRegion).getConnections().add(aux);
connections.add(aux);
//Si longitud mayor que 1, añadir el otro extremo
if (lastY != firstY) {
      aux = new Waypoint(regions.get(actualRegion),
                  regions.get(endRegion),
                  new Point(lastX, lastY));
      regions.get(actualRegion).getConnections().add(aux);
      connections.add(aux);
      aux = new Waypoint(regions.get(endRegion),
                        regions.get(actualRegion),
                        new Point(lastX + 1, lastY));
      regions.get(endRegion).getConnections().add(aux);
      connections.add(aux);
}
//Añadir waypoints cada 5 puntos
if (lastY > firstY + 5) {
      int j = firstY + 5;
      while (j < lastY) {
            int Y = j;
            aux = new Waypoint(regions.get(actualRegion),
                              regions.get(endRegion),
                              new Point(firstX, Y));
      regions.get(actualRegion).getConnections()add(aux);
            connections.add(aux);
            aux = new Waypoint(regions.get(endRegion),
                  regions.get(actualRegion - 1),
                  new Point(firstX + 1, Y));
      regions.get(endRegion -).getConnections().add(aux);
            connections.add(aux);
            j += 5;
      }
}// Fin if
break;
```

```
		case 5:
		// cada 3 puntos
		aux = new Waypoint(regions.get(actualRegion),
				regions.get(endRegion),
			new Point(firstX, firstY));
		regions.get(actualRegion).getConnections().add(aux);
		connections.add(aux);
		aux = new Waypoint(regions.get(endRegion),
				regions.get(actualRegion),
				new Point(firstX + 1, firstY));
		regions.get(endRegion - 1).getConnections().add(aux);
		connections.add(aux);
		//Si la apertura es mayor de 1 punto, añadir otro extemo
		if (lastY != firstY) {
			aux = new Waypoint(regions.get(actualRegion),
					regions.get(endRegion),
					new Point(lastX, lastY));
			regions.get(actualRegion).getConnections().add(aux);
			connections.add(aux);
			aux = new Waypoint(regions.get(endRegion),
					regions.get(actualRegion)
					new Point(lastX + 1, lastY));
			regions.get(endRegion).getConnections().add(aux);
			connections.add(aux);
			}
		//Añadir valores cada 3 puntos
		if (lastY > firstY + 3) {
			int j = firstY + 3;
			while (j < lastY) {
				int Y = j;
				aux = new Waypoint(regions.get(actualRegion),
						regions.get(endRegion),
						new Point(firstX, Y));
			regions.get(actualRegion).getConnections().add(aux);
				connections.add(aux);
				aux = new Waypoint(regions.get(endRegion),
						regions.get(actualRegion),
						new Point(firstX + 1, Y));
			regions.get(endRegion).getConnections().add(aux);
				connections.add(aux);
				j += 3;
			}
		}// Fin if
		break;
		}
	}
	// Comprobamos regiones vecinas abajo
	if (zone[x][y + 1] != actualRegion && zone[x][y + 1] != 0) {
		startRegion = actualRegion - 1;
		endRegion = zone[x][y + 1] - 1;
		int firstX = x;
		int firstY = y;
		while (zone[x + 1][y + 1] == endRegion
			&& zone[x + 1][y] == startRegion) {
			x++;
		}
		int lastX = x;
		int lastY = y;
		switch (mode) {
```

