



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Búsqueda de caminos en mapas hexagonales

Proyecto Final de Carrera

Ingeniería Técnica en Informática de Sistemas

Autor: Saúl Esteban Martínez

Director: Mario Gómez Martínez

27/09/2013

Resumen

El presente proyecto parte de un doble objetivo, por una parte, implementar una librería capaz de resolver problemas de búsqueda de caminos en un mapa hexagonal y, por otra, desarrollar una serie de herramientas que sean capaces de evaluar diferentes algoritmos de búsqueda de caminos. De esta manera, estas herramientas permitirán la realización de pruebas y análisis de resultados utilizando los componentes de la librería.

Palabras clave: grafo, arista, pathfinding, algoritmo, A*, mapa hexagonal, búsqueda, caminos, búsqueda de caminos en grafos, búsqueda de caminos en mapas hexagonales, comparación de algoritmos, benchmark, estrategia.



Tabla de contenidos

1. Introducción	10
2. Análisis del problema.....	12
2.1. Entorno.....	12
2.2. Problema.....	14
2.3. Descripción del entorno como un grafo.....	16
3. Diseño de la librería de búsqueda.....	18
3.1. Elementos de la librería	18
3.2. Estructura de la librería	19
4. Algoritmos de búsqueda.....	25
4.1. AStar.....	25
4.2. BeamSearch	28
4.3. BidirectionalSearch.....	30
5. Diseño de la aplicación de benchmark	36
5.1. Vistas	36
5.2. Controladores.....	39
6. Implementación	45
6.1. Entorno de desarrollo.....	45
6.2. Estructura del proyecto	45
6.2.1. Elementos externos utilizados.....	45
6.2.2. Estructuras de datos.....	46
6.3. Interfaz gráfica y casos de uso	47
6.3.1. Barra de menú	48
6.3.2. Barra de herramientas	50
6.3.3. Vista del comparador	50
6.3.4. Vista de benchmark.....	56
7. Análisis de resultados.....	59
8. Conclusión y opinión personal.....	64
9. Referencias.....	65



Tabla de ilustraciones

Ilustración 1 - Grafo bidireccional.....	12
Ilustración 2 - Nodo y sus adyacentes.....	13
Ilustración 3 - Coste de entrada a campos.....	14
Ilustración 4 - Coste de entrada a bosque.....	14
Ilustración 5 - Camino esquivando un obstáculo representado por unas marismas.....	16
Ilustración 6 - Esquema de nodos y aristas.....	16
Ilustración 7 - Esquema de coste entre nodos.....	17
Ilustración 8 - Estructura de la librería.....	21
Ilustración 9 - Estructura de los algoritmos (detalle).....	22
Ilustración 10 - Estructura Nodo-Camino.....	23
Ilustración 11 - Funciones de coste y heurísticas.....	24
Ilustración 12 - Estructura de la aplicación.....	36
Ilustración 13 - Estructura de las vistas.....	39
Ilustración 14 - Estructura de los controladores.....	43
Ilustración 15 - Estructura del generador de problemas y las soluciones.....	44
Ilustración 16 - Conexión entre los nodos y el entorno.....	47
Ilustración 17 - Interfaz del comparador.....	48
Ilustración 18 - Barra de menú.....	48
Ilustración 19 - Submenú "File".....	49
Ilustración 20 - Submenú "View".....	50
Ilustración 21 - Submenú "Tools".....	50
Ilustración 22 - Barra de herramientas.....	50
Ilustración 23 - Panel de configuraciones.....	51
Ilustración 24 - Panel de configuración mutua.....	51
Ilustración 25 - Capa de terreno.....	52
Ilustración 26 - Capa de rejilla.....	53
Ilustración 27 - Capa de búsqueda.....	54
Ilustración 28 - Capa de flechas.....	55
Ilustración 29 - Panel de estadísticas.....	56
Ilustración 30 - Panel de creación de problemas.....	57
Ilustración 31 - Panel de selección de algoritmos.....	58
Ilustración 32 - Lista de unidades y botón de ejecución.....	58
Ilustración 33 - "AStar" vs "BeamSearch" (buen resultado).....	59
Ilustración 34 - "AStar" vs "BeamSearch" (mal resultado).....	60
Ilustración 35 - "AStar" vs "Bidireccional".....	60

1. Introducción

El contexto en el que surge este proyecto es la existencia de un juego de estrategia por turnos simultáneos llamado “ARES”, este juego está siendo desarrollado por el director de este proyecto y necesitaba una librería que permitiera realizar búsquedas de caminos. De esta necesidad surge la idea de crear una librería que fuera capaz de realizar dicha tarea para mapas hexagonales (los que utiliza el juego).

Además, para asegurar el buen funcionamiento de la librería surge la idea de crear una aplicación a medida para hacer pruebas, visualizar resultados y realizar análisis sobre el funcionamiento de los algoritmos que se implementaran.

La búsqueda de caminos es un campo de la informática muy importante en sectores como el de los videojuegos, donde muchas veces se encuentra la problemática del cálculo del camino más corto para llegar de un punto a otro. Dentro de este campo se han desarrollado diversos métodos o algoritmos que permiten la obtención del camino en distintas situaciones, en distintos tipos de representaciones de planos o teniendo en cuenta diferentes factores, como los obstáculos.

Este campo de investigación se basa enormemente en el algoritmo de “Dijkstra”, el cual permite encontrar el camino más corto en un grafo buscando, a partir de un nodo inicial (un nodo es la representación de un punto en el mapa), el siguiente nodo más cercano hasta hallar el nodo objetivo. El problema surge cuando el sistema de representación es más complejo (por ejemplo, un mapa en tres dimensiones) o cuando se quiere minimizar el tiempo que se emplea en encontrar el camino. Con este último fin surgió una variante del algoritmo de “Dijkstra” llamada “A*”.

El algoritmo “A*” consigue una ejecución más rápida y eficiente mediante el uso de heurísticas, es decir, estimaciones del tiempo o coste necesario que lleva llegar desde un punto a otro, dando así prioridad a los nodos a los que más cuesta llegar. Utiliza

una función de coste que es el resultado de la suma de dos funciones, la función que calcula el coste que comprende desde el inicio hasta el nodo que está analizando, y la función que hace la estimación desde este último nodo hasta el nodo destino.

El tipo de representación con el que se ha desarrollado el algoritmo es un tablero de casillas hexagonales que poseen un coste de entrada y salida para cada una de las seis direcciones, así las heurísticas se pueden hacer a partir de las coordenadas de cada casilla y se pueden sumar al coste que se va calculando según se avanza en la búsqueda.

Por todos estos motivos es un algoritmo muy utilizado en la búsqueda de caminos y también ha sido utilizado para la implementación del motor de búsqueda de caminos de este proyecto. Además, se han desarrollado dos variantes del algoritmo “A*” con el fin de conseguir una mayor eficiencia y analizar de manera detallada el comportamiento de todos los algoritmos, además de proporcionar la posibilidad de comparar diferentes tipos de algoritmos o parámetros de uno solo. También, gracias al comparador visual que se ha desarrollado, se facilita la comprensión del funcionamiento de los algoritmos al tiempo que se le da una utilidad didáctica al proyecto, ya que ilustra varios conceptos sobre la búsqueda de caminos en grafos.

Las pruebas anteriormente mencionadas se llevan a cabo a través de la interfaz mediante el comparador visual de algoritmos y la herramienta “benchmark” que se explican más adelante, ambas generan unas estadísticas que son las que permiten establecer conclusiones sobre el funcionamiento y el rendimiento de los algoritmos.

2. Análisis del problema

En esta sección se explica a fondo el entorno que ha servido de base para desarrollar el motor de búsqueda y los problemas que se han presentado a la hora de desarrollarlo.

2.1. Entorno

Para entender bien el entorno en el que se ha desarrollado el problema es necesario conocer a fondo qué es un grafo. Un grafo es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan terminales y las aristas representan conexiones (las cuales, a su vez, pueden ser cables o conexiones inalámbricas).

Las aristas pueden llevar, o no, un coste asociado que defina cuánto cuesta llegar de un nodo a otro, así el camino más rápido no será siempre el más corto. Este tipo de grafo se llama grafo ponderado, y en la siguiente imagen se puede ver un claro ejemplo.

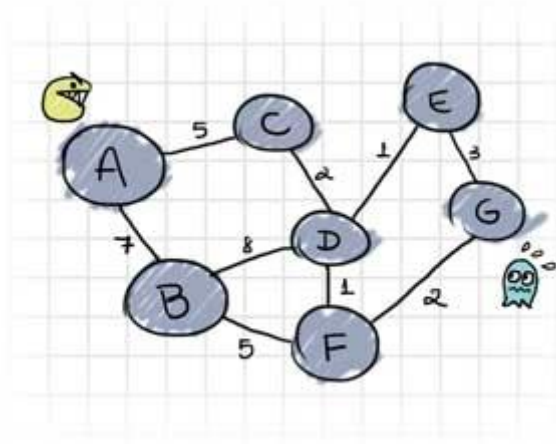


Ilustración 1 - Grafo bidireccional

El entorno en el que se ha desarrollado el motor de búsqueda es un tablero de casillas hexagonales que representan diferentes tipos de terreno. De este modo identificamos las casillas del tablero como vértices o nodos de un grafo y, las aristas de los hexágonos que las forman, como la relación de adyacencia entre los nodos (aristas o arcos). Las aristas de estos nodos son bidireccionales, es decir, se puede ir desde un nodo 'a' a un nodo 'b' y desde el mismo nodo 'b' se puede ir al nodo 'a'.

Los nodos que están unidos por un borde o arista al nodo que representa a una casilla específica son llamados sus adyacentes (o vecinos), y son el modo de expandir la lista de casillas candidatas a formar parte del camino.



Ilustración 2 - Nodo y sus adyacentes

Para llegar de un nodo a otro se tienen en cuenta diversos factores que alteran el coste de desplazamiento, el primero es el terreno, dependiendo del tipo de terreno de una casilla específica supondrá más o menos coste entrar en ella, también afecta el tipo de terreno que tenga la casilla desde la que nos movemos, por tanto, será diferente entrar en una casilla desde una de las seis direcciones posibles que de otra. A continuación se puede ver un ejemplo de un caso en el que se entra en una casilla que representa campos y otro en el que se entra en una casilla que representa bosque.



Ilustración 3 - Coste de entrada a campos



Ilustración 4 - Coste de entrada a bosque

Otro factor importante a la hora de obtener el coste de movimiento entre nodos es el tipo de movimiento de la unidad que quiera cruzar de una casilla a otra, por ejemplo, no supondrá el mismo coste cruzar para una unidad motorizada que para unidad de a pie. En algunos casos el tipo de movimiento puede provocar que el llegar de una casilla a otra sea imposible, ya que, por ejemplo, una unidad motorizada no puede pasar de una casilla que representa tierra a una que representa un río y una unidad anfibia sí.

2.2. Problema

La problemática que supone desarrollar un motor de búsqueda de caminos es asegurarse de que el método es eficiente pero sobre

todo preciso (es muy importante conseguir el mejor camino según las preferencias de búsqueda, por ejemplo, si se busca el más corto debe encontrarse el que tenga menor número de nodos), para esto, después de analizar cada nodo, se debe hacer una buena elección del siguiente que se va a analizar, por eso es importante hacer una buena estimación del coste para llegar desde los adyacentes del nodo analizado al nodo destino, ya que este es el principal criterio que se va a utilizar para hacer dicha elección y evitar casillas a las que cueste mucho llegar o que sean inaccesibles. Si la estimación no es buena puede dar lugar a que la búsqueda sea demasiado lenta, ya que se analizarían nodos innecesarios, o incluso a encontrar un camino que no sea el más corto.

También existe la problemática de los obstáculos, formados por uno o más nodos que son inaccesibles, ya que si el tamaño de un obstáculo es lo bastante extenso puede desviar la búsqueda y multiplicar el tiempo de ésta. En el escenario de este proyecto los obstáculos pueden resultar ser terrenos imposibles de cruzar en ciertas circunstancias, como montañas, ríos o mares. Una opción a la hora de considerar el funcionamiento del algoritmo A* cuando existen muchos obstáculos, o muy problemáticos, es relajar el peso de la heurística en las estimaciones para que la búsqueda se expanda en menor grado, lo que supone la obtención de un camino de manera más rápida pero no asegura que éste sea óptimo, sino que su coste final será más grande.





Ilustración 5 - Camino esquivando un obstáculo representado por unas marismas

2.3. Descripción del entorno como un grafo

En esta sección se expone la representación del entorno como un grafo. Las casillas hexagonales son nodos o vértices, éstos están unidos por un par de aristas por cada lado, como se representa en el siguiente esquema.

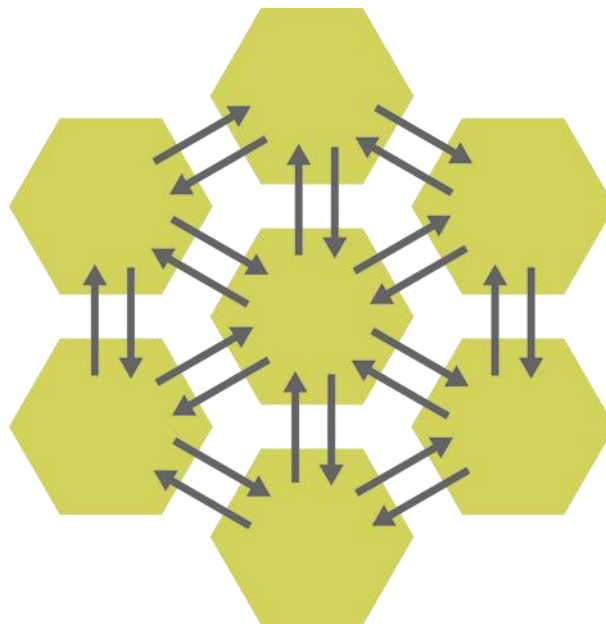


Ilustración 6 - Esquema de nodos y aristas

El peso o coste de cada arista depende de factores estáticos, como el tipo de terreno que representa la casilla en cuestión y sus adyacentes, y también de factores dinámicos, como el tipo de movimiento que se esté utilizando. Por ejemplo, en la siguiente imagen se muestra el coste para moverse entre dos casillas, éste podría ser mayor o menor dependiendo del tipo de movimiento que se estuviera usando.

