Búsqueda de Caminos Mediante MVC

Alejandro Rodríguez, Rubén Palmer y Sergi Mayol

Resumen— Esta aplicación presenta una interfaz gráfica de usuario que le permite interactivamente encontrar, empleando el patrón de diseño MVC, el Camino mínimo entre dos pueblos de Ibiza y Formentera, pasando por un tercero. La solución viene dado por el algoritmo de Dijkstra. El programa permite al usuario seleccionar el origen, el destino y el punto intermedio.

Vídeo - ver vídeo

1. Introducción

En este artículo explicaremos el funcionamiento y la implementación de nuestra práctica estructurando y dividiendo sus partes en los siguientes apartados:

Inicialmente, se citarán las librerías usadas en este proyecto. Para aquellas que se hayan desarrollado específicamente, se expondrá una breve guía de su funcionamiento y uso.

Seguidamente, se explicará cuál es la arquitectura, interfaces, funcionamiento e implementación de nuestro **M**odelo-**V**ista-**C**ontrolador (MVC) así como los métodos de mayor importancia. Para facilitar la legibilidad, se separará en cada uno de los grandes bloques de la estructura; en este caso Modelo, Vista, Controlador y Hub

Finalmente, se mostrará una breve guía de como usar nuestra aplicación y un ejemplo de esta.

2. Librerías

En esta sección se explicarán las librerías implementadas y usadas para llevar a cabo el desarrollo de las prácticas y permitir la reutilización de las mismas durante el transcurso de la asignatura.

El principal motivo que nos ha impulsado a implementar dichas librerías es la facilidad que proporcionan para centrarse únicamente en el desarrollo de la práctica en sí; Gracias a que durante el desarrollo de estas librerías se ha priorizado la facilidad de manejo, uso genérico y optimización para añadir el mínimo "overhead" al rendimiento del programa.

2.1. Better swing

Better swing [1] es una librería derivada de Java Swing que permite un desarrollo más amigable y sencillo de interfaces gráficas, al estar inspirado en el desarrollo web (HTML y CSS), concretamente, en el framework de "Bootstrap". Se basa en "JfreeChart" para pintar las gráficas de líneas creando "wrappers" para facilitar su uso y manejabilidad.

2.1.1. Funcionamiento

El funcionamiento del paquete es muy sencillo, básicamente, la idea es crear una o varias ventanas con una configuración deseada y acto seguido ir creando y añadiendo secciones (componentes), donde posteriormente se pueden ir borrando y actualizando los componentes individual o grupalmente. Por ello, este paquete proporciona una serie de métodos para la ventana y las secciones.

¿Cómo funciona la ventana?

Para hacer que la ventana funcione es necesario crear una instancia del objeto `Window`, inicializar la configuración de la misma empleando el método `initConfig` recibiendo

por argumento la ruta donde se encuentra el archivo. En caso contrario se empleará la predeterminada. Finalmente, para que se visualice la ventana se realizará con el método `start` que visualizará una ventana con la configuración indicada anteriormente.

En el caso de querer añadir componentes como una barra de progreso, un botón o derivados, es tan sencillo como crear una sección y emplear el método `addSection`, el cual recibe por parámetro la sección a añadir, el nombre de la sección y la posición de este en la ventana.

¿Cómo funciona la sección?

Las secciones son instancias de la clase `Section` que permiten formar los diferentes componentes de la ventana.

El funcionamiento de una sección es muy sencilla, se trata de instanciar un objeto de la clase `Section` y llamar a algún método de esta clase, pasando los parámetros adecuados, y finalmente añadir la sección a la ventana.

Otros

Adicionalmente, se dispone de una clase llamada `DirectionAndPosition`, que constituye las posibles orientaciones y direcciones que una sección puede tener.

2.1.2. Implementación

Para el desarrollo de esta librería se ha centrado en la simplicidad hacia el usuario final y la eficiencia de código mediante funciones sencillas de emplear y optimizadas para asegurar un mayor rendimiento de la interfaz de usuario. Este paquete se divide en dos principales partes:

Window: Consiste en un conjunto de funciones para la creación y configuración de la ventana.

Section: Consiste en un conjunto de funciones base para la creación de secciones en la ventana.

La implementación de la `Window` consiste en diversas partes: gestión de teclas, configuración, creación y actualización de la ventana y creación de las secciones.

La gestión de las teclas se realiza mediante una clase llamada `KeyActionManager`, que implementa la interfaz `KeyListener`. Esta clase permite gestionar los eventos de teclado de la ventana y se utiliza para la depuración y el desarrollo del programa.

En cuanto, la configuración, creación y actualización de la ventana y la creación de las secciones, se han desarrollado una serie de métodos que envuelven a un conjunto de funciones propias de java swing. Por tanto, con este conjunto de métodos y funciones se obtiene la clase `Window`.

A continuación se explicarán los principales métodos de `Better swing`:

`initConfig` es un método que permite cargar la configuración de la vista y crea el marco de la vista, incluyendo apariencia, posición, tamaño, icono, color de fondo y manejo de eventos de teclado.

`start` permite la visualización de la ventana, haciendo que la ventana sea visible para el usuario.

`stop` actua como wrapper de `JFrame::dispose`.

`addSection` permite añadir una sección a un objeto `Window` indicándole la posición y dirección del panel mediante el conjunto de variables definidas en `DirectionAndPosition`.