```
case 1:
// Para 1 waypoint
if (lastX != firstX) {
      int X = ((lastX + firstX) / 2);
      int Y = ((lastY + firstY) / 2);
      aux = new Waypoint(regions.get(actualRegion),
                         regions.get(endRegion),
                         new Point(X, Y));
      regions.get(actualRegion).getConnections().add(aux);
      connections.add(aux);
      aux = new Waypoint(regions.get(endRegion),
                         regions.get(actualRegion),
                         new Point(X,Y + 1));
      regions.get(endRegion).getConnections().add(aux);
      connections.add(aux);
} else { // Si la longitud es de 1 punto
      aux = new Waypoint(regions.get(actualRegion),
                         regions.get(endRegion - 1),
                         new Point(firstX,firstY));
      regions.get(actualRegion).getConnections().add(aux);
      connections.add(aux);
      aux = new Waypoint(regions.get(endRegion),
                         regions.get(actualRegion),
                         new Point(firstX, firstY + 1));
      regions.get(endRegion).getConnections().add(aux);
      connections.add(aux);
}// Fin else
break;
case 2:
// Para dos waypoints
aux = new Waypoint(regions.get(actualRegion),
                   regions.get(endRegion),
                   new Point(firstX, firstY));
regions.get(actualRegion - 1).getConnections().add(aux);
connections.add(aux);
aux = new Waypoint(regions.get(endRegion - 1),
                   regions.get(actualRegion),
                   new Point(firstX,firstY + 1));
regions.get(endRegion).getConnections().add(aux);
connections.add(aux);
//Si la longitud es mayor de 1 punto, añadir otro extremo
if (lastX != firstX) {
      aux = new Waypoint(regions.get(actualRegion),
                         regions.get(endRegion),
                         new Point(lastX, lastY));
      regions.get(actualRegion ).getConnections().add(aux);
      connections.add(aux);
      aux = new Waypoint(regions.get(endRegion),
                         regions.get(actualRegion),
                         new Point(lastX,lastY + 1));
      regions.get(endRegion).getConnections().add(aux);
      connections.add(aux);
}
break;
case 3:
// Para 3 waypoints
aux = new Waypoint(regions.get(actualRegion,
                   regions.get(endRegion),
                   new Point(firstX, firstY));
regions.get(actualRegion).getConnections().add(aux);
connections.add(aux);
```

```
aux = new Waypoint(regions.get(endRegion),
                   regions.get(actualRegion),
                   new Point(firstX,firstY + 1));
regions.get(endRegion).getConnections().add(aux);
connections.add(aux);
if (lastX != firstX) {
      aux = new Waypoint(regions.get(actualRegion),
                         regions.get(endRegion),
                         new Point(lastX, lastY));
      regions.get(actualRegion).getConnections().add(aux);
      connections.add(aux)
      aux = new Waypoint(regions.get(endRegion),
                   regions.get(actualRegion),
                   new Point(lastX,lastY + 1));
      regions.get(endRegion).getConnections().add(aux);
      connections.add(aux);
}
//Si hay punto medio añadirlo
if (lastX > firstX + 2) {
      int X = ((lastX + firstX) / 2);
      int Y = ((lastY + firstY) / 2);
      aux = new Waypoint(regions.get(actualRegion),
                         regions.get(endRegion),
                         new Point(X, Y));
      regions.get(actualRegion.getConnections().add(aux);
      connections.add(aux);
      aux = new Waypoint(regions.get(endRegion),
                         regions.get(actualRegion),
                         new Point(X,Y + 1));
      regions.get(endRegion).getConnections().add(aux);
      connections.add(aux);
}// Fin if
break;
case 4:
// Cada 5 puntos
aux = new Waypoint(regions.get(actualRegion),
                   regions.get(endRegion),
                   new Point(firstX, firstY));
regions.get(actualRegion - 1).getConnections().add(aux);
connections.add(aux);
aux = new Waypoint(regions.get(endRegion),
                   regions.get(actualRegion),
                   new Point(firstX, firstY + 1));
regions.get(endRegion - 1).getConnections().add(aux);
this.connections.add(aux);
//Añadir el otro extremo si hay
if (lastX != firstX) {
      aux = new Waypoint(regions.get(actualRegion)
                         regions.get(endRegion),
                         new Point(lastX, lastY));
      regions.get(actualRegion).getConnections().add(aux);
      connections.add(aux);
      aux = new Waypoint(regions.get(endRegion),
                         regions.get(actualRegion),
                         new Point(lastX,lastY + 1));
      regions.get(endRegion - 1).getConnections().add(aux);
      connections.add(aux);
}
//Añadir cada 5 puntos
if (lastX > firstX + 5) {
      int j = firstX + 5;
```