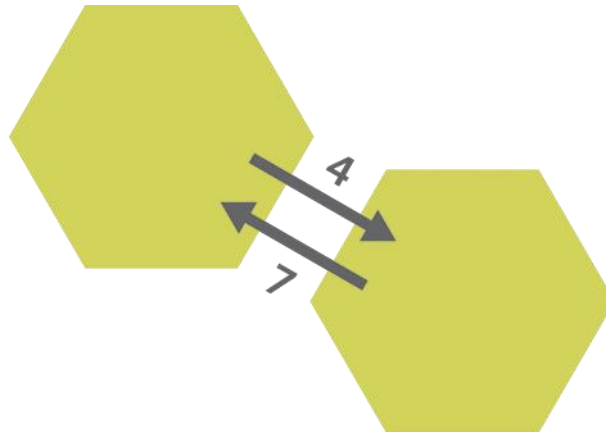


Ilustración 7 - Esquema de coste entre nodos

3. Diseño de la librería de búsqueda

En esta sección se describe la estructura del módulo de búsqueda de caminos y su funcionamiento independiente como librería, que utilizan los tres algoritmos que se han desarrollado durante el proyecto. A continuación se presentan los elementos utilizados y se expone la estructura de la librería, explicando cada uno de sus componentes excepto los algoritmos, que se explican en la siguiente sección.

3.1. Elementos de la librería

- a) **Nodo (Node):** Representa una casilla del tablero, además de contener la información de ésta, contiene la información correspondiente al análisis realizado por el algoritmo que lo crea, como el nodo anterior a él en el camino, el siguiente, la dirección hacia la que se encuentra este último, el coste para llegar a su casilla y la estimación de llegada a la casilla destino seleccionada.
- b) **Camino (Path):** Este objeto representa el recorrido por la unidad y es el objetivo de los algoritmos de búsqueda. El propio objeto no contiene todos los componentes del camino, sino que contiene solamente el nodo inicial y el nodo final de éste.
- c) **Camino extendido (ExtendedPath):** Esta extensión del objeto "Path" se creó con la intención de almacenar las colecciones de nodos analizados, o puestos en cola para analizar, para que puedan ser representados gráficamente en el tablero. Sólo se utiliza en el módulo de análisis de búsqueda de caminos, ya que hace un uso de memoria al almacenar los nodos y, ya que no siempre se va a hacer una representación gráfica del proceso de búsqueda, este uso de memoria no es siempre necesario.
- d) **Nodo inverso (InvertedNode):** Este objeto es una extensión del objeto nodo, su creación fue necesaria para la implementación

de uno de los algoritmos que se han realizado, llamado “BidirectionalSearch”, ya que realiza una búsqueda inversa y, por tanto, requiere este tipo de nodo que cuyos constructores y métodos funcionan de manera diferente y se explican más adelante.

- e) Función de coste (CostFunction): Se trata del objeto encargado de calcular el coste real que supone llegar a una casilla desde una de sus adyacentes.
- f) Heurística (Heuristic): Este tipo de objeto es el encargado de calcular el coste estimado desde una casilla a otra, lo que permitirá elegir las mejores casillas en cada momento.

3.2. Estructura de la librería

- a) Pathfinder: Es la interfaz que posee los métodos para buscar un camino estándar o uno extendido, y para asignar una heurística o una función de coste a la clase que lo implementa.
- b) AbstractPathfinder: Esta es la clase a la que extienden los tres algoritmos y la única que implementa la interfaz “Pathfinder”, por tanto posee los métodos “getters” y “setters” para la heurística y la función de coste que se usan en los métodos “findPath” y “getExtendedPath”.
- c) Path: La clase “Path” posee dos objetos “Node” que representan el inicio y el fin del camino, además incluye el número de nodos que lo componen. Su constructor recibe los dos nodos y calcula su longitud.
- d) ExtendedPath: Esta clase añade la colección de nodos del “openSet” y la del “closedSet”, además del número total de nodos que se han visitado en su búsqueda.
- e) Node: Contiene un objeto de tipo “Tile” (una casilla), el nodo previo a él, el siguiente, la dirección desde la que se llega al nodo, el coste total para llegar a él, el coste estimado para llegar



desde él a la casilla destino, y la suma de ambos. Además de los métodos para hacer comparaciones entre instancias de esta clase, posee un método para cambiar su nodo previo, que se usa en caso de encontrar una sección de camino cuyo coste sea menor a otra que se haya encontrado previamente.

- f) **InvertedNode**: Esta clase que extiende a “Node” utiliza la dirección de manera inversa, pero su constructor solo varía con el de su clase materna en que el nodo siguiente es de esta nueva clase. Además sobrescribe el método “getNext” para que devuelva un nodo de tipo “InvertedNode” y añade un método para cambiar el nodo siguiente.
- g) **CostFunction**: Se trata de la interfaz que implementan las clases que calculan el coste que supone llegar a un nodo desde uno de sus adyacentes. Estas clases implementan la función “getCost”, que devolverá dicho coste.
- h) **TerrainCostFunctions**: Esta clase numerable contiene dos opciones que devuelven un coste cada una, la primera es “FASTEST” la cual devuelve el coste que le supone a un tipo de unidad entrar en una casilla desde una de sus adyacentes (para esto se usa la dirección desde la que se quiere llegar); la segunda es “SHORTEST”, ésta comprueba si el coste que se ha obtenido (de la misma manera que la opción anterior) es suficientemente alto para considerarse inaccesible, en este caso se devuelve el valor calculado y, si es accesible se devuelve siempre el valor 1, para que el coste para moverse entre casillas sea siempre el mismo y se encuentre el camino más corto, sin tener en cuenta el coste de éste.
- i) **UnitCostFunctions**: Esta clase tiene el mismo funcionamiento que la anterior pero con una diferencia, en lugar de obtener simplemente el coste para llegar a una casilla, obtiene este coste teniendo en cuenta circunstancias como fenómenos climatológicos, presencia enemiga e incluso la densidad de tráfico en las carreteras.

- j) **Heuristic:** Es la interfaz que implementan todos los tipos de heurísticas que se creen. Posee el método “getCost”, que acepta como argumentos una casilla origen, otra casilla destino y una unidad, y será el que haga la estimación del coste para llegar desde la casilla destino a la casilla origen.
- k) **MinimumDistance:** Esta clase que implementa la interfaz “Heuristic” sobrecarga el método “getCost” utilizando una instancia de la clase “DistanceCalculator”, cuyo funcionamiento se explica más adelante.
- l) **EnhacedMinimumDistance:** Se trata de otro tipo de heurística que, en la función “getCost”, trata de mejorar el valor obtenido por el “DistanceCalculator”.
- m) **DistanceCalculator:** Esta clase numerable contiene diversas opciones para realizar el cálculo de la distancia entre dos casillas, cada opción utiliza un método distinto.

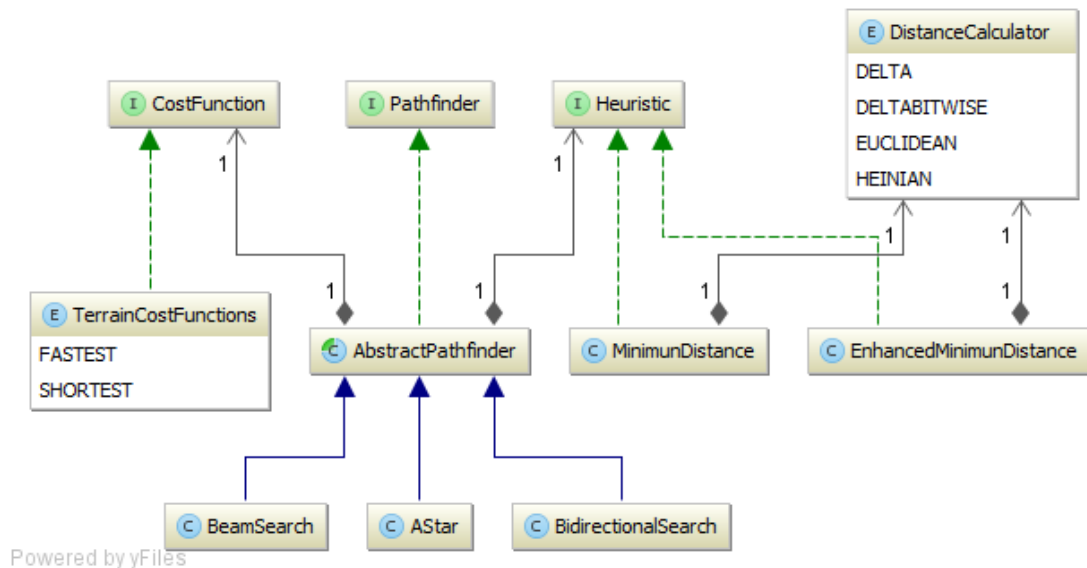


Ilustración 8 - Estructura de la librería



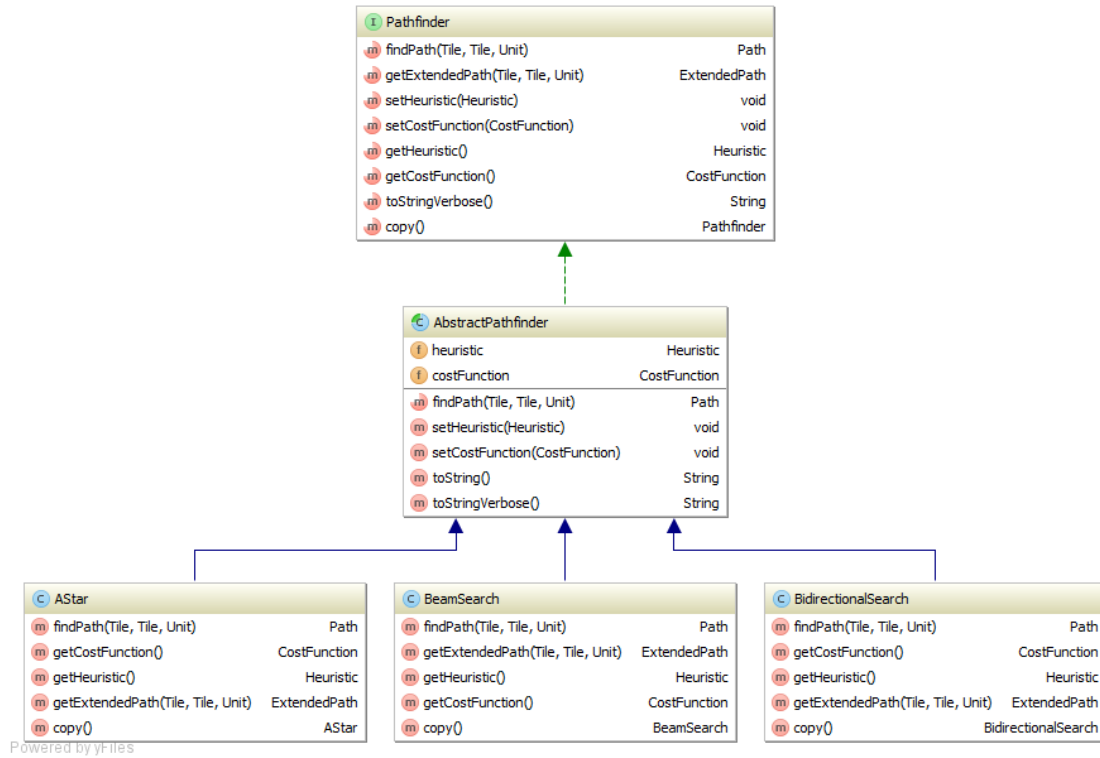
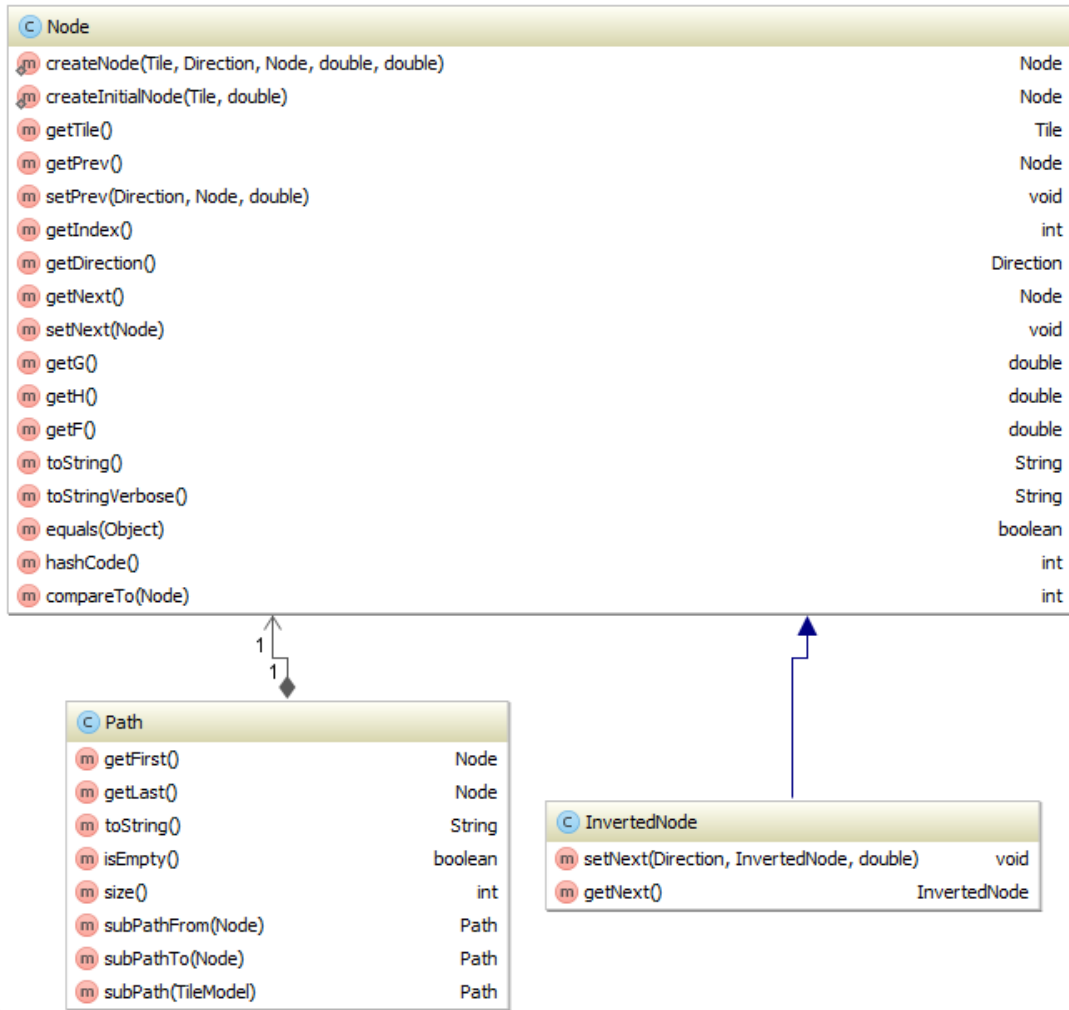


