Algoritmos Ávidos

Martin Mitrovski Delov, Josep Antoni Naranjo Llompart, Sergi Moreno Pérez, Pau Rosado Muñoz

Resumen — El ejercicio consiste en desarrollar un programa capaz de encontrar el camino mínimo entre dos pueblos pasando de entre medias por un tercero. El algoritmo que hará este cálculo está basado en una metodología iterativa de tipo Dijkstra, debido a que se recorren todos los estados, decimos que estamos hablando de una estrategia ávida.

Por motivos de tamaño del fichero de la práctica, hemos decidido subir el vídeo a YouTube y no incluirlo en el archivo comprimido entregado. El enlace al vídeo de la práctica en YouTube es el siguiente:
https://www.youtube.com/watch?v=teEllaK_EnQ

I. INTRODUCCIÓN

En este artículo se explicará cómo se ha realizado la cuarta práctica de la asignatura Algoritmos Avanzados. Se desarrollarán algunos conceptos teóricos expuestos en el cuarto tema y también se describirá la implementación de una aplicación que calcula la ruta mínima, siguiendo ciertos criterios que pueden variar entre ejecuciones, entre dos pueblos pasando por un tercero de un mapa, siendo el mapa por defecto el de Ibiza y Formentera. Para ello hacemos uso de lo algoritmos ávidos, en concreto uno de tipo Dijkstra sobre un grafo de topología no dirigida.

Los puntos por tratar son los siguientes: el entorno de programación utilizado, los conceptos de MVC, el patrón por eventos, los algoritmos voraces o ávidos y la implementación del MVC para el desarrollo de nuestra aplicación junto a todos los matices necesarios.

Por parte del concepto MVC, explicaremos qué es y por qué es el método de diseño que utilizaremos durante todo el curso para la implementación de las prácticas.

Explicaremos en detalle que son los algoritmos voraces o ávidos, cuando se deberían usar, los tipos como por ejemplo Prim, Kruskal, Dijkstra, montículo de Fibonacci y Huffman y por último los ejercicios más característicos como el cálculo de la distancia mínima.

Una vez explicadas estas secciones teóricas, procederemos a mostrar la implementación de la aplicación usando el patrón de diseño MVC y aplicando la técnica de un algoritmo ávido de búsqueda del camino mínimo siguiendo el método de Dijkstra para el cálculo. Explicaremos cada una de las partes, junto a los conceptos de código más importantes. Cabe destacar que haremos bastante hincapié en el nivel de abstracción que se ha perseguido tener en la implementación y comunicación entre los diferentes componentes de la arquitectura. Primero explicaremos el modelo de datos, después el controlador y finalmente la vista o interfaz gráfica.

Del apartado gráfico, explicaremos algunas características para asegurar la consistencia del programa.

Para acabar habrá un apartado de conclusiones dónde se realizará una breve reflexión sobre los resultados obtenidos y sobre el trabajo dedicado al proyecto.

II. ENTORNO DE PROGRAMACIÓN

En nuestra práctica usamos el entorno de programación NetBeans, entorno de desarrollo integrado libre, orientado principalmente al desarrollo de aplicaciones Java. La plataforma NetBeans permite el desarrollo de aplicaciones estructuradas mediante un conjunto de componentes denominados “módulos”. La construcción de aplicaciones a partir de módulos permite que estas sean extendidas agregándoles nuevos componentes. Como los módulos pueden ser desarrollados independientemente, las aplicaciones implementadas en NetBeans pueden ser escaladas fácilmente por otros desarrolladores de *software*.

III. MODELO VISTA CONTROLADOR (MVC)

El modelo-vista-controlador (MVC) [1] es una arquitectura de software que separa los datos, la interfaz de usuario y la lógica de control en tres componentes distintos. El modelo representa los datos, la vista muestra los datos al usuario de forma visualmente atractiva y el controlador actúa como intermediario entre el modelo y la vista, gestionando el flujo de información entre ellos. El flujo de una aplicación que usa el MVC comienza con la interacción del usuario a través de la interfaz de usuario, entonces el controlador recibe la acción del usuario y actualiza el modelo acorde al nuevo estado del programa, seguidamente la vista utiliza los datos del modelo para actualizar la interfaz.