```
		while (j < lastX) {
			int X = j;
			aux = new Waypoint(regions.get(actualRegion),
					regions.get(endRegion),
					new Point(X, firstY));
		regions.get(actualRegion).getConnections().add(aux);
			connections.add(aux);
		aux = new Waypoint(regions.get(endRegion),
				regions.get(actualRegion),
				new Point(X,firstY + 1));
		regions.get(endRegion).getConnections().add(aux);
			connections.add(aux);
			j += 5;
		}// fin while
	}// fin if
	break;
	case 5:
	// Un waypoint cada 3 puntos
	aux = new Waypoint(regions.get(actualRegion),
				regions.get(endRegion),
				new Point(firstX, firstY));
	regions.get(actualRegion).getConnections().add(aux);
	connections.add(aux);
	aux = new Waypoint(regions.get(endRegion),
				regions.get(actualRegion),
				new Point(firstX,firstY + 1));
	regions.get(endRegion).getConnections().add(aux);
	connections.add(aux);
	//Si longitud mayor que 1 añadir otro extremo
	if (lastX != firstX) {
		aux = new Waypoint(regions.get(actualRegion),
				regions.get(endRegion),
				new Point(lastX, lastY));
		regions.get(actualRegion).getConnections().add(aux);
		connections.add(aux);
		aux = new Waypoint(regions.get(endRegion),
				regions.get(actualRegion),
				new Point(lastX,lastY + 1));
		regions.get(endRegion).getConnections().add(aux);
		connections.add(aux);
	}
	//Añadir cada 3 puntos
	if (lastX > firstX + 3) {
		int j = firstX + 3;
		while (j < lastX) {
			int X = j;
			aux = new Waypoint(regions.get(actualRegion),
					regions.get(endRegion),
					new Point(X, firstY));
		regions.get(actualRegion).getConnections().add(aux);
			connections.add(aux);
			aux = new Waypoint(regions.get(endRegion),
					regions.get(actualRegion),
					new Point(X,firstY + 1));
		regions.get(endRegion).getConnections().add(aux);
			connections.add(aux);
			j += 3;
		}// fin while
	}// fin if
	break;
	}
```

```
		}
		x++;
		if (zone[x][y] == 0 || zone[x][y] != actualRegion) {
			x = regionX;
			y++;
			if (zone[x][y] == 0)
				x++;
				if (zone[x - 1][y] == actualRegion) {
					while (zone[x - 1][y] == actualRegion) {
						x--;
					}
					regionX = x;
				}
			}
			if (x >= this.width) {
				x = regionX;
				y++;
				if (y > this.heigh) {
					break;
				}
			}
		}
actualRegion++;
} while (actualRegion <= regions.size() - 1);
}
```

10.5. Cálculo de distancias

```
public void calculateDistances() {
int i = 0;
while (i < regions.size()) {
      for (Waypoint way : regions.get(i).getConnections()) {
            for (Waypoint aux : way.getSecond().getConnections()) {
                  if (way.getFirst().equals(aux.getSecond()))
                        continue;
                  distances.add(new Distance(way, aux,
                  EuclideanDistance(way.getPoint(), aux.getPoint())));
            }
      }
i++;
}
}

public double EuclideanDistance(Point start, Point goal) {
      double result;
      int difY = Math.abs(start.y - goal.y);
      int difX = Math.abs(start.x - goal.x);
      result = Math.sqrt(Math.pow(difX, 2) + Math.pow(difY, 2));
      return result;
}
```

10.6. Dijkstra

```
//recorrido de distancias
public void execute(Waypoint source) {
	settledNodes = new HashSet<Waypoint>(); //lista de nodos
añadidos
	unsettledNodes = new HashSet<Waypoint>();//lista de nodos a
comprobar
	distance = new HashMap<Waypoint, Integer>();//tabla de
distancias
	previousNodes = new HashMap<Waypoint, Waypoint>();
	distance.put(source, 0);
	unsettledNodes.add(source);
	while (unsettledNodes.size() > 0) {
		Waypoint node = getMinimum(unsettledNodes);
		settledNodes.add(node);
		unsettledNodes.remove(node);
		findMinimalDistances(node);
	}
}