Ilustración 9 - Estructura de los algoritmos (detalle)



Powered by yFiles

Ilustración 10 - Estructura Nodo-Camino



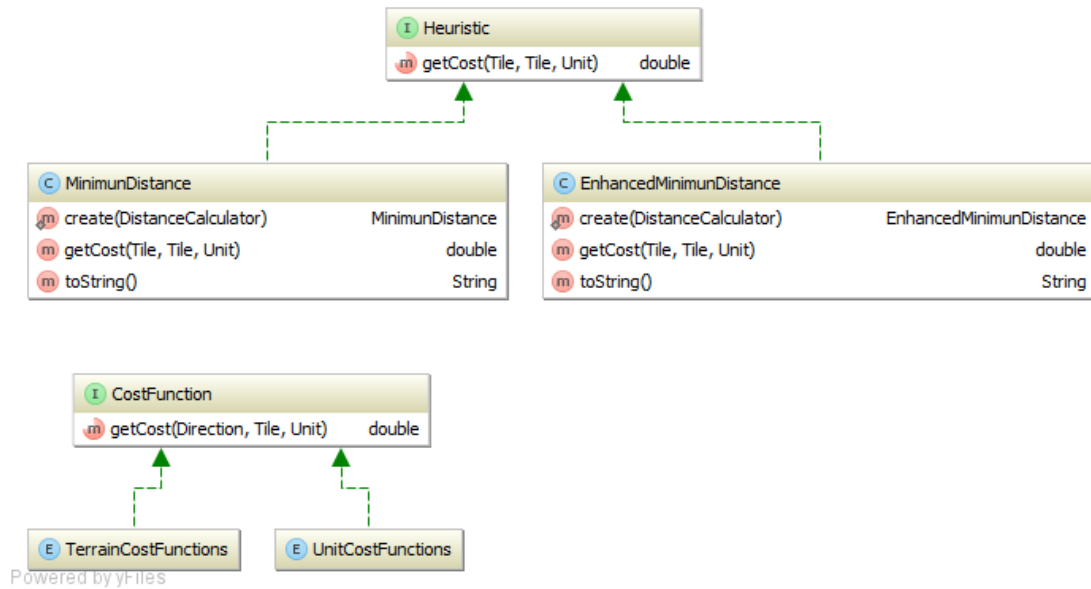


Ilustración 11 - Funciones de coste y heurísticas

4. Algoritmos de búsqueda

A continuación se explica el funcionamiento de los tres algoritmos que se han implementado para el módulo. Todos ellos poseen dos colecciones de nodos, una representa el “OpenSet” (nodos en cola para analizar) y la otra el “ClosedSet” (nodos analizados). La clase interna OpenSet posee una cola de prioridad que representa la primera de las colecciones, por otro lado, el ClosedSet se representa mediante un “HashMap”. Los algoritmos analiza inicialmente el nodo correspondiente a la casilla de inicio del camino, después van analizando los nodos que se van añadiendo al OpenSet según la estimación de coste que se realiza hasta llegar a la casilla destino y formar el camino.

Además, todos implementan la interfaz “Pathfinder” y, por lo tanto, tienen que implementar los métodos “getPath” y “getExtendedPath”, que ejecutan el algoritmo y reciben como argumentos la casilla inicial, la casilla destino y la unidad que va a utilizar el camino. Sólo se diferencian en que el segundo devuelve un objeto del tipo “ExtendedPath”, cuyas características se han explicado en la sección anterior.

4.1. AStar

El algoritmo empieza inicializando el OpenSet y el ClosedSet, después crea el nodo que servirá de punto de partida con la casilla de inicio, lo añade al OpenSet y comienza el bucle que analiza todos los nodos necesarios para encontrar el camino, el cual no parará hasta que lo encuentre o hasta que el OpenSet se quede vacío.

En cada iteración del bucle se recoge el primer nodo de la cola de prioridad del OpenSet (el cual tendrá el menor coste total en toda la colección), y se mueve al ClosedSet. Acto seguido se comprueba si la casilla a la que corresponde dicho nodo es la casilla destino, si es así se crea un “Path” a partir del nodo actual y el inicial, en caso contrario se inicia un bucle que analizará los nodos adyacentes al nodo actual.



En este bucle se recorren todos los nodos que se encuentran en una posición inmediata a la del nodo actual, al estar utilizando un mapa de casillas hexagonales éstos siempre serán seis (excepto cuando se esté analizando una casilla del borde del mapa). Para cada uno de los adyacentes se calcula el coste que supone llegar a él desde el actual, si resulta que el adyacente es inalcanzable desde el actual no se le tiene en cuenta y se pasa directamente a analizar el siguiente adyacente, si no es así, al coste calculado se le añade el coste real del nodo actual (este es el coste que supone llegar desde el nodo inicial al actual), así se obtiene el coste real para llegar al adyacente. Después se comprueba si el nodo adyacente pertenece al ClosedSet y si el coste real que se acaba de calcular es mayor o igual al que tenía anteriormente, si es así se pasa directamente a analizar el siguiente adyacente, si no, el análisis continúa, el siguiente paso es comprobar si el adyacente pertenece al OpenSet y si el coste real que hemos calculado es inferior al que tenía previamente, si se cumple esta condición significa que se ha encontrado un camino mejor para llegar al adyacente, así que se le actualiza el coste y se le asigna el nodo actual como nodo previo a él. Si el nodo no pertenece a ninguna de las colecciones es que no existe, así que se crea con el nodo actual como previo, y se calcula el coste estimado para llegar a la casilla destino mediante la heurística que tenga asignada el algoritmo, acto seguido se añade al OpenSet y se continúa con el siguiente adyacente (en caso de existir).

En resumen, el algoritmo selecciona el nodo del OpenSet con menor coste total (que es la suma del coste real y el estimado), añade sus adyacentes al OpenSet si son alcanzables y no están ya en dicha colección (de ser así se actualizarán sus datos si se han mejorado), y repite la operación hasta llegar a la casilla destino.

A continuación se describe todo el funcionamiento en pseudocódigo:

```
function getPath(origin, destination)
    if origin = destination
        return null
```

```

closedset := empty set
openset := empty set
first_node := the node containing 'origin'
add first_node to openset

while openset is not empty
    best_node := the one having the lowest f value
    remove best_node from openset
    add best_node to closedset
    if best_node = destination
        return build_path(first_node, best_node)
    neighbors := best_node->neighbors
    for each neighbor in neighbors
        from_dir := neighbor->direction
        to_dir := from_dir->opposite
        local_cost := cost_function(to_dir, neighbor, unit)
        if local_cost is impassable
            continue
        tentative_g := get_g(best_node) + local_cost
        if neighbor in closedset
            neighbor_node := neighbor from closedset
            if tentative_g >= get_g(neighbor_node)
                continue
        if neighbor in openset
            neighbor_node := neighbor from openset
            if tentative_g < get_g(neighbor_node)
                set_prev(neighbor_node, to_dir, best_node,
                    tentative_g)
        else
            neighbor_node := create_node(neighbor, to_dir,

```



```

        best_node, tentative_g, get_estimated_cost(
            neighbor, destination, unit))
    add neighbor_node to openset

return null

```

4.2. BeamSearch

Se trata de la primera variante del algoritmo A*, su principal característica es la limitación de tamaño en el OpenSet, con el objetivo de centrar la búsqueda en las mejores opciones que encuentra el algoritmo y evitar análisis innecesarios para conseguir una producción más rápida y eficiente.

El funcionamiento de este algoritmo es idéntico al del algoritmo A*, su característica es que limita el tamaño del OpenSet, de esta manera se evita analizar y almacenar nodos cuyo coste es más elevado, así se reduce el coste computacional del algoritmo y la memoria que utiliza.

Dependiendo del límite se pueden obtener mejores o peores resultados, si éste es demasiado elevado se reduce la mejora en costes y memoria, pero si es demasiado pequeño se pueden descartar nodos que pertenezcan al camino óptimo y obtener como resultado un camino más largo/costoso o, incluso, no llegar a obtener ningún camino. Este último hecho significa que esta variante no es un algoritmo A*, ya que dicho algoritmo siempre encuentra el mejor camino.

Esta característica se lleva a cabo en la clase interna OpenSet, a la que se le ha añadido la funcionalidad de extraer el último elemento de la cola y se ha modificado la funcionalidad de añadir elementos para que cuando se quiera añadir uno nuevo, en caso de haber alcanzado ya el límite de nodos, se extraiga el último de la cola y, si el nuevo posee un coste total menor al del antiguo, se añada al OpenSet y, si no, se restaure la cola.

A continuación se describe todo el funcionamiento en pseudocódigo:

```
function getPath(origin, destination)
```

```

if origin = destination
    return null

closedset := empty set
openset := empty set
first_node := the node containing 'origin'
add first_node to openset

while openset is not empty
    best_node := the one having the lowest f value
    remove best_node from openset
    add best_node to closedset
    if best_node = destination
        return build_path(first_node, best_node)
    neighbors := best_node->neighbors
    for each neighbor in neighbors
        from_dir := neighbor->direction
        to_dir := from_dir->opposite
        local_cost := cost_function(to_dir, neighbor, unit)
        if local_cost is impassable
            continue
        tentative_g := get_g(best_node) + local_cost
        if neighbor in closedset
            neighbor_node := neighbor from closedset
            if tentative_g >= get_g(neighbor_node)
                continue
        if neighbor in openset
            neighbor_node := neighbor from openset
            if tentative_g < get_g(neighbor_node)
                set_prev(neighbor_node, to_dir, best_node,

```



```

        tentative_g)
    else
        neighbor_node := create_node(neighbor, to_dir,
            best_node, tentative_g, get_estimated_cost(
            neighbor, destination, unit))
        if openset_size < node_limit
            add neighbor_node to openset
        else
            last_node := costliest node from openset
            if tentative_g < get_g(last_node)
                remove last_node from openset
                add neighbor_node to openset
    return null

```

4.3. BidirectionalSearch

Esta es la variante del algoritmo A* que me pareció más interesante para desarrollar, ya que realiza dos búsquedas simultáneas, una desde la casilla inicial y otra desde la casilla destino. El funcionamiento de la primera búsqueda es el mismo que el del algoritmo A*, en cambio la segunda búsqueda se realiza de forma inversa a esta última, para ello se creó el InvertedNode que permite la realización de este tipo de búsqueda. Este algoritmo utiliza cuatro colecciones de nodos, un OpenSet y un ClosedSet de nodos estándar y un Openset y un ClosedSet de nodos inversos. El camino puede hallarse de tres maneras distintas, cuando la búsqueda frontal alcanza un nodo del ClosedSet inverso, cuando la búsqueda inversa alcanza un nodo del ClosedSet estándar y en un último caso mucho menos frecuente que es cuando las dos búsquedas coinciden en la misma casilla.

Esta variante es más compleja que el BeamSearch, primero inicializa el OpenSet y el ClosedSet (los normales y los inversos), después crea el nodo inicial para la búsqueda frontal y otro para la búsqueda inversa con la casilla inicial y la casilla destino

respectivamente. Posteriormente se añade cada uno a su OpenSet y se emprende el bucle principal del algoritmo, que esta vez continuará mientras al menos uno de los dos OpenSets tenga algún nodo o hasta que encuentre un camino.

En este caso, en cada iteración del bucle se ejecutan las mismas acciones que en el algoritmo A^* , pero para las dos búsquedas. Seguidamente, aparte de comprobar si alguna de las dos búsquedas ha llegado a su destino, se comprueba si el nodo actual de la búsqueda frontal está contenido en el ClosedSet inverso, si el nodo actual de la búsqueda inversa está contenido en el ClosedSet estándar o si la casilla de los dos nodos actuales es la misma, si se cumple alguna de estas condiciones se procederá a la creación del camino, la cual se explica al final de este apartado.