`updateSection` permite actualizar una sección indicándole la posición y dirección del panel mediante el conjunto de variables definidas en `DirectionAndPosition`.

`deleteComponent` permite borrar un componente específico de la ventana que se le haya pasado por parámetro al método.

`repaintComponent` permite repintar un componente específico de la ventana que se le haya pasado por parámetro al método.

`repaintAllComponents` repinta todos los componentes de la ventana.

La implementación de la `Section` consiste en diversas partes, las cuales son las siguientes: Los métodos que permiten crear las secciones ya configuradas y las clases en las que se basan los métodos anteriores.

La propia librería contiene muchas más clases y métodos que si el lector desea explorar puede acceder a su documentación a través del código fuente proporcionado en la entrega de la práctica.

2.1.3. Manual de uso

En este apartado se describe como emplear la librería para su correcto funcionamiento.

Para emplear la librería es tan sencillo como crear una instancia de la clase `Window`, llamar al método `initConfig` indicando el fichero de configuración de la ventana, si se desea, en caso contrario se deberá pasar un `null` y se emplearán la configuración por defecto, y finalmente llamar a la función `start` cuando se desee inicializar la ventana.

Es importante inicializar la configuración antes de ejecutar la función `start`, ya que en caso contrario se producirá una excepción. A continuación se muestra un sencillo ejemplo:

```
1 Window view = new Window();
2 view.initConfig("config.json");
3 view.start();
```

Como se observa, la librería permite cargar la configuración e iniciar la ventana independientemente, permitiendo una mayor flexibilidad.

Además, en el caso de querer reiniciar la configuración, cambiar la visibilidad de la ventana o guardar el contenido de esta sin tener que volver a compilar el código, se puede realizar a través de unos atajos de teclado, que son los siguientes:

Q: Cerrar programa.
R: Reiniciar configuración.
V: Cambiar visibilidad.
G: Guardar contenido.

Por ejemplo, al cambiar la configuración y reiniciar la ventana se aplicarán los cambios automáticamente sin compilar de nuevo el código, es decir, se permite el conocido “Hot Reloading”.

En el caso de querer crear una sección y añadirla, se realizaría de la siguiente forma:

```
1  Window view = new Window();
2  Section section = new Section();
3  // Los datos pueden ser de longitudes irregulares
4  long[][] data = { ... };
5  Color chartColors[] = { Color.RED, Color.BLACK };
6  String chartColumnLabels[] = { "Linea 1",
7                                 "Linea 2" };
8  section.createLineChart(labels,
9                          data,
10                         chartColors,
11                         chartColumnLabels,
12                         "Ejemplo Lineas");
13 view.addSection(section,
14                 DirectionAndPosition.POSITION_TOP,
15                 "Chart");
```

En el ejemplo anterior se crearía una gráfica de líneas con dos líneas, una de color rojo y otra negra, con los nombres “Linea 1” y “Linea 2”, con el título “Ejemplos Lineas” y con los puntos del array data.

Finalmente, comentar que hay ilimitadas posibilidades de creación y personalización de componentes con los ya configurados y las posibilidades de crear libremente los propios componentes.

3. Conceptos MVC

El Modelo Vista Controlador (MVC) es un patrón arquitectónico de software que divide una aplicación en tres elementos interconectados, separando la representación interna de la información, como se muestra al usuario y donde se hacen cómputos con esa información respectivamente.

Para esta práctica, se ha decidido modificar parcialmente este patrón al implementar la comunicación entre los módulos mediante un sistema de peticiones. Por ende, encontramos 4 módulos en nuestro MVC: Modelo, Vista, Controlador y Hub. Este último se encargará de gestionar toda la comunicación.

Esta decisión de modificación del MVC ha sido respaldada por la facilidad que obtenemos de escalar el patrón, permitiendo crear tantos módulos como queramos. Esto permitiría que, si en un futuro se quisiera añadir otro controlador en otro lenguaje de programación o encapsular controladores por funcionalidad, el único cambio a efectuar ocurriría en el hub.

A continuación se explicarán cada uno de los módulos y como han sido adaptados y empleados en la práctica.

3.1. Implementación

El Modelo vista controlador (MVC), en esta práctica, ha sido implementado como si fueran aplicaciones diferentes, es decir, cada parte del MVC es independiente de otro elemento de este ya que todos las partes se comunican a trvés de un hub. Esta implementación, trata de imitar una serie de microservicios que se comumican entre ellos a través de una API (application programming interface).

En cuanto a la implementación interna, se trata de una serie de servicios (modelo, vista y controlador) que se conectan a un servidor (hub), donde, entre ellos, se comunican a través de la red local del dispositivo.

¿Cómo hemos conseguido implementar esta idea en Java?

Para implementar el concepto de servicios y servidor se ha realizado a través de los sockets de Java, el problema, de únicamente emplear los sockets, es que los procesos se bloquean cuando envían una petición al servidor hasta que reciben una respuesta, por ello, hemos decidido crear una comunicación asíncrona no bloqueante, es decir, los servicios y el servidor no se pueden bloquear. El concepto empleado es similar al que usa el entorno en tiempo de ejecución multiplataforma NodeJs. Básicamente, un servicio realizará una petición y la respuesta que obtendrá es una promesa diciendo que le llegará una respuesta con los datos que desea, en el caso de que desee obtener información de vuelta, donde le informará de que en algún momento será notificado con los datos solicitados en la petición realizada. Con este planteamiento, se consigue que los procesos, como se ha mencionado anteriormente, no se bloquen y puedan seguir con su ejecución. En el caso, de que se desee esperar a la información se puede realizar uso de una función similar a un "await", en la cual, se encargará de bloquear el proceso hasta que la petición tenga los datos esperados.