//elegir el nodo con distancia más corta
private void findMinimalDistances(Waypoint start) {
	List<Waypoint> adjacentNodes = getNeighbors(start);
	for (Waypoint target : adjacentNodes) {
		if (getShortestDistance(target) >
getShortestDistance(start)
					+ getDistance(start, target)) {
			distance.put(target,
				getShortestDistance(start) +
				getDistance(start, target));
			previousNodes.put(target, start);
			unsettledNodes.add(target);
		}
	}
}

//Devuelve el valor de la distancia (coste)
private int getDistance(Waypoint node, Waypoint target) {
	for (Distance way : distances) {
		if (way.getFirst().equals(node) &&
			way.getSecond().equals(target)) {
				return (int) way.getDistance();
		}
	}
}

//Devolver vecinos del nodo dado
private List<Waypoint> getNeighbors(Waypoint node) {
	List<Waypoint> neighbors = new ArrayList<Waypoint>();
	for (Distance way : distances) {
		if (way.getFirst().equals(node) &&
!isSettled(way.getSecond())){
			neighbors.add(way.getSecond());
		}
	}
	return neighbors;
}

//Elige el mínimo de la lista dada
private Waypoint getMinimum(Set<Waypoint> nodes) {
	Waypoint minimum = null;
```

```
		for (Waypoint waypoint : nodes) {
			if (minimum == null) {
				minimum = waypoint;
			} else {
				if (getShortestDistance(waypoint) <
					getShortestDistance(minimum)) {
					minimum = waypoint;
				}
			}
		}
		return minimum;
	}

	//Si el nodo ya está en la lista de nodos añadidos
	private boolean isSettled(Waypoint node) {
		return settledNodes.contains(node);
	}

	//Devuelve la distancia más corta al nodo dado
	private int getShortestDistance(Waypoint destination) {
		Integer d = distance.get(destination);
		return d;
	}

	//Devuelve el recorrido hasta el nodo dado
	public LinkedList<Waypoint> getPath(Waypoint goal) {
		LinkedList<Waypoint> path = new LinkedList<Waypoint>();
		Waypoint step = goal;
		//comprueba si existe
		if (previousNodes.get(step) == null) {
				return null;
			}
			path.add(step);
			while (previousNodes.get(step) != null) {
				step = previousNodes.get(step);
				path.add(step);
			}
			// Put it into the correct order
			Collections.reverse(path);
			return path;
		}
	}
```

10.7. A estrella

```
//Función con algoritmo A*
public Result search(Point start, Point end) {
	int expandedNodes = 0;
	int generatedNodes = 1;
	PriorityQueue<Node> openList = new PriorityQueue<>();
	// Lista abierta
	HashSet<Node> closeList = new HashSet<>(); // Lista cerrada
	HashMap<Point, Double> costList = new HashMap<>();
	long time = System.currentTimeMillis();
	//Añadir estado inicial
	openList.add(new Node(start, 0,
			this.heuristic(start, end), 0));
	costList.put(start, 0.0);
	while (!openList.isEmpty()) {
		Node actualState = openList.poll();
		//Si se alcanza la solución
		if (actualState.getPosition().equals(end)) {
			Object[] path = this.getPath(actualState);
			return new Result((List<Point>) path[0],
				generatedNodes, expandedNodes,
				(int) path[1],System.currentTimeMillis()-time);
		}
	closeList.add(actualState);
	expandedNodes++;
	//Generar sucesores
	List<Successor> successors =
			this.successors(actualState.getPosition());
	for (Successor successor: successors) {
	//Añadirlo a la lista si no está repetido
		if (!closeList.contains(successor)) {
			double newg = actualState.getG()
					+ successor.getCost();
			if (!costList.containsKey(successor.getPoint())
				|| costList.get(successor.getPoint()) > newg){
				costList.put(successor.getPoint(), newg);
				for (Iterator<Node>it=openList.iterator()
					;it.hasNext();) {
					if (it.next().getPosition()
						.equals(successor.getPoint()))
							it.remove();
					}
					Node newNode = new Node
					(successor.getPoint(),
					actualState, newg,
					this.heuristic(successor.getPoint(),
						end), successor.getCost());
					openList.add(newNode);
					generatedNodes++;
				}
			}
		}
	}

	return new Result(null, generatedNodes, expandedNodes, 0,
	System.currentTimeMillis() - time);
}
```