Si las comprobaciones han resultado negativas se analizan los adyacentes de ambos nodos, el análisis de los nodos adyacentes al nodo estándar actual es idéntico que el del algoritmo A^* , en cambio el análisis de los adyacentes del nodo inverso actual se diferencia en que se crean nodos inversos utilizando la dirección opuesta, ya que el camino no avanza desde el nodo actual, sino hacia él, además al crear el nodo inverso se le asigna el nodo actual como nodo anterior (al contrario que el A^*). El cálculo de los costes se hace de la misma manera, sólo que en vez de calcularlo con la dirección del nodo actual al adyacente se hace desde el adyacente al actual.

La creación del camino se hace de la siguiente manera, se considera cerrado el fragmento que recorre desde el nodo inicial hasta el nodo estándar actual (si la búsqueda frontal ha llegado a un nodo del ClosedSet inverso o ha coincidido con el nodo actual inverso) o hasta el nodo del ClosedSet estándar al que ha llegado la búsqueda inversa si se ha dado este caso. Acto seguido se recorren los nodos inversos hasta la casilla destino convirtiéndolos en nodos estándar para crear el "Path".

A partir de este algoritmo se podría realizar una variante utilizando la misma estrategia que se usó para crear el "BeamSearch" limitando el tamaño de los "OpenSet".



A continuación se describe todo el funcionamiento en pseudocódigo:

```

function getPath(origin, destination)
  if origin = destination
    return null

  closedset_forward := empty set
  closedset_backwards := empty set
  openset_forward := empty set
  openset_backwards := empty set
  first_node_forward := the node containing 'origin'
  first_node_backwards := the node containing 'destination'
  add first_node_forward to openset_forward
  add first_node_backwards to openset_backwards

  while openset_forward is not empty or openset_backwards is not
    empty
    if openset_forward is not empty
      best_node_forward := the one having the lowest f value
      remove best_node_forward from openset_forward
      add best_node_forward to closedset_forward

    if openset_backwards is not empty
      best_node_backwards := the one having the lowest f value
      remove best_node_backwards from openset_backwards
      add best_node_backwards to closedset_backwards

  if best_node_forward = destination
    return build_path(first_node, best_node)

```



```

if best_node_forward = best_node_backwards or
best_node_backwards = origin or
best_node_forward in closedset_backwards or
best_node_backwards in closedset_forward
    if best_node_backwards = origin
        best_node_forward := first_node_forward
    else if best_node_backwards in closedset_forward
        best_node_forward := best_node_backwards

if best_node_forward in closedset_backwards and
best_node_backwards not in closedset_forward
    current_node := best_node_forward
else
    current_node := best_node_backwards

while current_node != destination
    current_node = current_node->next node
    to_dir := get_dir(current_node,
                    best_node_forward)
    local_cost := cost_function(to_dir,
                               best_node_forward, unit)
    tentative_g := get_g(current_node) + local_cost
    neighbor_node := create_node(current_node,
                                to_dir,
                                best_node_forward,
                                tentative_g, 0)
    best_node_forward := neighbor_node

return build_path(first_node_forward,
                 best_node_forward)

```



```

neighbors := best_node_forward->neighbors
for each neighbor in neighbors
    from_dir := neighbor->direction
    to_dir := from_dir->opposite
    local_cost := cost_function(to_dir, neighbor, unit)
    if local_cost is impassable
        continue
    tentative_g := get_g(best_node_forward) + local_cost
    if neighbor in closedset_forward
        neighbor_node := neighbor from closedset_forward
        if tentative_g >= get_g(neighbor_node)
            continue
    if neighbor in openset_forward
        neighbor_node := neighbor from openset_forward
        if tentative_g < get_g(neighbor_node)
            set_prev(neighbor_node, to_dir,
                    best_node_forward, tentative_g)
    else
        neighbor_node := create_node(neighbor, to_dir,
            best_node_forward, tentative_g,
            get_estimated_cost(neighbor, destination,
                unit))
        add neighbor_node to openset_forward

neighbors := best_node_backwards->neighbors
for each neighbor in neighbors
    from_dir := neighbor->direction
    local_cost := cost_function(from_dir,
        best_node_backwards, unit)

```

```

if local_cost is impassable
    continue
tentative_g := get_g(best_node_backwards) + local_cost
if neighbor in closedset_backwards
    neighbor_node := neighbor from
                        closedset_backwards
    if tentative_g >= get_g(neighbor_node)
        continue
if neighbor in openset_backwards
    neighbor_node := neighbor from
                        openset_backwards
    if tentative_g < get_g(neighbor_node)
        set_next(neighbor_node, from_dir,
                best_node_backwards, tentative_g)
else
    neighbor_node := create_inverted_node(neighbor,
        from_dir, best_node_backwards,
        tentative_g, get_estimated_cost(neighbor,
            origin, unit))
    add neighbor_node to openset_backwards

return null

```



5. Diseño de la aplicación de benchmark

Para el desarrollo de la aplicación de benchmark se ha utilizado el patrón de arquitectura software “Vista-Controlador”, que separa la representación de la información de la interacción que el usuario establece con ella. Una vista puede ser cualquier tipo de representación gráfica de datos, en este caso son los componentes de la interfaz, lo que permite la existencia de múltiples vistas diferentes que utilicen el mismo controlador. Un controlador recoge los datos de las vistas para trabajar con ellos internamente y/o modificar los datos mostrados en las vistas.

En las siguientes subsecciones se explica cada uno de los elementos creados dentro de esta arquitectura.

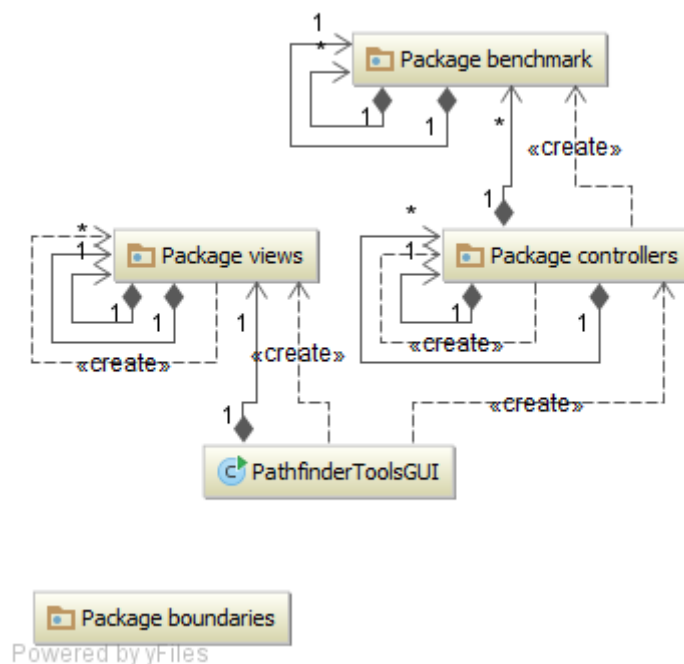


Ilustración 12 - Estructura de la aplicación

5.1. Vistas

- a) **PathfinderToolsGUI**: Esta es la clase principal de la aplicación, por lo que contiene las vistas más importantes de ésta. Estas vistas son la barra de menú, la barra de herramientas, la vista del comparador de algoritmos y la vista de benchmark, estas dos últimas están dentro de un panel con distribución del tipo “CardLayout”, que permite tener solo una de estas vistas activa que se pueda visualizar e intercambiarlas a voluntad. Desde el método “main” de esta clase se inicializa la interfaz y el controlador principal de ésta, que se explica más adelante y es del tipo “PathfinderToolsController”.

- b) **MenuBarView** y **ToolBarView**: La primera vista es una barra de menú y la segunda una barra de herramientas, ambas se inicializan vacías y se les añade botones desde los controladores de la interfaz.

- c) **ComparatorView**: Esta vista contiene un panel con dos vistas “AlgorithmConfigurationView”, un panel con dos listas desplegables, la primera para elegir un tipo de unidad y la segunda para elegir el tipo de coste a mostrar en las siguientes vistas, de tipo “PathfindingBoardView”, por último, también contiene un panel de estadísticas que contiene dos etiquetas, donde se muestran los resultados de las búsquedas simultáneas que se realizan en las vistas “PathfindingBoardView”.

- d) **AlgorithmConfigurationView**: Está compuesta por tres etiquetas seguidas de una lista desplegable cada una. Estas listas permiten seleccionar el tipo de algoritmo, heurística y función de coste, respectivamente, que se van a utilizar para realizar las búsquedas en los “PathfindingBoardView”.

- e) **PathfindingBoardView**: Es la vista principal de la parte del comparador, está compuesta por un panel de “scroll” el cual contiene un panel que contiene diversas capas, éstas son la capa del terreno, la capa de la rejilla, la capa de búsqueda y la capa de caminos.



- f) **BenchmarkView**: Esta vista contiene una vista de tipo “ProblemGeneratorView”, que permite seleccionar diversos escenarios y el número de problemas a generar para cada uno, una vista “AlgorithmSelectionView”, donde se pueden seleccionar varias configuraciones de algoritmos, una lista desplegable para elegir el tipo de unidad que se utiliza en el análisis y un botón para ejecutar éste último.

- g) **ProblemGeneratorView**: Los elementos que componen esta vista son un campo de texto, donde especificar la ruta donde están alojados los escenarios, una lista de escenarios para visualizar todos los que existan en la ruta especificada, un botón para añadir escenarios a la selección que se va a utilizar en el análisis, una lista donde se visualiza dicha selección, un botón para eliminar escenarios de la selección, un campo de texto para introducir el número de problemas a generar y un botón para generarlos.

- h) **AlgorithmSelectionView**: Esta vista está compuesta por una vista “AlgorithmConfigurationView” para crear algoritmos con diferentes configuraciones, un botón para añadirlos a la selección de algoritmos a utilizar en el análisis, una lista donde se puede visualizar dicha selección y un botón para eliminar elementos de ésta.

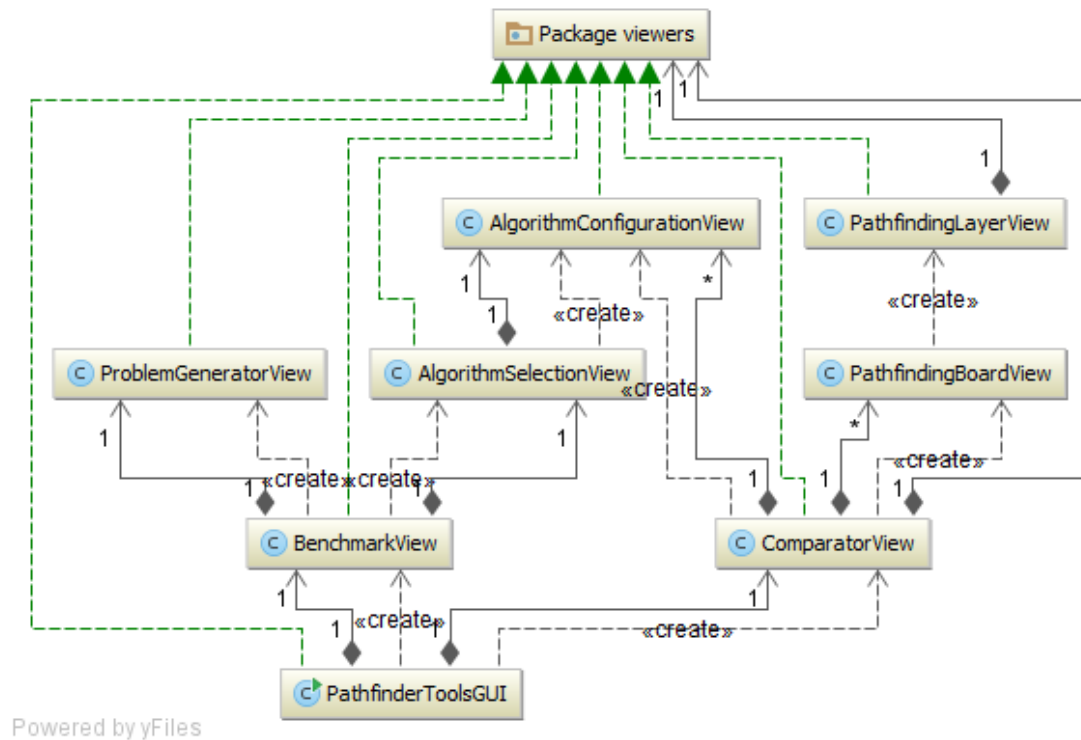


Ilustración 13 - Estructura de las vistas

5.2. Controladores

- a) PathfinderToolsController: Se trata del controlador principal de la aplicación, se sirve de los controladores “ComparatorController” y “BenchmarkController” para controlar las acciones de las vistas “ComparatorView” y “BenchmarkView” respectivamente. Además se encarga de añadir a las barras de menú y herramientas los botones que las componen, y de las acciones que ejecuta cada uno de estos botones. Para realizar esto último se sirve de un “ScenarioController” y del “ComparatorController” anteriormente mencionado.

- b) ScenarioController: Este controlador añade cinco botones cuyas acciones están relacionadas con el escenario con el que se trabaja a la barra de menú y a la de herramientas, sólo tres de estos botones tienen una acción asociada, las cuales se explican a continuación:

- I. **OpenScenario:** Abre un diálogo en el que se puede elegir un fichero, una vez seleccionado abre el escenario en las vistas “PathfindingBoardView” de la “ComparatorView” de la interfaz. Además selecciona esta última como vista activa.
 - II. **CloseScenario:** Cierra el escenario que se encuentre activo.
 - III. **Exit:** Esta acción se encarga de cerrar completamente la aplicación.
- c) **ComparatorController:** De este controlador se extraen cuatro botones para la barra de menú y la barra de herramientas, que ejecutan las siguientes acciones:
- I. **ViewGrid:** Muestra u oculta la capa de rejilla de las dos “PathfindingBoardView” de la “ComparatorView”.
 - II. **ViewUnits:** Muestra u oculta la capa de unidades de las dos “PathfindingBoardView” de la “ComparatorView” (esta acción no tiene ningún efecto real, ya que la vista que se ha desarrollado no posee capa de unidades).
 - III. **ZoomIn:** Redimensiona las vistas “PathfindingBoardView” a un tamaño mayor para conseguir un efecto de zoom.
 - IV. **ZoomOut:** Redimensiona las vistas “PathfindingBoardView” a un tamaño menor para conseguir un efecto de zoom.