La siguiente imagen muestra un diagrama de como está implemtentado el sistema de comunicación entre los elementos:

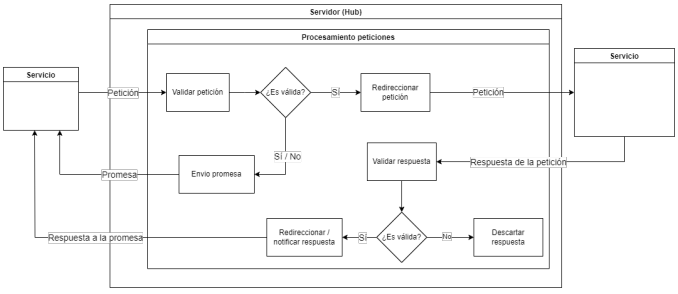

Figura 1. Comunicación entre elementos MVC y hub.

Además, de la implementación comentada anteriormente, los "endpoints"/rutas del servidor se definen a través de un archivo de configuración JSON, el cual, mapea cada petición con los servicios, al igual que su respuestas pertinentes. Esto se puede ver en el siguiente ejemplo:

```
{
  "requestMap": [
    {
      "code": "SEND_GEOPOINTS",
      "services": ["MODEL", "CONTROLLER"]
    }
  ],
  "responseMap": [
    {
      "code": "LOAD_MAP",
      "services": ["VIEW"]
```

```
    },
  ],
}
```

3.2. Hub

El flujo de datos empleado en este patrón de diseño ha sido inspirado en como se comunican los diferentes elementos en un servidor. Es decir, cualquier módulo del MVC deberá realizar una petición al servidor (Hub) y este se encargará de solventar y dar servicio a dicha petición. Se puede ver con mayor claridad como funciona y su implementación en la siguiente figura:

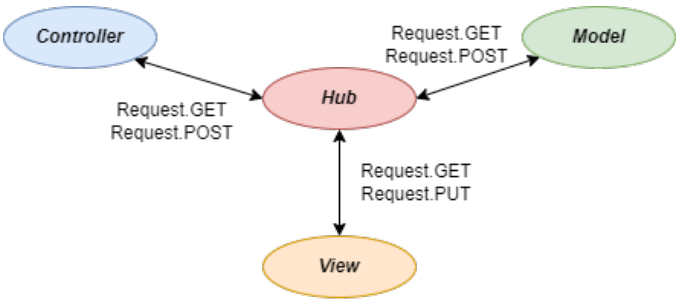

Figura 2. Patrón MVC usado en la práctica.

El Servidor/Hub se encarga de administrar los diferentes tipos de peticiones que llegan de los diferentes componentes. En la figura 2 se observan los diferentes tipos de peticiones posibles que se pueden realizar, en concreto: GET, POST y PUT. En un entorno web, GET sirve para obtener datos, POST para enviar y PUT para actualizar. En el programa los diferentes tipos de peticiones se han traducido en:

GET Realiza un retorno del dato solicitado.
POST Añade un el dato enviado.
PUT Realiza un set del dato enviado.

Para permitir esta implementación, se ha creado la interfaz `Notify` con el método `notifyRequest` el cual deberá estar implementado en todos los módulos del MVC, ya que será el responsable de gestionar las peticiones y asegurar que cada uno de ellos puede realizar sus tareas designadas de manera eficiente y precisa.

3.3. Modelo

El modelo es la representación de los datos que maneja el software. Contiene los mecanismos y la lógica necesaria para acceder a la información y para actualizar el estado del modelo. Gráficamente, se podría expresar de la siguiente manera:

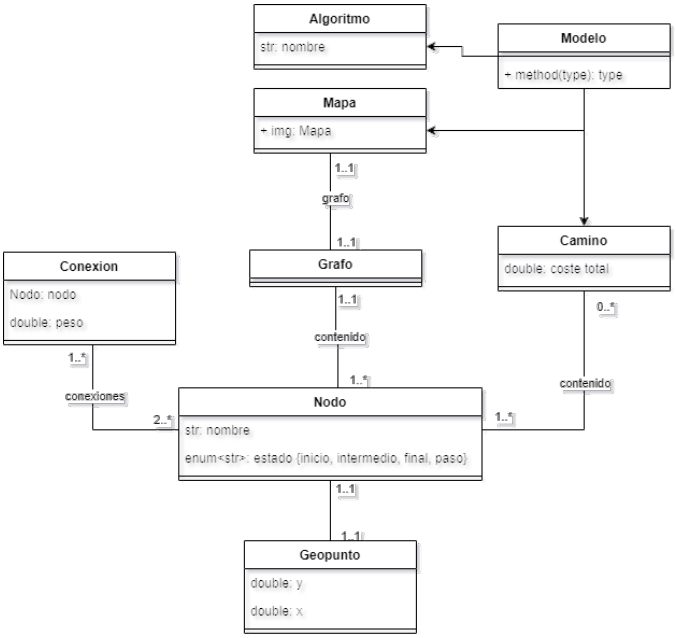

Figura 3. Modelo

Connection Representa la conexión de un nodo con otro nodo y su peso.
GeoPoint Representa un punto en el mapa.
Graph Representa un conjunto de nodos “Nodes”
Map Permite guardar la imagen del mapa con su respectivo grafo
Node Estructura de datos para guardar los puntos, tiene un identificador, un estado “NodeState”, un punto en mapa “GeoPoints”, y unas conexiones determinadas con el resto de nodos vecinos “Connections”
NodeState Representa el estado de un nodo
PairPoint Permite guardar una pareja de “GeoPoints”.
Path Permite guardar una lista de nodos y su coste

Finalmente, al ser un módulo de nuestro MVC, implementa la interfaz `Notify` y su

método `notifyRequest` que le permite recibir notificaciones de los otros módulos del MVC.