```java
//Función para generar sucesores
public List<Successor> successors(Point actualState) {
	List<Successor> LSuccessor = new ArrayList<>();
	Point punto;
	for(int i =-1;i<2;i++){
		for(int j= -1 ;j<2;j++){
			if(i==0 && j==0)continue;
				if((int)actualState.getY()+i>0
				||(int)actualState.getX()+j>0
				||(int)actualState.getX()+i<map.getWidth()
				||(int)actualState.getY()+j<map.getHeigh()){
					if(map.getMap()
					[(int)actualState.getX()+i]
					[(int)actualState.getY()+j] != 0){
						punto=new
						Point((int)actualState.getX()+i,
						(int)actualState.getY()+j);
					LSuccessor.add(new Successor(punto));
				}
			}
		}
	}
	return LSuccessor;
}

//Función heurística
public double heuristic(Point state, Point goalState) {
	int dif_x=Math.abs(state.x-goalState.x);
	int dif_y=Math.abs(state.y-goalState.y);
	double dist=Math.min(dif_x, dif_y)
			*Math.sqrt(32)
			+(Math.max(dif_x, dif_y)
			-Math.min(dif_x, dif_y))*4;
	return dist;
}

//Devolver camino
 private Object[] getPath(Node goalState) {
	int cost = 0;
	List<Point> path = new ArrayList<>();
	Node state = goalState;
	while (state != null) {
		path.add(0, state.getPosition());
		cost += state.getCost();
		state = state.getFromTile();
	}
	return new Object[]{path, cost};
}
```

10.8. Manual de usuario

Se ha utilizado el entorno Eclipse para realizar la programación y el compilado. Este entorno puede obtenerse de forma gratuita en su página web [11].

Para ejecutar el programa se reciben por parámetro el archivo que contiene el mapa, los dos puntos entre los que se quiere encontrar la solución y el modo de búsqueda a ejecutar codificado numéricamente. En las opciones de ejecución habrá que escribir los siguientes argumentos como en la ilustración 31.

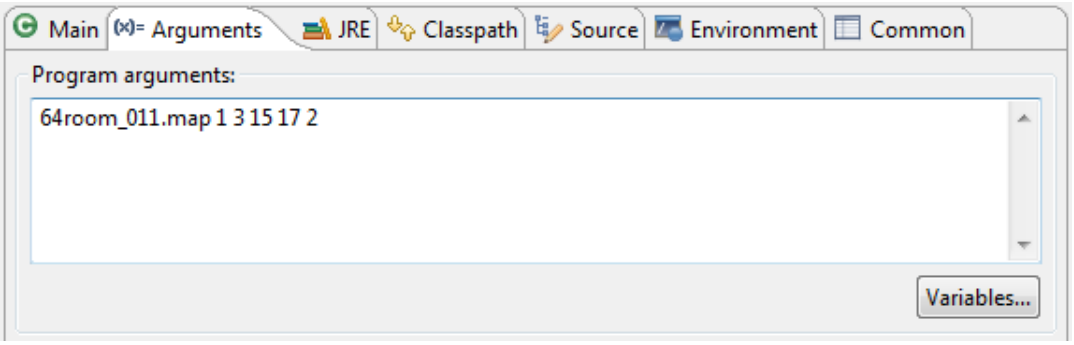


Ilustración 31: pantalla de configuración de ejecución

Donde el primer argumento es el nombre del archivo, el segundo y tercero el primer punto en forma (x, y), el cuarto y quinto el punto destino y el último un número que corresponde al tipo de búsqueda con el código:

- 1: búsqueda con 1 waypoint en el punto central.
- 2: búsqueda con waypoints en los extremos.
- 3: búsqueda con waypoints en los extremos y el punto central.
- 4: búsqueda con waypoints en los extremos y cada 3 puntos.
- 5: búsqueda con waypoints en los extremos y cada 5 puntos.

Una vez se inicia la ejecución el mapa se lee y se genera un documento en la carpeta que contiene el programa llamado *RegionMap.txt* con las regiones que se han generado en el mapa como el de la ilustración 32.