Este controlador maneja los eventos de ratón que tienen lugar en las “PathfindingBoardView” mediante dos oyentes que se explican a continuación, el primero se encarga de cambiar el modo de interacción en el que se encuentra la aplicación y el funcionamiento del segundo depende de este modo:

- I. **BoardMouseListener:** Captura los eventos de click de ratón y ejecuta una función distinta dependiendo del botón que se haya pulsado.
 - i. **Botón izquierdo:** Marca la casilla sobre la que se ha hecho click como la casilla seleccionada,

deseleccionando así la que hubiera sido seleccionada anteriormente (en caso de haberse seleccionado). También cambia el modo de interacción a “UNIT_ORDERS”, se limpian las capas de costes y caminos y, además, el panel de estadísticas.

- ii. Botón central: Deselecciona la casilla que se encuentra seleccionada (si es el caso), el modo de interacción pasa a ser “FREE” y, en la capa de caminos, se limpia el camino generado por el “BoardMouseMotionListener”.
- iii. Botón derecho: Si el modo de interacción es “UNIT_ORDERS”, se realiza una búsqueda, con las configuraciones que se hayan seleccionado en el panel de configuraciones, para llegar desde la casilla que se encuentre seleccionada hasta la casilla sobre la que se ha hecho click. También se dibuja el camino encontrado en la capa de caminos y el coste de los nodos creados por el “Pathfinder” utilizado en la capa de búsqueda y se borra el camino que se había obtenido en el “BoardMouseMotionListener”. Por último, se actualiza el panel de estadísticas con los datos obtenidos en la búsqueda que se ha hecho en cada tablero.

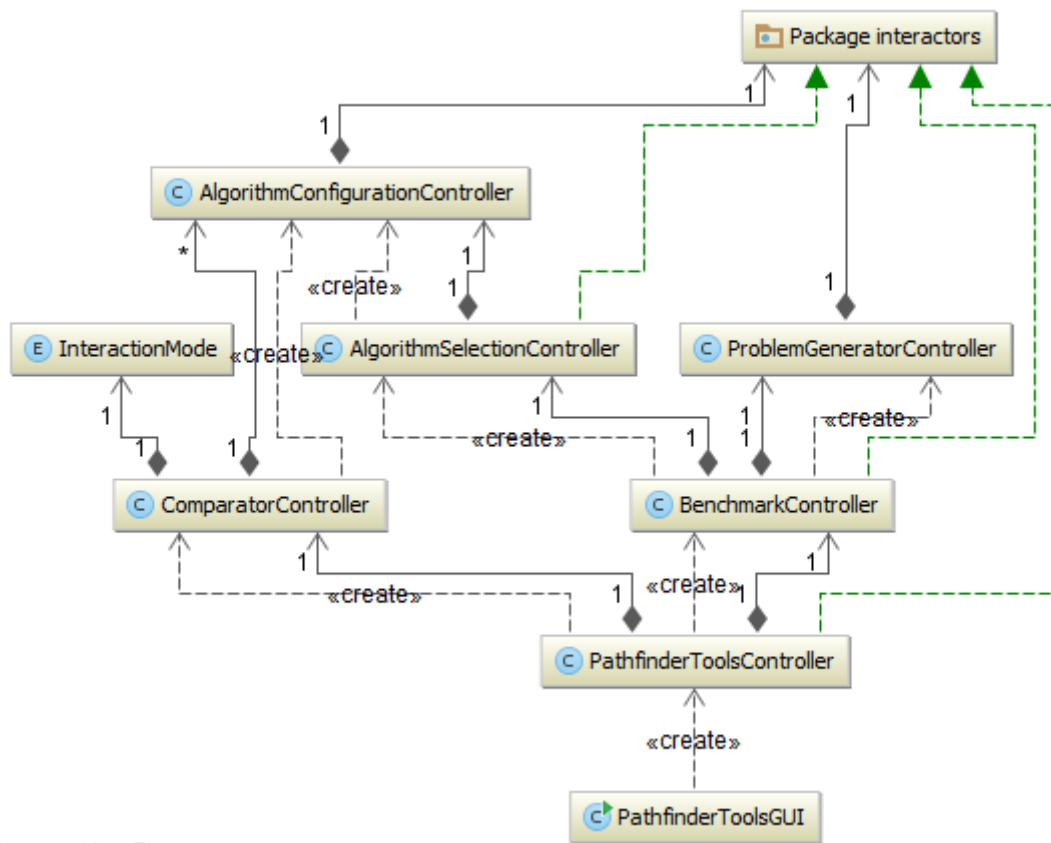
- II. BoardMouseMotionListener: En el caso de que se haya seleccionado una casilla con el oyente explicado anteriormente (el estado de interacción activo será “UNIT_ORDERS”), este oyente capturará los eventos de movimiento de ratón sobre la vista realizando una búsqueda desde la casilla seleccionada hasta la casilla donde se encuentre el puntero del ratón, pero sin representar el coste de los nodos analizados, sino mostrando sólo el camino encontrado y el coste real de cada nodo que lo compone.

El “ComparatorController” también utiliza un controlador llamado “AlgorithmConfigurationController” para manejar



las acciones de las vistas de configuración “AlgorithmConfigurationView” del panel de configuraciones.

- d) AlgorithmConfigurationController: Este controlador captura los eventos que tienen lugar en las listas desplegables de una “AlgorithmConfigurationView” mediante oyentes, que reciben la opción seleccionada y realizan los cambios pertinentes en la configuración asociada.
- e) BenchmarkController: Este controlador utiliza los controladores “ProblemGeneratorController” y “AlgorithmSelectionController” para controlar las acciones que tienen lugar en las vistas “ProblemGeneratorView” y “AlgorithmSelectionView”, que contiene la vista “BenchmarkView”. Aparte, captura los eventos de la lista desplegable de tipos de unidad para realizar el análisis y es el encargado de realizar este último.
- f) ProblemGeneratorController: Se encarga de capturar los eventos de los botones de añadir y eliminar escenarios de la lista de selección mediante oyentes y, además utiliza un oyente con el botón de generar problemas para llamar ejecutar la función que los genera y almacena. Un problema contiene datos sobre el escenario para el que se ha creado, un punto de origen y otro destino para realizar la búsqueda.
- g) AlgorithmSelectionController: Se sirve de un controlador del tipo “AlgorithmConfigurationController” para controlar las acciones de la “AlgorithmConfigurationView” contenida en la vista de tipo “AlgorithmSelectionView” que controla, además, contiene los oyentes que manejan los eventos de los botones que añaden o eliminan algoritmos de la lista de selección de éstos.



Powered by yFiles

Ilustración 14 - Estructura de los controladores

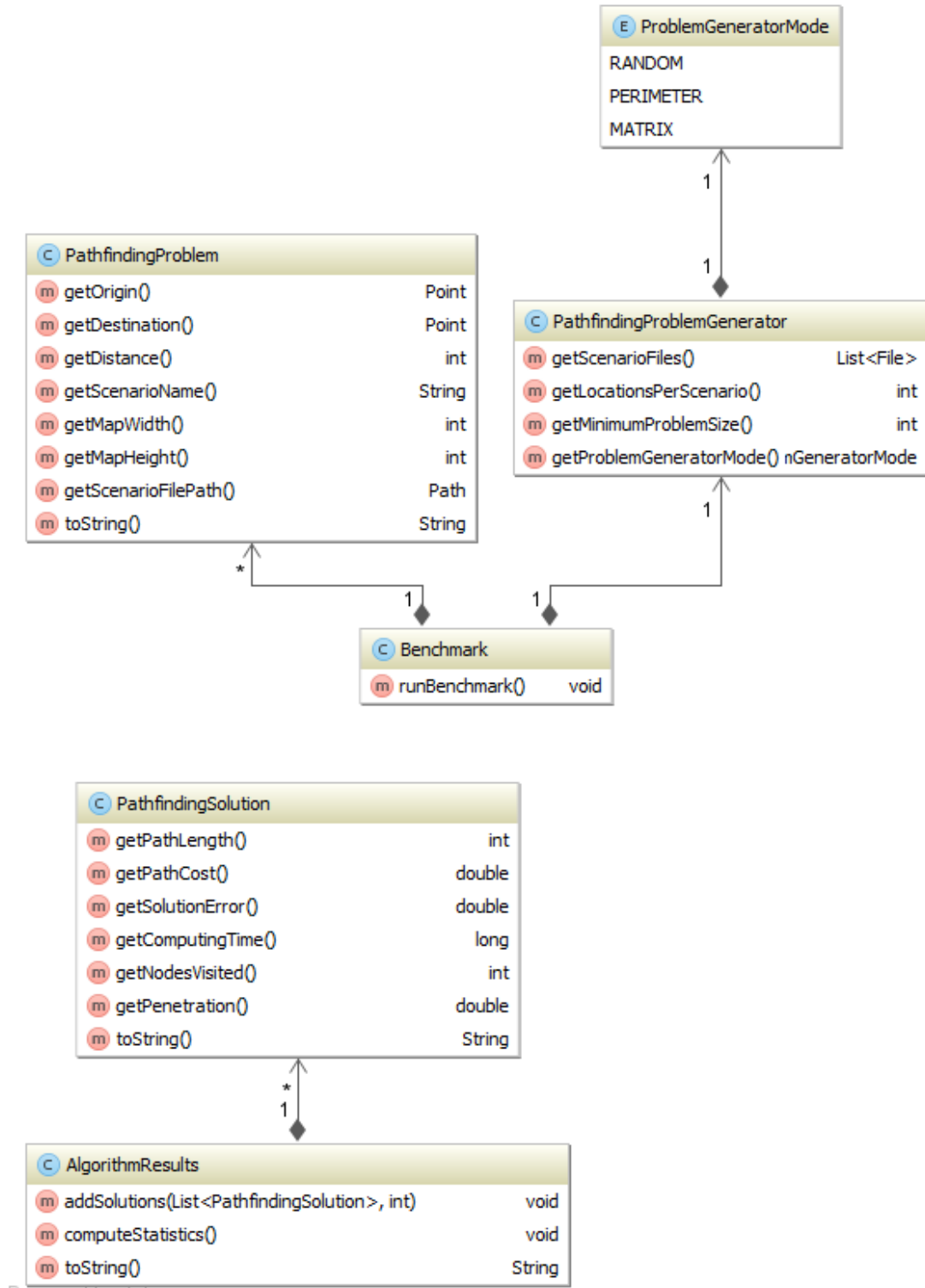


Ilustración 15 - Estructura del generador de problemas y las soluciones

6. Implementación

En esta sección se explican diferentes aspectos, como el entorno donde se ha desarrollado el proyecto o algunas de las características de su implementación.

6.1. Entorno de desarrollo

El lenguaje utilizado para la implementación de la librería y la aplicación ha sido el mismo que en el que se encontraba implementado el juego original, éste es “Java”, más concretamente la versión del kit de desarrollo 1.7.0_25 de Java. Se ha utilizado el Entorno de Desarrollo Integrado “NetBeans” por varias razones, como su ayuda visual a la hora de programar, su compatibilidad con Java pero, sobre todo, por la posibilidad de utilizar la herramienta “Git”, que ofrece un buen control de versiones del proyecto y ha permitido trabajar de una manera más fácil mediante la creación de ramas. También se ha utilizado “BitBucket” como repositorio, el cual ofrece una visión muy buena del flujo de trabajo sobre un proyecto y la creación y asignación de tareas desde su web.

6.2. Estructura del proyecto

En el proyecto se han creado interfaces que sirven para dar una mayor escalabilidad a los componentes de éste, por ejemplo, las vistas de la aplicación de benchmark implementan interfaces para que, si en algún momento se quiere crear una vista muy parecida a otra que sólo difiera en alguna funcionalidad se pueda utilizar de una manera más sencilla desde el controlador que la controle.

→Diagrama

6.2.1. Elementos externos utilizados

En el desarrollo del proyecto se han utilizado clases del proyecto Ares original, éstos se comentan a continuación:

- a) **Tile:** Esta clase representa una casilla de un escenario, y contiene datos sobre el terreno que se han utilizado para calcular los costes reales durante las búsquedas, como son los costes de entrada a la casilla y los costes mínimos de salida por cada una de las seis direcciones posibles (en cada caso). Además, un objeto de tipo “Tile” también contiene una colección de Tiles que representan los adyacentes de ésta.

- b) **Unit:** Representa los datos de una unidad. De este objeto se ha utilizado la propiedad “movementType”, que sirve para calcular el coste de movimiento entre casillas. Por tanto, también se ha utilizado la estructura de datos “MovementType” de tipo enumerable la cual devuelve distintos factores de movimiento dependiendo del tipo de ésta.

6.2.2. Estructuras de datos

Aquí se explican las estructuras de datos que se han utilizado para la realización de los algoritmos de búsqueda a la hora de representar el “OpenSet” y el “ClosedSet”.

- a) **ClosedSet:** Para representar este elemento se ha utilizado una estructura de datos de tipo “HashMap”, el motivo por el que se ha elegido este tipo es que permite una manera fácil y rápida de almacenar y recuperar los nodos que se almacenan dentro mediante un número identificativo (llamado índice), además el coste de acceso a los datos almacenados es $O(1)$, es decir, que es siempre constante, independientemente del número de elementos que haya almacenados (esto no ocurre en otras estructuras de datos, como la “ArrayList”, donde el coste sí depende del número de elementos).