3.4. Vista

La vista contiene los componentes para representar la interfaz del usuario (IU) del programa y las herramientas con las cuales el usuario puede interactuar con los datos de la aplicación. Adicionalmente, la vista se encarga de recibir e interpretar adecuadamente los datos obtenidos del modelo. Cabe mencionar, que al igual que el resto de componentes del MVC, la clase View implementa la interfaz "Notify", la cual permite la comunicación con el resto de elementos.

`loadContent` es el encargado de cargar el contenido inicial en la ventana. Para esta práctica, carga los siguientes elementos semánticos:

`menu`, función que crea y configura el menú de utilidades de la ventana principal. En esta, se incluyen las siguientes opciones: Abrir ventanas de estadísticas y salir del programa.

`body`, función que crea, configura y actualiza la gráfica de la ventana principal. Se trata de un mapa en el que se pueden seleccionar los puntos o poblaciones, estos puntos están conectados por una serie de caminos. Para ello, se han usado las clases JFreeChart, y una clase que hemos creado para añadir elementos sobre el mapa, MapPlot.

`sidebar`, función que crea y configura la barra de opciones de la derecha de la interfaz principal. Esta permite al usuario interactuar y configurar el entorno de ejecución de los algoritmos y seleccionar los mapas configurados. Además, se ha añadido una pequeña ventana de logs para saber que está ocurriendo en todo momento.

`footer`, función que crea y configura los botones para manipular los puntos seleccionados sobre el mapa. El botón Deshacer, elimina el último punto añadido. El botón Rehacer, añade el último punto borrado. El botón Limpiar elimina todos los datos añadidos sobre el mapa. Por último, el botón Confirmar ejecuta el algoritmo y encuentra un camino óptimo entre los puntos seleccionados.

Adicionalmente, a la vista principal, esta permite desplegar dos ventanas más. Una ventana muestra a tiempo real, el uso y consumo de la memoria de la Java virtual machine (JVM) y la otra ventana muestra las estadísticas de la ejecución de los algoritmos además de su comparación. A continuación, se muestran dos imágenes (4 y 5) de como se verían las ventanas al iniciarlas junto a una breve explicación de la misma.

Finalmente, al ser un módulo de nuestro MVC, implementa la interfaz `Notify` y su método `notifyRequest` que le permite recibir notificaciones de los otros módulos del MVC.

3.4.1. Estadísticas JVM

Este apartado de la vista principal, es el encargado de enseñar a tiempo real las estadísticas de la máquina virutal de java. Concretamente, se actualiza cada 0.5s a apartir de los datos obtenidos de la clase de java "Runtime" y muestra la memoria libre, la memoria total y el uso de esta en una gráfica de lineas, donde el eje x es instante en el tiempo que se han obtenido los datos y el eje y su valor. Todos los datos de la memoría obtenidos están en MB.

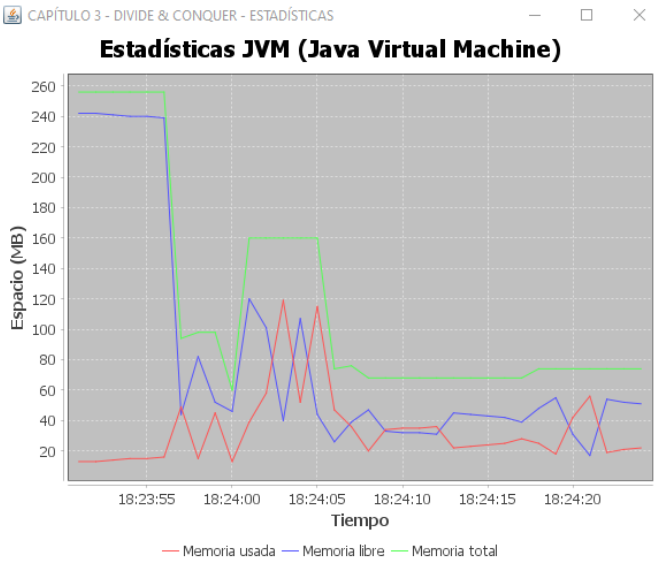

Figura 4. Interfaz estadísticas JVM

3.4.2. Estadísticas Algoritmos

Este apartado de la vista principal, es el encargado de enseñar las estadísticas de la ejecución de los algoritmos, $Dijkstra$.... Estas estadísticas incluyen dos gráficas, las estadísticas por iteración y las estadísticas globales, ambas de la ejecución del algoritmo con la distancia seleccionada. En la gráfica por iteración se encuentran los datos obtenidos de cada ejecución por separado del algoritmo, concretamente, el número de iteraciones que ha realido esa ejecución, cuantos nodos se ha visitado en cada iteración y la memoria empleada. Y en la otra gráfica, se muestra el tiempo que ha tardado en encontrar la solución, los nodos visitados de media, la memoria media empleada, el número de iteraciones medio por ejecución del algoritmo y la cantidad de nodos que tiene el grafo.