```
1  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
2  @1111111111111111@2222222222@
3  @1111111111111111@2222222222@
4  @1111111111111111@2222222222@
5  @@@3333333333333333@2222222222@
6  @@@3333333333333333222222222222@
7  @3333333333333333@2222222222@
8  @3333333333333333@2222222222@
9  @3333333333333333@2222222222@
10 @@@44444@@@@@@5@@@@@2222222222@
11 @444444@5555555555@2222222222@
12 @444444@5555555555@2222222222@
13 @444444@5555555555@2222222222@
14 @4444445555555555@2222222222@
15 @444444@5555555555@2222222222@
16 @@@66@@@5555555555@2222222222@
17 @666666@@555@@@@@@2222222222@
18 @6666666666666666@2222222222@
19 @66666666666666662222222222@
20 @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Ilustración 32: ejemplo de mapa procesado en regiones

Y tras la búsqueda un documento en la misma carpeta llamado *MapaBusqueda.txt* con el camino generado como en la ilustración 33.

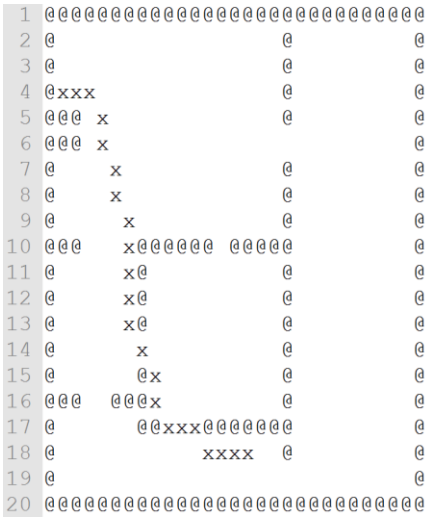


Ilustración 33: ejemplo de camino de búsqueda

Y se imprime por pantalla un resumen de la búsqueda como la ilustración 34.

```
Regiones 7
Gateways 24
Generado mapa en: 4
Tiempo Dijkstra: 3
Tiempo A*: 4
**********************************************************
Resumen del proceso de búsqueda
Solución encontrada con coste 91.0
Estado inicial: [1,3]
Estado final: [15,17]
Longitud del camino: 22
Tiempo de ejecución: 7 milisegundos
**********************************************************
```

Ilustración 34: ejemplo de resumen de búsqueda

Estos datos también se guardan en un archivo llamado *Busqueda.txt*.