- b) **OpenSet:** Para el OpenSet se ha creado una clase interna en la clase de los algoritmos que contiene una “PriorityQueue” donde se almacenan los nodos. El motivo de utilizar dicha estructura de datos es el de mantener un orden de los nodos para ser utilizados según su valor de su coste total calculado (f), además, también se almacenan en un “HashMap” para

cuando se necesita comprobar si el “OpenSet” contiene un nodo específico (a través de su índice, como en el “ClosedSet”) o se quiere obtener un nodo sin extraerlo de la cola de prioridad.

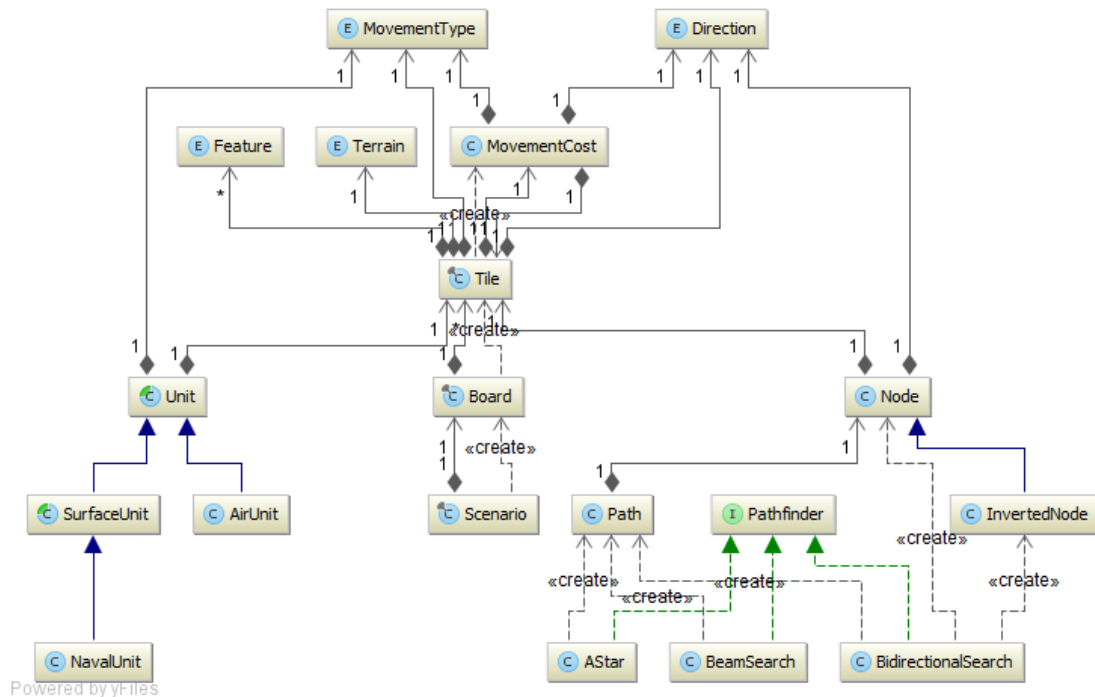


Ilustración 16 - Conexión entre los nodos y el entorno

6.3. Interfaz gráfica y casos de uso

En este apartado se describen todas las herramientas que componen la interfaz y su funcionamiento. La ventana principal (con la vista del comparador seleccionada) está compuesta por una barra de menú (1), una barra de herramientas (2), un panel de configuraciones (3), otro de configuración mutua (4), los dos tableros (5) y un panel de estadísticas (6).

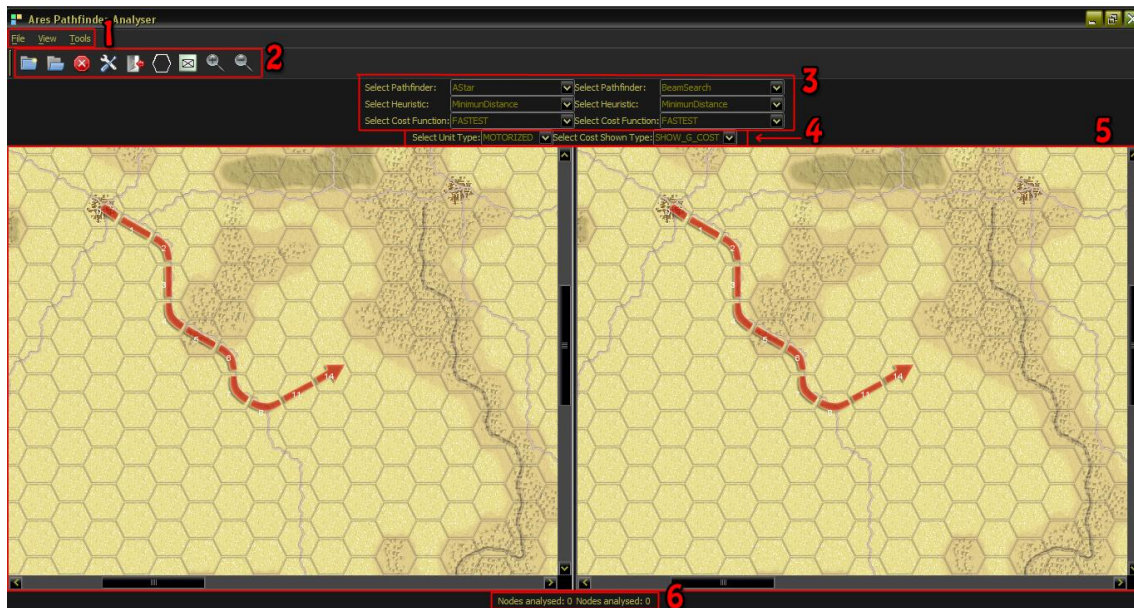


Ilustración 17 - Interfaz del comparador

6.3.1. Barra de menú

La barra de menú posee tres submenús, cada uno de ellos posee una serie de botones que afectan a un aspecto diferente de la interfaz. Algunos de estos botones no llegan a tener una funcionalidad útil en esta aplicación, pero sí que la tienen en el proyecto Ares, de ahí su presencia aquí. A continuación se explica cada uno de estos submenús por separado.

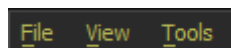


Ilustración 18 - Barra de menú

6.3.1.1. Menú "File"

Las acciones que realizan los botones de este submenú permiten al usuario realizar operaciones relacionadas con el escenario, seguidamente se detalla el funcionamiento de cada uno de ellos.

a) New Game

Este botón abre un diálogo que permite seleccionar un escenario de la carpeta de escenarios de la plataforma. Una vez se selecciona el archivo del escenario a cargar, éste se carga en los dos tableros de la interfaz.

b) Load Game

El funcionamiento de este botón no se ha definido aún.

c) Close Game

El botón “close game” borra el escenario que se tenga abierto en el momento en el que se ha pulsado (en caso de haber alguno abierto).

d) Settings

El funcionamiento de este botón no se ha definido aún.

e) Exit

Este botón se encarga de cerrar la aplicación por completo.

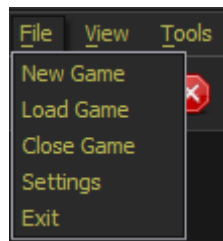


Ilustración 19 - Submenú “File”

6.3.1.2. Menú “View”

En este submenú existen cuatro botones cuyas acciones están relacionadas con las vistas de los tableros, su funcionamiento se explica a continuación:

a) Show Grid

Este botón permite mostrar u ocultar la rejilla de los tableros.

b) Show Units

Este botón permite mostrar u ocultar las unidades en los tableros, aunque en este aplicación no se llega a mostrar ninguna unidad.

c) Zoom In

Con este botón se puede aumentar el tamaño de los tableros, dentro de sus vistas, para hacer un efecto de zoom positivo.

d) Zoom Out

El funcionamiento de este botón es el mismo que el del anterior pero en sentido negativo.

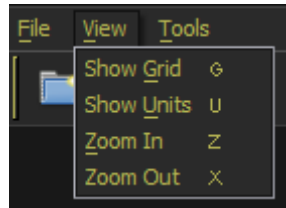


Ilustración 20 - Submenú "View"

6.3.1.3. Menú "Tools"

Este submenú permite cambiar la vista principal que se esté visualizando, y sus dos botones representan a cada una de estas vistas (vista del comparador y vista de benchmark).

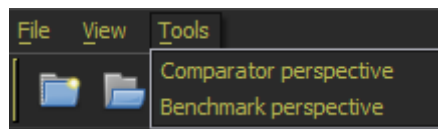


Ilustración 21 - Submenú "Tools"

6.3.2. Barra de herramientas

La barra de herramientas está compuesta por los botones que corresponden a los elementos del menú, por tanto éstos ejecutan las mismas acciones. En resumen, esta barra de herramientas es un acceso rápido y visual a todos los elementos del menú.



Ilustración 22 - Barra de herramientas

6.3.3. Vista del comparador

Esta vista permite comparar el funcionamiento de dos configuraciones de los algoritmos desarrollados, a continuación se explica cada uno de sus componentes.

6.3.3.1. Panel de configuraciones

El panel de configuraciones consta de dos vistas de tipo “PathfinderConfigurationView”, cada una ligada a uno de los tableros. Este tipo de vista está compuesto por tres cajas desplegadas que permiten seleccionar un algoritmo, una heurística y una función de coste para encontrar un camino en su tablero correspondiente. El controlador del comparador es el que inicializa estas vistas asignándoles los valores seleccionables de cada caja.

Aquí se almacena la información seleccionada por el usuario que es utilizada posteriormente en los tableros.

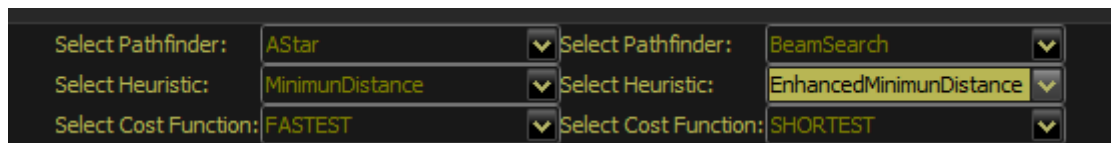


Ilustración 23 - Panel de configuraciones

6.3.3.2. Panel de configuración mutua

Este panel de configuración permite seleccionar opciones que afectan a ambos tableros y está compuesto por dos cajas desplegadas, una corresponde al tipo de unidad a utilizar para las búsquedas de caminos y otra al tipo de coste a mostrar en los tableros. Los tipos de coste disponibles son: coste real (G), coste estimado (H) y coste total (F).

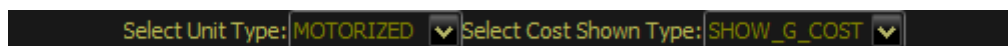


Ilustración 24 - Panel de configuración mutua

6.3.3.3. Tableros

Los tableros son los elementos principales de la interfaz, en ellos se cargan los escenarios en los que se hace la búsqueda de caminos. La vista que representa un tablero es la clase “PathSearchBoardView”, que está compuesta por varias capas alojadas en un panel con scroll, el cual permite ver el escenario cuando sus dimensiones superan las de la vista. Las capas que componen la vista son:

- a) TerrainLayerView

Esta capa es la responsable de mostrar la imagen que representa el escenario.

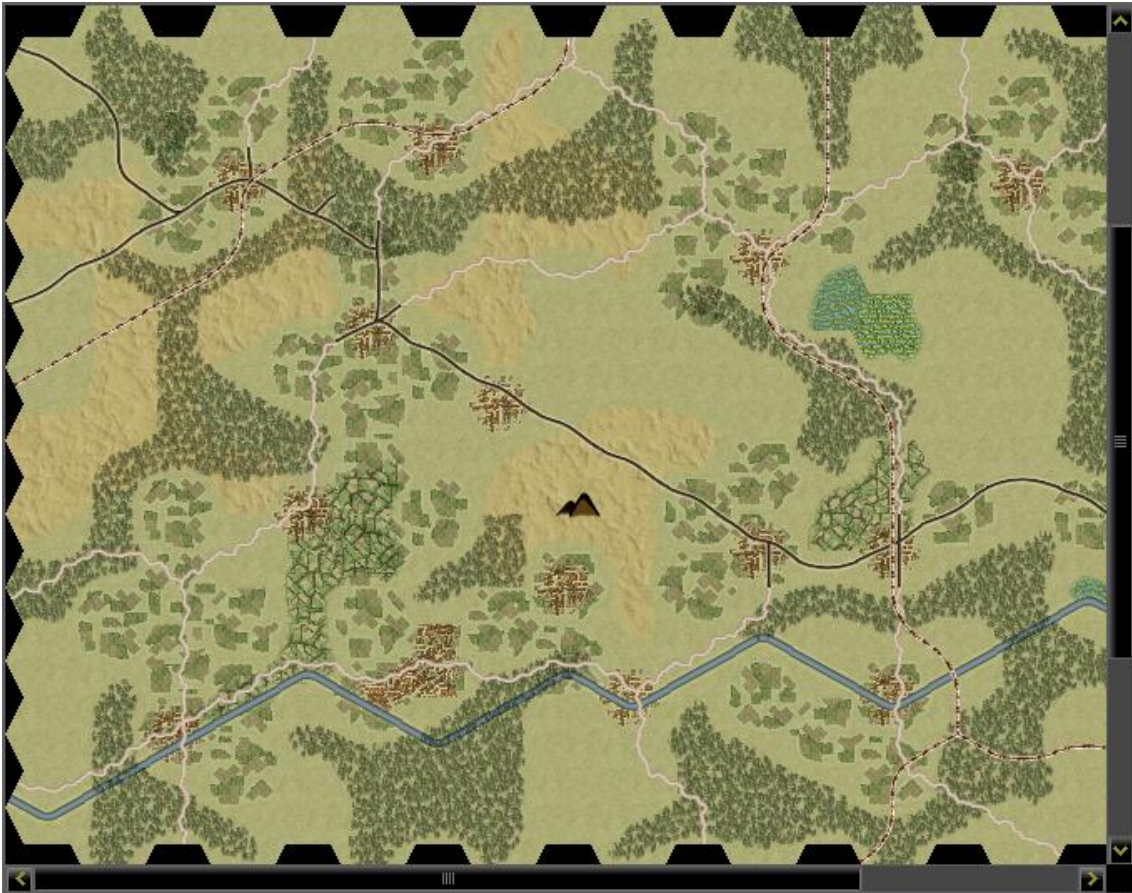


Ilustración 25 - Capa de terreno

b) GridLayerView

La rejilla hexagonal del tablero se dibuja en esta capa, es posible ocultarla o mostrarla desde la barra de menú o la de herramientas.