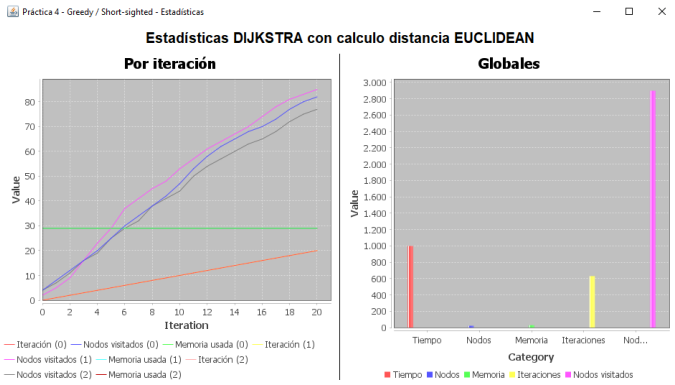

Figura 5. Interfaz estadísticas algortimos

Como se ha podido ver anteriormente en la imagen (5), los datos de las estadísticas están representados con un diagrama de barras y un diagrama de lineas, donde el eje x es la categoría a la que pertenecen y el eje y el valor correspondiente, en la caso de la segunda gráfica. En el caso de la primera gráfica, el eje X es la iteración y el eje y el valor.

3.5. Controlador

El controlador en un MVC es el responsable de recibir y procesar la entrada del usuario y actualizar el modelo. En esta práctica, dicha responsabilidad es la de verificar el input del usuario, transformar el los datos de los mapas a un conjunto de objetos que el modelo pueda entender y ejecutar los algoritmos de "pathfinding" para el conjunto de puntos definidos por el usuario.

3.5.1. Diseño

El controlador se separa en tres grandes secciones como ya se ha mencionado previamente.

El usuario puede, bajo su libre albedrío, interactuar en cualquier punto del mapa. Por ende, será necesario procesar la entrada del usuario. Así pues, cada vez que el usuario interactúe con el mapa, se notifica al controlador que determine si está cerca de algún punto del grafo. Para ello, buscamos el primer nodo cuya distancia entre él y el punto interactuado con el usuario sea menor que una distancia predefinida x. De esta manera, podemos controlar el tamaño virtual de cada nodo. Una vez detectado (si es que se detecta) se devuelve tanto a la vista como al modelo, para visualizarlo y guardarlo respectivamente.

Los datos de los mapas se guardan en formato JSON, proporcionando las conexiones y posiciones de cada uno de los nodos, entre otros datos. Mediante el uso de la librería GSON, transformamos todos los datos contenidos en él y generamos nuestro mapa. Posteriormente, se entrega al modelo para mantener la instancia actual.

Finalmente, el controlador permite ejecutar los algoritmos de Dijkstra y Greedy aplicando una de las siguientes heuristicas para definir la distancia entre dos nodos:

- EUCLIDEAN,
- MANHATTAN,
- CHEBYSHEV,
- COSINE,
- MINKOWSKI,
- HAVERSINE

Creando un conjunto de doce posibles ejecuciones.

3.5.2. Algoritmos

3.5.2.1. Dijkstra: El algoritmo de Dijkstra es un popular algoritmo de determinación de caminos mínimos entre dos puntos de un grafo. Se aplica manteniendo un conjunto de nodos no visitados y un conjunto de distancias tentativas a estos. Seguidamente, selecciona el nodo con la menor distancia tentativa y examina los nodos conectados a él. Por cada uno de estos nodos "vecinos", calcula la distancia a ese nodo sumando la distancia al nodo actual y la distancia entre el nodo actual y él. Si la distancia calculada es menor a la distancia tentativa del nodo "vecino", se actualiza su distancia y se le asigna el actual nodo como su previo. De esta manera, podemos saber cuál es el recorrido inverso desde cualquier nodo al inicio. Este proceso se ejecuta hasta que se encuentra el nodo objetivo.

```
1  function Dijkstra(Graph, source, target):
2
3      for each vertex v in Graph.Vertices:
4          dist[v] := INFINITY
5          prev[v] := UNDEFINED
6          add v to Q
7      dist[source] := 0
8
9      while Q is not empty:
10         u := vertex in Q with min dist[u]
11         if u is target:
12             exit(dist[], prev[])
13
14         remove u from Q
15
16         for each neighbor v of u still in Q:
17             alt := dist[u] + Graph.Edges(u, v)
18             if alt < dist[v]:
19                 dist[v] := alt
20                 prev[v] := u
21                 add v to Q
22
23     exit(solution_not_found)
```

Con respecto a la complejidad temporal podemos asegurar que es $O((V + A)logV)$ donde V son los vértices y A las aristas; Siendo asintóticamente $O(nlogn)$. Por cada nodo el algoritmo debe examinar todos sus "vecinos" costando $O(A)$. Al añadir todos los elementos a la cola de prioridad como se ve en la `línea 3` cuesta $O(V)$ y la extracción del mínimo elemento cuesta $O(logV)$. Finalmente, si componemos todos los costes aseguramos que el coste es de $O((V + A)logV)$.