10.9. Mapas

10.9.1. Mazmorras

Alto: 512 Ancho: 512

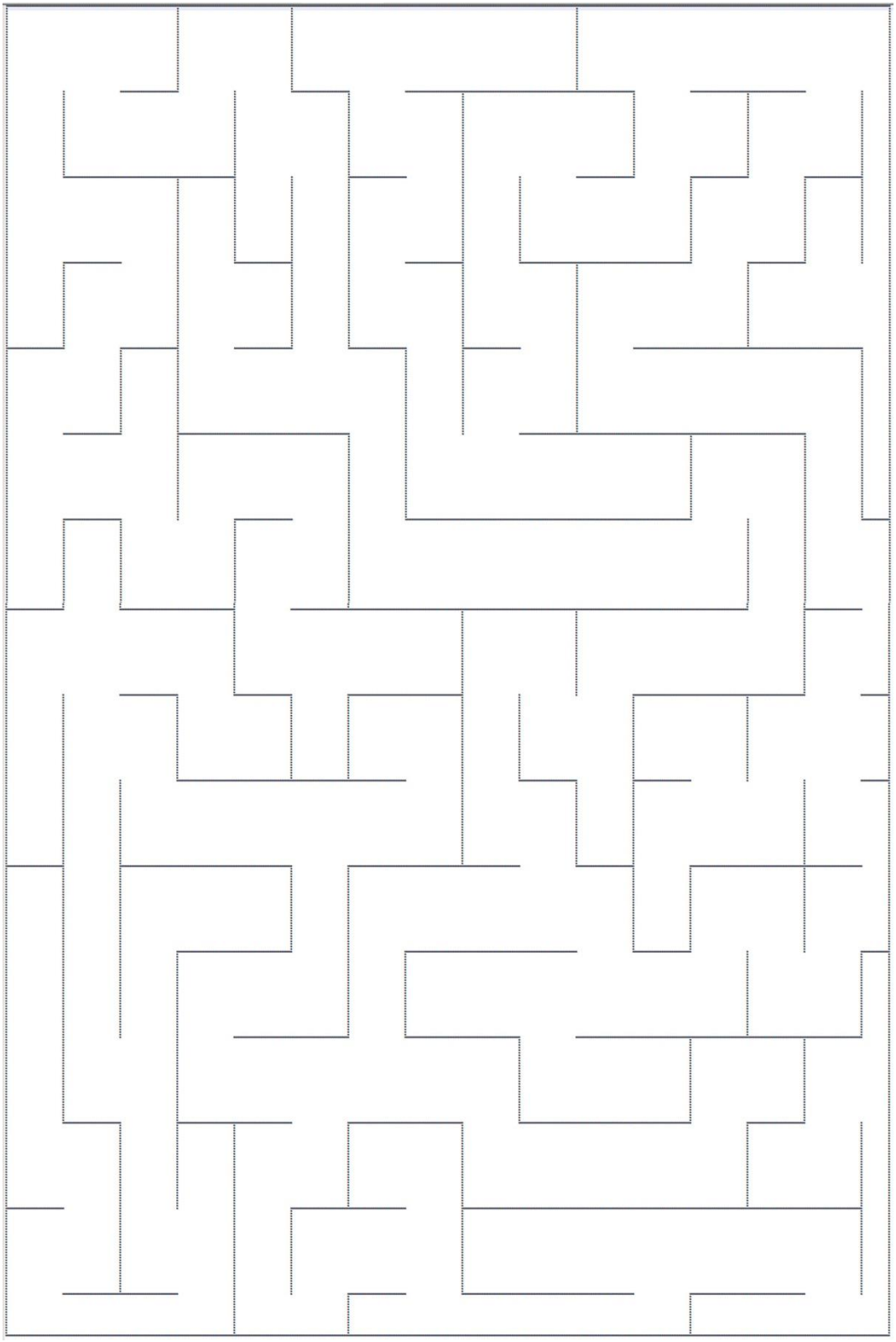


Ilustración 35: mapa mazmorras

10.9.2. Dragon Age Origins

Alto: 261 Ancho: 257

Ilustración 36: mapa Dragon Age Origins

10.9.3. Baldurs Gate

Alto: 512 Ancho: 512

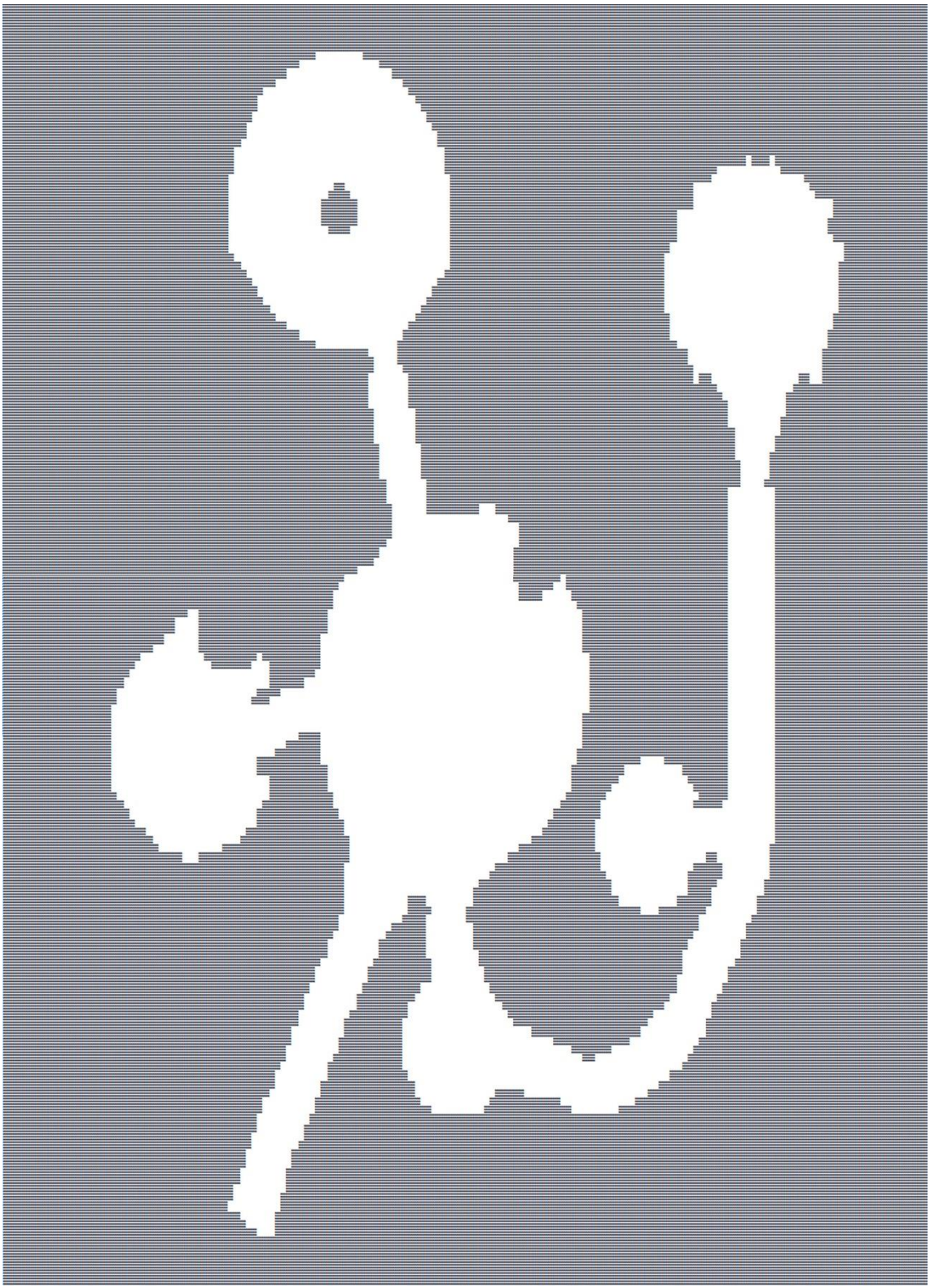


Ilustración 37: mapa del juego Baldurs Gate

11. Referencias

[1] Grid-Based Path Planning Competition (GPPC): http://movingai.com/GPPC/

[2]Tenis for Two: https://es.wikipedia.org/wiki/Tennis_for_Two

[3] OXO : https://es.wikipedia.org/wiki/OXO

[4] Atari: https://www.atari.com/

[5] Algoritmo A*: https://prezi.com/uagchtutnyhx/busqueda-por-el-metodo-algoritmico-a/

[6] Alvaro H., Salas S., (2008) *Acerca del Algoritmo de Dijkstra.* Departamento de matemáticas, Universidad de Caldas. De ttps://arxiv.org/pdf/0810.0075.pdf

[7] Moving AI Lab: http://movingai.com/

[8] Parra, Á, Torralba, Á y Linares, C.(2012) *Precomputed-Direction Heuristics for Suboptimal Grid-Based Path-Finding* .Proceddings of The Fifth Annual Symposium on Combinatorial Search 211-212.

[9] Mapas Moving AI: http://movingai.com/benchmarks/

[10] Björnsson, Y. y Halldórsson, K. (2006) *Improved Heuristics for Optimal Pathfinding on Game Maps.* Proceedings of the Second Artificial Intelligence and Interactive Digital Entertainment Conference 9-14. http://www.aaai.org/Library/AIIDE/2006/aiide06-006.php

[11] Eclipse: https://eclipse.org/