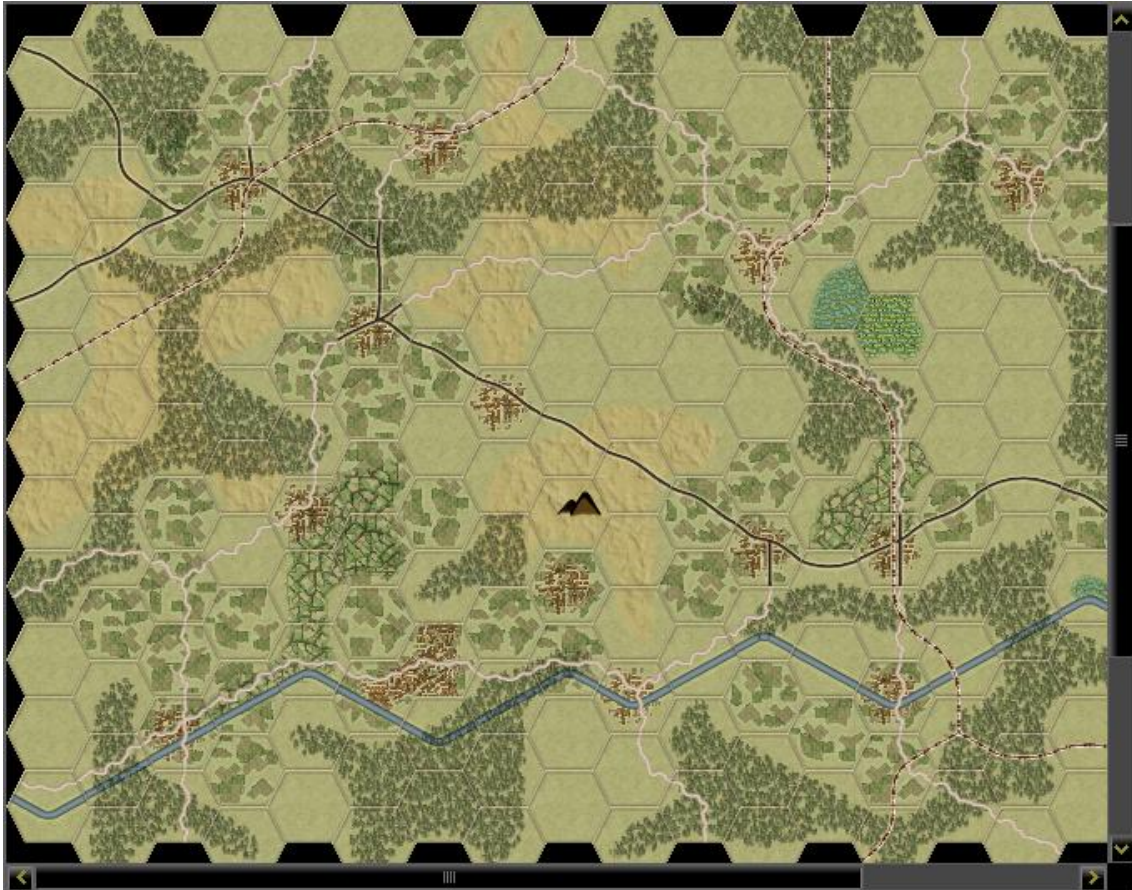


Ilustración 26 - Capa de rejilla

c) PathSearchLayerView

En esta capa se produce la representación de los costes de los nodos analizados en el proceso de búsqueda de caminos, estos costes se representan por un color. El código del color mostrado se calcula según el coste total del camino en un espectro de verde a rojo, por tanto, en una escala de cero a cien (siendo cien el coste total del camino), el coste del nodo definirá si el color a mostrar es más verde o más rojo.

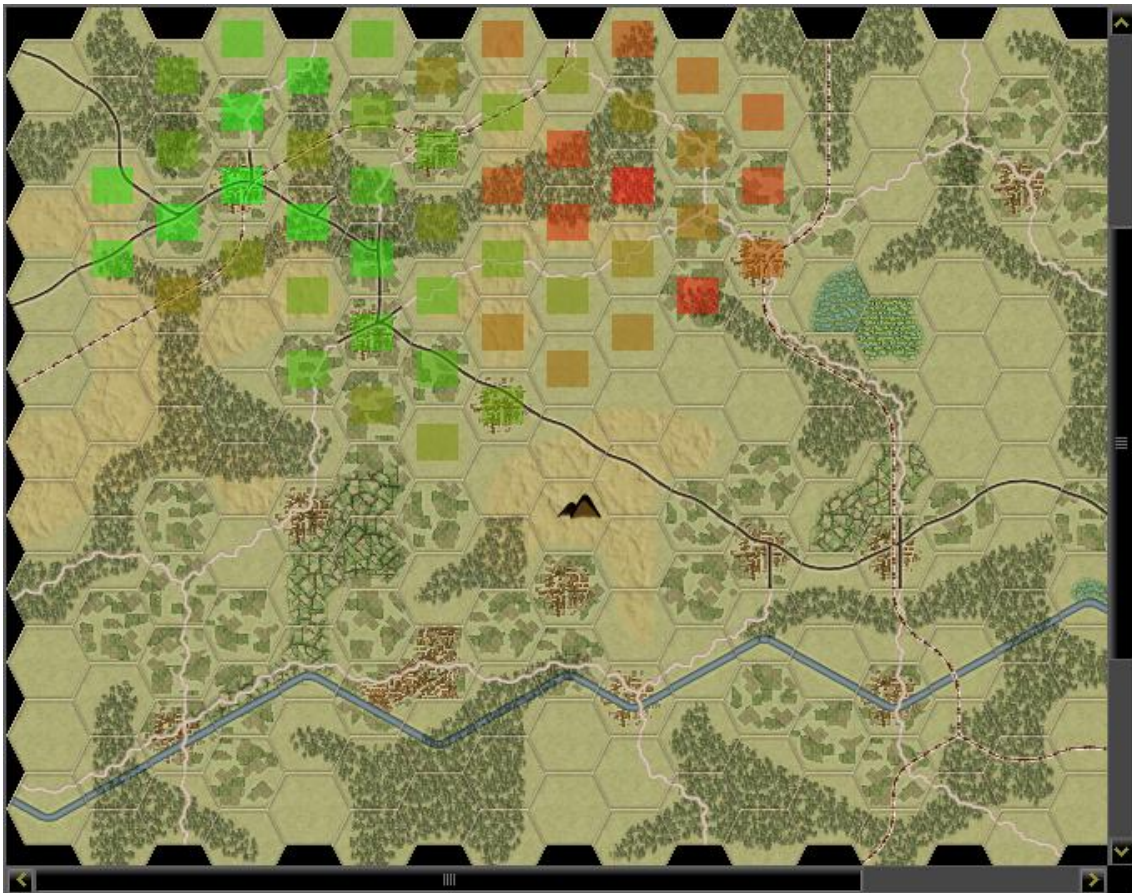


Ilustración 27 - Capa de búsqueda

d) ArrowLayerView

Esta capa se encarga de representar el camino obtenido ya sea habiendo seleccionado una casilla o moviendo el ratón por encima del tablero. Dada una instancia de la clase "Path", recorre todos sus nodos dibujando una línea, una curva o una punta de flecha según corresponda.

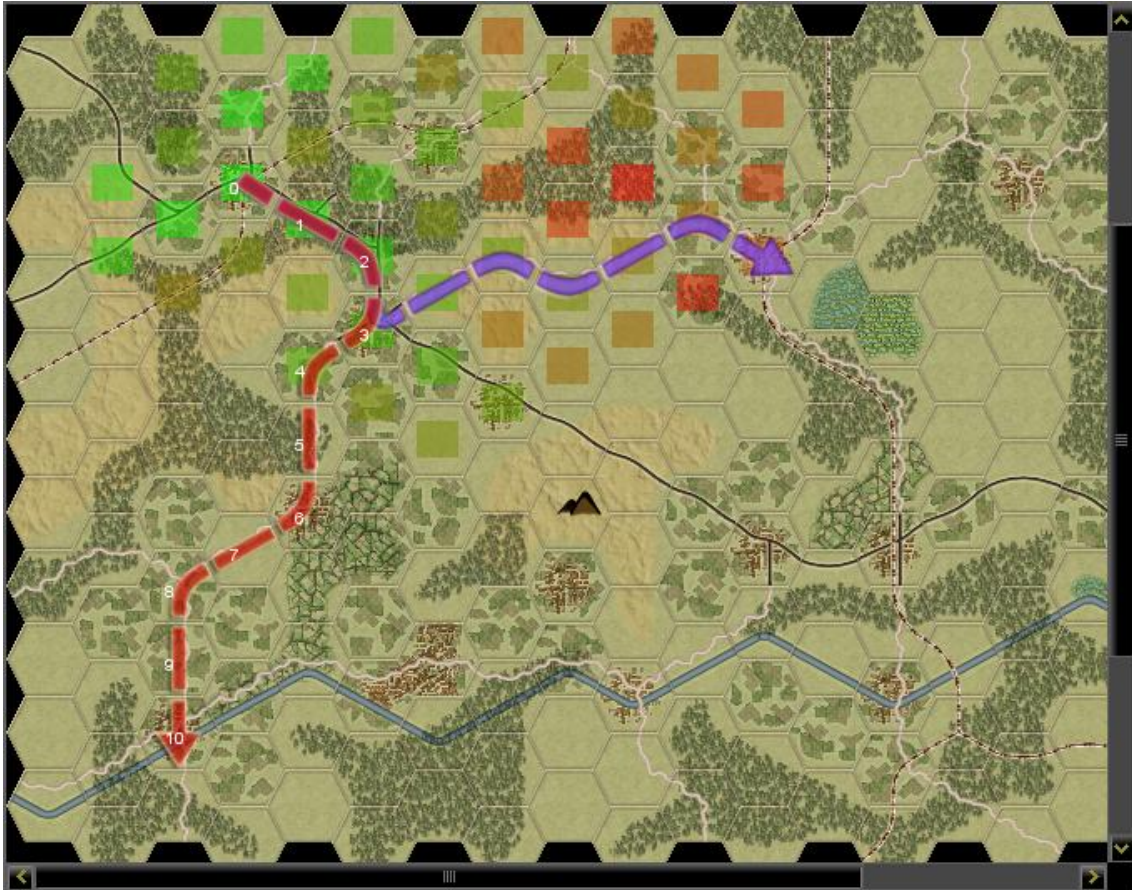
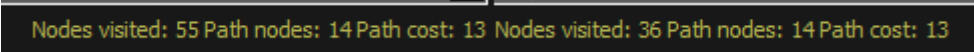


Ilustración 28 - Capa de flechas

La interacción con los tableros es simultánea, es decir, los eventos de ratón capturados por uno de los tableros ejecutan sus acciones correspondientes en ambos. Esto permite una comparación visual más rápida y cómoda

6.3.3.4. *Panel de estadísticas*

El panel de estadísticas consta de dos etiquetas (una para cada tablero), y se ocupa de ofrecer datos sobre el proceso de búsqueda de los caminos encontrados en los tableros. El contenido de estas etiquetas se actualiza cada vez que se selecciona un camino nuevo, cuando esto ocurre se muestra el número de nodos analizados en el proceso de búsqueda, el número de nodos que componen el camino y el coste total de éste.

A dark rectangular panel with yellow text. The text is split into two groups by a vertical line. The left group shows 'Nodes visited: 55 Path nodes: 14 Path cost: 13' and the right group shows 'Nodes visited: 36 Path nodes: 14 Path cost: 13'.

Nodes visited: 55 Path nodes: 14 Path cost: 13 Nodes visited: 36 Path nodes: 14 Path cost: 13

Ilustración 29 - Panel de estadísticas

6.3.4. Vista de benchmark

En esta vista se seleccionan todos los elementos necesarios para ejecutar el análisis de los problemas creados desde ella también. A continuación se explican todos los elementos que lo componen.

6.3.4.1. Panel de creación de problemas

Este panel posee un campo de texto que permite seleccionar una carpeta que contenga escenarios que puedan ser cargados por la aplicación. Una vez se selecciona un directorio se llena una lista con el nombre de cada uno de los escenarios que contiene dicho directorio. El botón de añadir se encarga de añadir el escenario que se encuentre seleccionado a la lista de escenarios para los que se van a generar problemas, y el botón de eliminar se encarga de eliminar el escenario seleccionado en la lista de selección de ésta. Por último, también existe un campo de texto donde introducir el número de problemas que se quieren generar para cada uno de los escenarios seleccionados, y el botón que se encarga de generar estos problemas.