Con respecto al coste espacial, debido a que tenemos que generar dos listas de elementos `previos` y `distancias`, obtenemos $O(V)$.

3.5.2.2. Greedy (Monticulo de fibonacci): Este algoritmo utiliza dos búsquedas heurísticas, una desde el nodo inicial y otra desde el nodo objetivo, que se ejecutan simultáneamente. El objetivo es encontrar un camino óptimo desde el nodo inicial hasta el nodo objetivo del grafo, utilizando una función heurística (el peso de cada arista).

La función heurística utilizada en este algoritmo se llama "heurística admisible", lo que significa que siempre subestima el costo real para llegar al objetivo. En este caso, se utiliza una función de costo heurístico que se define en la clase "Node" y se actualiza durante el proceso de búsqueda.

El algoritmo comienza inicializando dos colas de prioridad, que se utilizan para almacenar los nodos visitados en las dos búsquedas simultáneas. Cada cola es ordenada por el valor de la función heurística de cada nodo con respecto al nodo objetivo, es decir, en la primera cola con respecto al nodo final y viceversa en la segunda cola.

Luego, se inicializan cuatro mapas, que se utilizan para almacenar los padres de cada nodo visitado en las dos búsquedas simultáneas, y los otros, que se emplean para almacenar las distancias desde el nodo inicial y el nodo objetivo a cada nodo visitado en las dos búsquedas simultáneas.

A continuación, el algoritmo ejecuta un bucle que se ejecutará mientras ambas colas no estén vacías y no se haya encontrado un nodo en común entre las dos búsquedas. En cada iteración, el algoritmo extrae el nodo de cada cola con el menor valor de la función heurística y los compara para buscar una intersección. Si se encuentra un nodo común, se almacena en el nodo de encuentro, se detiene, él finaliza el bucle y se procede a construir la ruta óptima. En caso contrario, se expanden los nodos

vecinos de ambos nodos extraídos de las colas y se actualizan los mapas.

Finalmente, después de encontrar el nodo de encuentro, se construye la ruta óptima utilizando los mapas para recorrer el camino de ambos extremos hacia el nodo de reunión, donde, por último, se unen los dos caminos para obtener el camino resultado.

4. MANUAL DE USUARIO

Para el correcto uso de la aplicación, el conjunto de acciones que se pueden realizar en dicho programa serán definidas a continuación. También, la distribución de las secciones de la interfaz junto a su explicación y funcionalidad.

Cuando se ejecute el programa por primera vez en la pantalla se debe mostrar la siguiente interfaz de usuario:

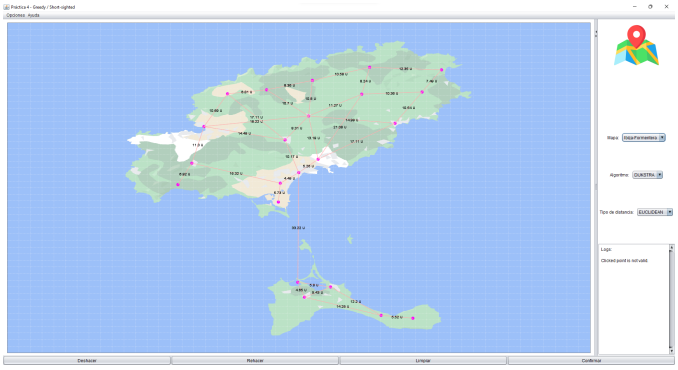

Figura 6. Interfaz de usuario.

4.1. Menu

El menú de la aplicación se trata de la barra que se sitúa debajo del marco superior de la ventana. En esta se puede encontrar un conjunto de opciones para que el usuario pueda interactuar con la aplicación, modificar y analizar el comportamiento de esta. En concreto, se encuentran las opciones de "Opciones" y "Ayuda".

En la primera se sitúan las acciones para salir del programa y para borrar y/o resetear datos e iniciar las ventanas de estadísticas explicadas anteriormente en los apartados 3.4.1 y 3.4.2.

En la otra opción se encuentra un menú desplegable con un manual de usuario con la explicación del funcionamiento de la aplicación.

4.2. Main

El "Main" es el bloque principal de la vista, donde se representará el mapa con las poblaciones, aquí el usuario podrá seleccionar los puntos/poblaciones para encontrar el camino mínimo. El usuario deberá clicar de manera precisa en el punto, debido a que éste tiene un radio de detección del ratón determinado. De esta manera, al clicar en el mapa, no se seleccionará ningún punto de manera errónea. No podrá clicar dos veces sobre el mismo punto.

4.3. Sidebar

El "Sidebar" contiene un conjunto de opciones para modificar los datos y el modo de ejecución de la aplicación. A continuación, se explicará cada una de estas opciones y su función.

En primer lugar, se encuentra la opción para seleccionar el "Mapa". En esta, se puede escoger el mapa que queremos representar. En esta práctica, se han implementado los siguientes mapas:

- Ibiza-Formentera
- Ibiza
- Formentera

En segundo lugar, se halla la opción para elegir el "Algoritmo". En esta opción, se tiene la opción cambiar el algoritmo a aplicar sobre los puntos seleccionados. De manera que el resultado podría cambiar dependiendo del algoritmo ejecutado. Los algoritmos implementados son:

- Dijkstra
- Greedy

En tercer lugar, se encuentra el "Tipo de distancia". Esta opción permite definir una heurística para calcular la distancia. Las heurísticas implementadas son:

- Euclidean
- Manhattan
- Chebyshev
- Cosine
- Minkowski
- Haversine

En tercer lugar, se encuentra una ventana de "logs". En esta ventana el usuario tendrá una noción de lo que está ocurriendo en el programa, indicando los puntos seleccionados y si se ha seleccionado fuera del mapa.

4.4. Footer

En la sección "Footer", se hallan cuatro botones para interactuar con los puntos seleccionados sobre el mapa. En primer lugar, empezando por la izquierda de la interfaz, tenemos el botón "Deshacer". Cuando el botón sea pulsado, el último punto añadido al mapa será borrado.

En segundo lugar, tenemos el botón "Rehacer". Cuando se pulse este botón, el último punto eliminado será añadido sobre el mapa de nuevo. En el momento en el que el usuario añada un nuevo punto después de haber pulsado "Deshacer", no podrá rehacer ningún punto.

En tercer lugar, se encuentra el botón "Limpiar", con este botón desaparecerán todos los elementos añadidos sobre el mapa tras haber iniciado el programa y volverá a su estado inicial. Este botón es útil para borrar todos los puntos seleccionados o la solución del camino.

Por último, se halla el botón "Confirmar". Este botón envía al Controlador los puntos seleccionados y ejecuta el algoritmo, una vez ejecutado se representará el camino en el mapa, con una línea de color rojo.

4.5. Ejemplo ejecución

Al iniciar la aplicación se mostrará una interfaz como la expuesta en la imagen 6. Para poder iniciar la ejecución del algoritmo, es necesario seleccionar dos o más puntos sobre el mapa, en el caso de equivocarse podrá "Deshacer" y "Rehacer" los puntos. Además, de poder borrarlos todos con el botón "Limpiar". Al clicar en "Confirmar", si hay dos o más puntos en el mapa, se ejecutará el algoritmo y se pintará sobre el mapa el camino mínimo entre los puntos seleccionados. A continuación, la siguiente imagen, muestra un ejemplo de la interfaz en mitad de la ejecución:

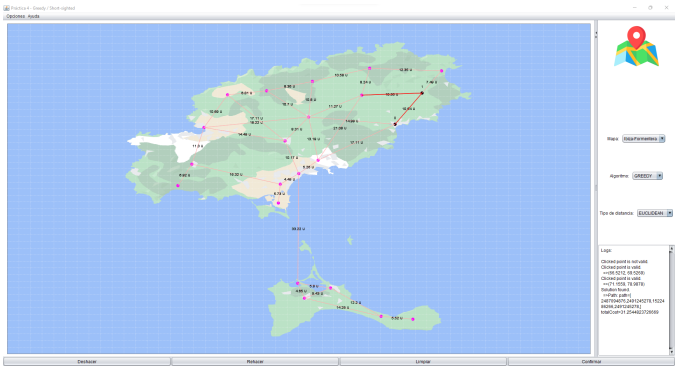

Figura 7. Interfaz de usuario en ejecución.

A partir de aquí, el usuario puede reiniciar la ejecución para ejecutar nuevamente el algoritmo, borrar las soluciones obtenidas, ejecutar otros algoritmos, cambiar el tipo de distancia, cambiar el mapa y ver estadísticas de la ejecución, ver apartados 3.4.1 y 3.4.2.

5. Conclusión

Teniendo en cuenta el desarrollo expuesto, la aplicación permite generar un punto de referencia de un conjunto de algoritmos de diferente complejidad asintótica mediante una representación gráfica e interactiva de la media de los tiempos de ejecución a través de unos parámetros de entrada definidos por el usuario. Adicionalmente, se ha implementado mediante una versión modificada del patrón de diseño de software **M**odelo **V**ista **C**ontrolador (MVC) añadiendo un cuatro módulos que permite la comunicación entre los diferentes elementos de

este mediante peticiones. Esta modificación ha sido fuertemente inspirada en el diseño de un servidor, lo que permite gran flexibilidad y escalabilidad de desarrollo, al poder interceptar y gestionar las peticiones a gusto del desarrollador.

Además, se ha hecho énfasis en el uso de las librerías creadas por los miembros del grupo para facilitar el trabajo y la reutilización de código en futuras prácticas. Reiterar que el principal motivo que nos ha impulsado a implementar dichas librerías es la facilidad que proporcionan para centrarse únicamente en el desarrollo de la práctica en sí; Gracias a que durante el desarrollo de estas librerías se ha priorizado la facilidad de manejo, uso genérico y optimización para añadir el mínimo "overhead" al rendimiento del programa.

Concluir que, con respecto al ámbito académico, este proyecto nos ha permitido consolidar el concepto de patrón de diseño MVC gracias a un primer proceso de investigación y discusión del diseño a implementar entre los integrantes del grupo.