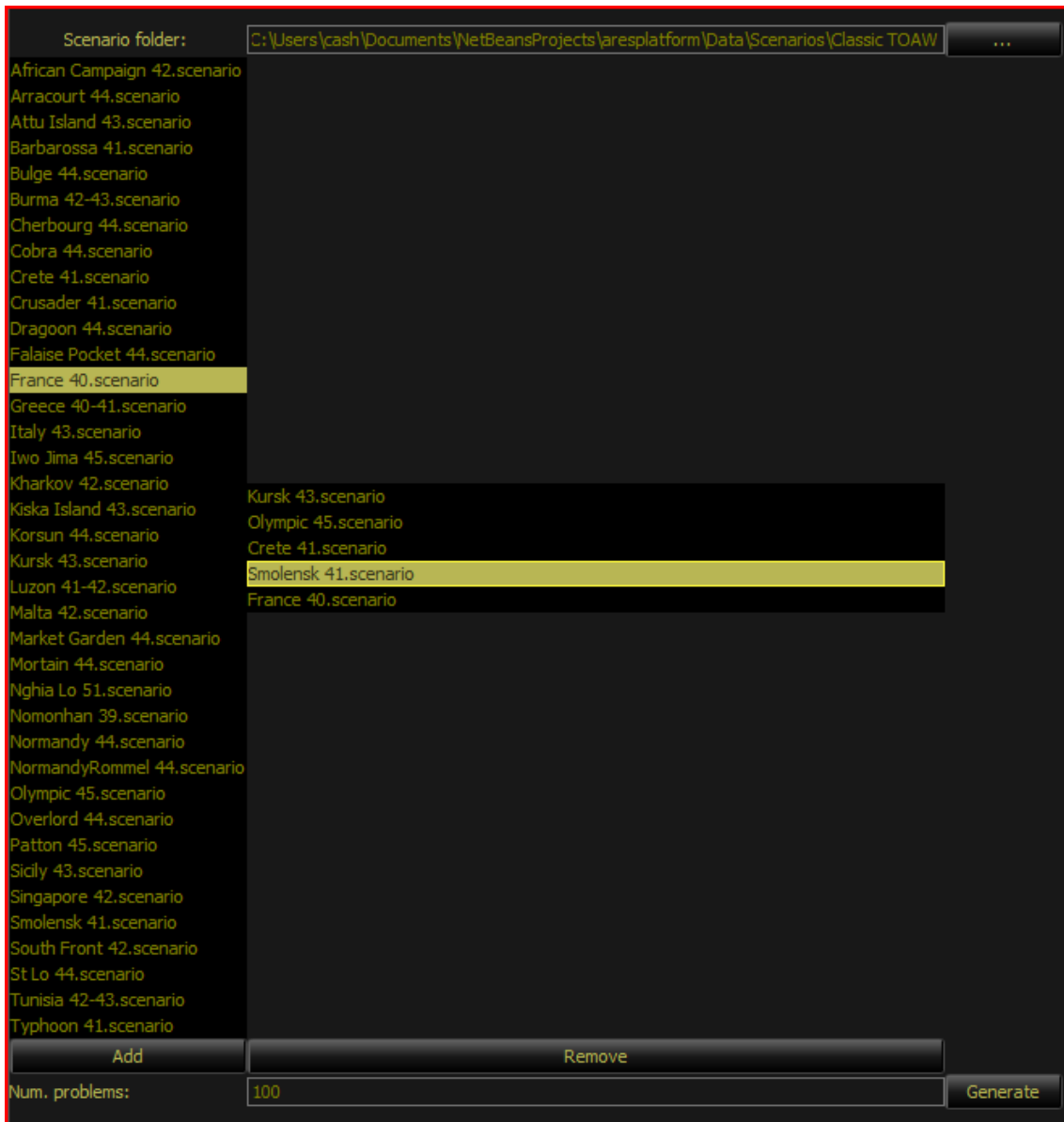


Ilustración 30 - Panel de creación de problemas

6.3.4.2. Panel de selección de algoritmos

En este panel se utiliza una vista de las utilizadas en el panel de configuraciones de la vista del comparador, que permite seleccionar una configuración. También contiene un botón para añadir la configuración que se haya construido a la lista de configuraciones creadas y otro botón que permite eliminar la configuración que se haya seleccionado de la lista.

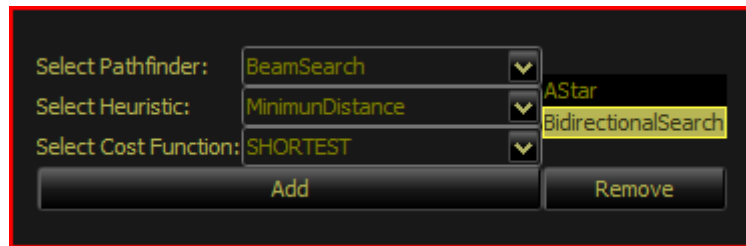


Ilustración 31 - Panel de selección de algoritmos

Además, la vista de benchmark contiene una lista desplegable donde se puede elegir el tipo de unidad que se va a utilizar en los algoritmos seleccionados y un botón que ejecuta el análisis.



Ilustración 32 - Lista de unidades y botón de ejecución

7. Análisis de resultados

En esta sección se comentan los resultados que se obtienen al hacer comparaciones con el comparador de algoritmos y al realizar un análisis con la herramienta benchmark, el último caso se explica el proceso que se sigue para obtener estos resultados utilizando un ejemplo.

En el primer ejemplo se compara una búsqueda realizada por el algoritmo “AStar” con la realizada por el algoritmo “BeamSearch” para el mismo caso. Como se puede observar, el algoritmo “BeamSearch” obtiene el camino óptimo (al igual que el “AStar”) visitando muchos menos nodos.



Ilustración 33 - "AStar" vs "BeamSearch" (buen resultado)

En el siguiente ejemplo se puede ver un ejemplo de la no optimalidad del algoritmo “BeamSearch”, pues en este caso no encuentra el camino óptimo.

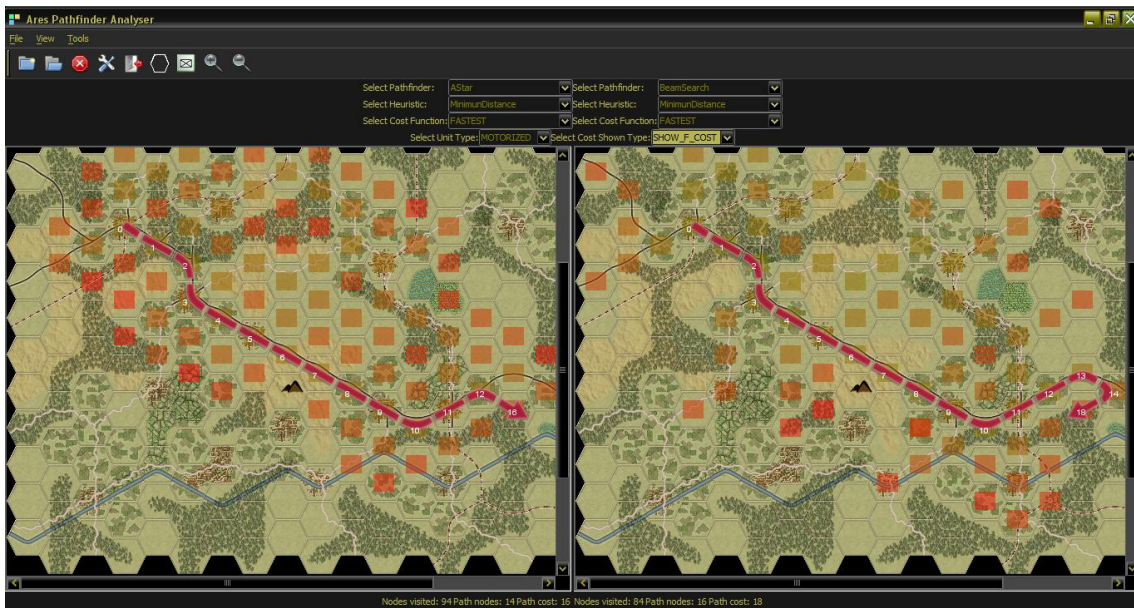


Ilustración 34 - "AStar" vs "BeamSearch" (mal resultado)

Por último, en este ejemplo se puede ver un ejemplo del funcionamiento del algoritmo “BidirectionalSearch” comparado con la del “AStar”. Se puede observar como la mayor parte de los nodos analizados se encuentran alrededor de la casilla origen y de la casilla destino, y como se han unido las dos búsquedas en el centro.



Ilustración 35 - "AStar" vs "Bidireccional"

Ahora se explica el análisis de la herramienta benchmark. Primero se intentan generar tantos problemas como el usuario ha introducido en el panel de generación de problemas. Para generar un problema se escogen dos casillas aleatorias del tablero, pero esto no significa que ambas casillas sean válidas, así que se generan tantos como sea posible para cada uno de los escenarios seleccionados. Acto seguido se muestra el número de problemas generados para cada escenario:

Scenario Arracourt 44 , area = 441, numProblems = 6

Scenario Crusader 41 , area = 3192, numProblems = 43

Después, al ejecutar la función de benchmark se construye un algoritmo de cada tipo de los seleccionados para cada uno de los problemas, y se muestra las soluciones encontradas por cada algoritmo para cada escenario, como se muestra a continuación:

*Solutions found for Arracourt 44 (6/6) by
AStar{heuristic=MinimunDistance, costFunction=FASTEST}*

*Solutions found for Arracourt 44 (6/6) by
BeamSearch{heuristic=MinimunDistance, costFunction=FASTEST}*

*Solutions found for Arracourt 44 (6/6) by
BidirectionalSearch{heuristic=MinimunDistance,
costFunction=FASTEST}*

*Solutions found for Crusader 41 (23/43) by
AStar{heuristic=MinimunDistance, costFunction=FASTEST}*

*Solutions found for Crusader 41 (22/43) by
BeamSearch{heuristic=MinimunDistance, costFunction=FASTEST}*

*Solutions found for Crusader 41 (23/43) by
BidirectionalSearch{heuristic=MinimunDistance,
costFunction=FASTEST}*

Como se puede observar, los tres algoritmos que se han seleccionado han encontrado una solución para cada uno de los problemas que se han generado para el escenario “Arracourt 44”, pero para el escenario “Crusader 41” el algoritmo “Beamsearch” ha obtenido una solución menos que los demás algoritmos. Esto último se debe a la forma de funcionar de “BeamSearch” ya que al limitar su



capacidad en el “OpenSet” se descartan algunos caminos, como ya se ha explicado en la sección de algoritmos.

Por último, se realizan una serie de estadísticas para cada uno de los algoritmos seleccionados, estas son: el número de problemas afrontados, el número de soluciones encontradas, la media de la longitud de los caminos construidos, la media del coste de los caminos construidos, el tiempo que ha tardado en obtener las soluciones y la cantidad total de nodos visitados durante las búsquedas. Estos son los resultados del ejemplo:

**** Results for AStar{heuristic=MinimumDistance, costFunction=FASTEST}:*

numProblems=49

numSolutions=29

pathLength=27,97

pathCost=35,66

computingTime=3227606,90

nodesVisited=385,03

**** Results for BeamSearch{heuristic=MinimumDistance, costFunction=FASTEST}:*

numProblems=49

numSolutions=28

pathLength=29,11

pathCost=37,14

computingTime=1671439,29

nodesVisited=154,54

**** Results for BidirectionalSearch{heuristic=MinimumDistance, costFunction=FASTEST}:*

numProblems=49

numSolutions=29

pathLength=27,14

pathCost=36,28

computingTime=4841410,34

nodesVisited=246,45

En este ejemplo se pueden sacar las siguientes conclusiones utilizando los resultados obtenidos para el algoritmo “AStar” como base:

- a) El algoritmo “BeamSearch” ha obtenido el menor número de nodos visitados y el menor tiempo, pero hay un problema para el que no ha obtenido solución y la media de longitud y coste que ha obtenido está por encima de las demás, por tanto se puede decir que ha sido el algoritmo más eficiente en cuanto a costes pero menos fiable en cuanto a resultados ya que la calidad de las soluciones es peor, lo que demuestra que no es un algoritmo A*, ya que no garantiza la optimalidad de las soluciones.
- b) El algoritmo “BidirectionalSearch” ha obtenido un número de nodos visitados menor al de “AStar” pero mayor al de “BeamSearch”, también ha obtenido una media de coste intermedia pero su tiempo es el más alto de todos, aunque la media de longitud es buena. Se podría decir que la ejecución de este algoritmo ha sido fiable pero costosa.



8. Conclusión y opinión personal

La realización de este proyecto fue un reto desde el principio, ya que mis nociones de búsqueda de caminos al principio de ésta no eran muy extensas, pero la ilusión y las ganas de aprender sobre el tema me han empujado a seguir, aunque las circunstancias no me hayan dejado dedicarle todo el tiempo y energía que me hubiera gustado. Aun así he disfrutado mucho creando los algoritmos y aprendiendo a utilizar el patrón Vista-Controlador.

Este proyecto podría tener varias ampliaciones, como el desarrollo de nuevos algoritmos o variaciones de los que ya existen, o incluir elementos circunstanciales tales como presencia enemiga o elementos climatológicos que puedan modificar el funcionamiento de los algoritmos.

Por último, quiero dar las gracias a Mario, el director del proyecto, por implicarse tanto en el desarrollo del proyecto y también por su tiempo y esfuerzo, además de haberme brindado las herramientas necesarias para trabajar lo más cómodamente posible. Espero que la realización de este proyecto suponga un avance en el desarrollo de la inteligencia artificial proyecto Ares.

9. Referencias

- [http://en.wikipedia.org/wiki/Graph_\(mathematics\)](http://en.wikipedia.org/wiki/Graph_(mathematics))
- http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- http://en.wikipedia.org/wiki/A*_search_algorithm
- <http://theory.stanford.edu/~amitp/GameProgramming/>
- <http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- <https://netbeans.org/>
- <https://bitbucket.org/>