6. Distribución del trabajo realizado

El trabajo realizado por cada miembro de la práctica ha sido el siguiente:

- Alejandro Rodríguez:
  - Documentación
  - Diseño del proyecto
  - Desarrollador de funcionalidades extra de la IU.
  - Desarrollador del "frontend"
  - Desarrollador principal del footer.
  - Desarrollador de la plantilla base del proyecto
- Rubén Palmer:
  - Documentación
  - Diseño del proyecto
  - Principal desarrollador del "backend" de la aplicación
  - Desarrollador general de la aplicación
  - Desarrollador del "frontend"
  - Diseñador de la implementación del patrón MVC de la aplicación
  - Desarrollador de la plantilla base del proyecto
  - Desarrollador de la plantilla de la documentación
- Sergi Mayol:
  - Documentación
  - Diseño del proyecto
  - Desarrollador de la librería `Better Swing`
  - Principal desarrollador del "frontend" de la aplicación
  - Desarrollador general de la aplicación
  - Desarrollador del "backend" de la aplicación
  - Diseñador de la implementación del patrón MVC de la aplicación
  - Desarrollador de la plantilla base del proyecto

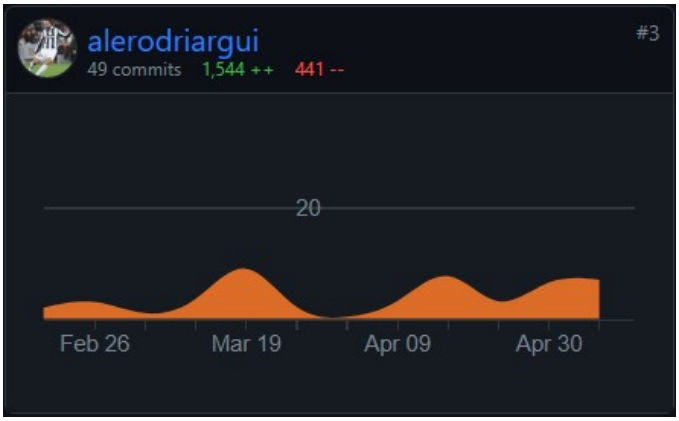

Figura 8. Contribución en el proyecto de Alejandro Rodríguez

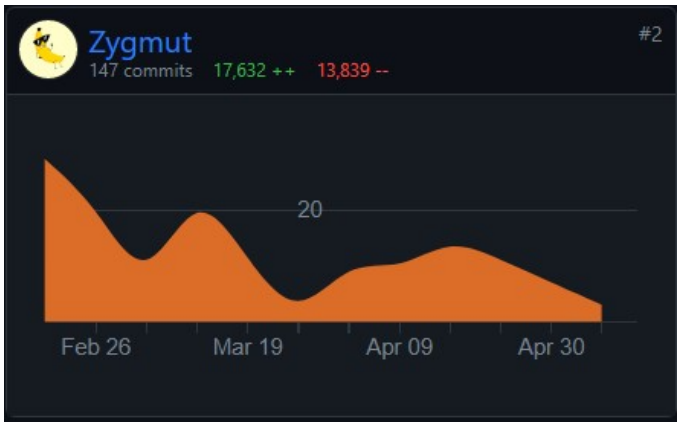

Figura 9. Contribución en el proyecto de Rubén Palmer

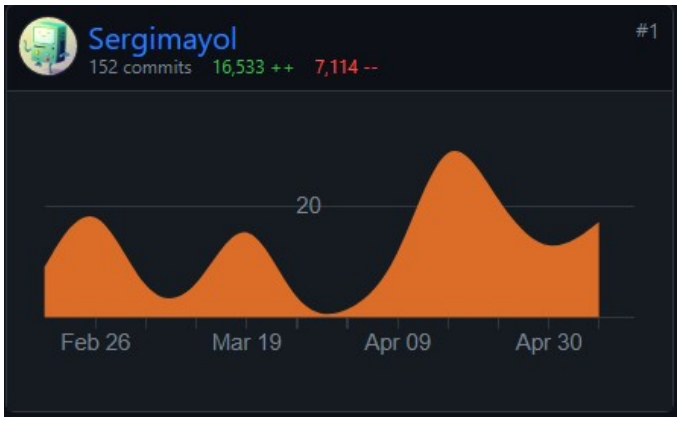

Figura 10. Contribución en el proyecto de Sergi Mayol

RECONOCIMIENTOS

- Se agradece la colaboración del Dr. Miquel Mascaró Portells en la resolución de dudas y supervisión del proyecto.

REFERENCIAS

[1] Better Swing, *An easy way to develop java GUI apps*, V0.0.3. Ver documentación completa aquí.
[2] Graphical engine, *How to develop a graphical engine from scratch*. Ver enlace aquí.
[3] Dijkstra's algorithm, *how does it work and possible implementations*. Ver enlace aquí.
[4] Overpass turbo, *An easy way of a data mining web tool for OpenStreetMap (OSM)*. Ver enlace aquí.
[5] Google json (Gson), *Java serialization/deserialization library to convert Java Objects into JSON and back*. Ver enlace aquí.
[6] Distancia Euclidiana, *The Euclidean distance between two points*. Ver enlace aquí.
[7] Distancia de Manhattan, *The Manhattan distance between two points*. Ver enlace aquí.
[8] Distancia de Chebyshev, *The Chebyshev distance between two points*. Ver enlace aquí.
[9] Similitud coseno, *The Cosine similitary distance between two points*. Ver enlace aquí.
[10] Similitud coseno, *The Cosine similitary distance between two points*. Ver enlace aquí.
[11] Distancia de Minkowski, *The Minkowski distance between two points*. Ver enlace aquí.
[12] Distancia de Haversine, *The Haversine distance between two points*. Ver enlace aquí.
[13] Node.js Architecture, *Architecture of a single thread cross-platform, open-source server environment*aquí.