Existen tres formas básicas de implementación del MVC: distribuido, centralizado y centralizado en el controlador. El uso de este patrón de diseño por capas tiene varias ventajas, como la división del trabajo, la reutilización de código, la flexibilidad y escalabilidad que supone, así como la facilidad de realizar el *testing* de la aplicación. Sin embargo, también

tiene algunas desventajas, como la complejidad adicional que puede presentar debido a que requiere más código y esfuerzo para mantener la separación de tareas entre los componentes.

En la Figura 1 aparece el esquema habitual del patrón de diseño MVC.

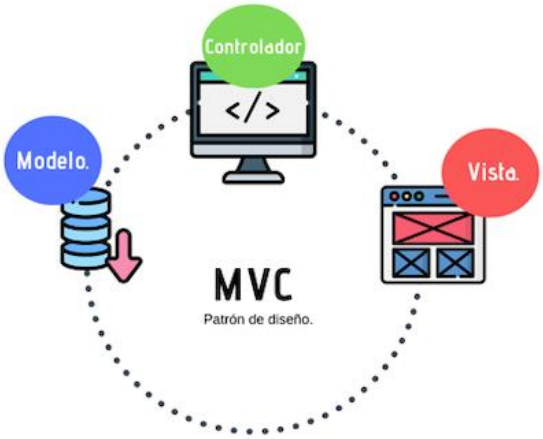

Figura 1: Modelo Vista Controlador

IV. PATRÓN POR EVENTOS

La arquitectura orientada a eventos o *Event-driven architecture* (EDA) es una forma de diseñar software que se enfoca en los eventos que ocurren dentro del sistema [2]. Los eventos son ocurrencias notificadas, como cuando un usuario hace clic en un botón. En una arquitectura EDA, los componentes del sistema están diseñados para reaccionar a estos eventos en tiempo real. Los componentes se comunican entre sí mediante eventos como se muestra en la Figura 2, propagándose a los componentes relevantes que responderán al evento adecuadamente.

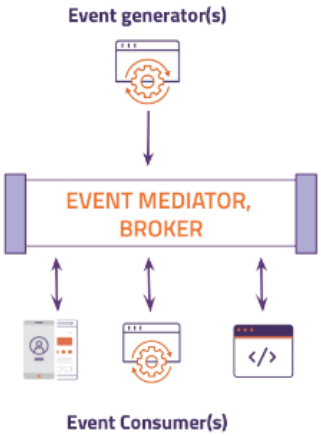

Figura 2: Patrón por eventos

La arquitectura de eventos es especialmente útil [3] en sistemas que requieren procesamiento en tiempo real, escalabilidad y flexibilidad, como redes de sensores, comercio financiero, sitios web de comercio electrónico y redes sociales. Los sistemas basados en esta arquitectura pueden reaccionar rápidamente a cambios y son tolerantes a fallos debido al bajo acoplamiento entre los componentes.

Sin embargo, esta arquitectura también puede tener desventajas, como la complejidad y dificultad de depuración debido a la naturaleza distribuida del sistema y la necesidad de sincronización de los eventos. También se deben tener en cuenta posibles problemas de latencia, ya que los eventos pueden tardar un cierto tiempo en propagarse a través de los componentes.

V. ALGORITMOS ÁVIDOS

Los algoritmos ávidos (también conocidos como "greedy" en inglés) [4] son un tipo de algoritmo de optimización que busca resolver problemas complejos mediante la toma de decisiones locales en cada etapa, esperando que estas decisiones a nivel local lleven a una solución óptima a nivel global.

Se trata de la primera técnica avanzada pura la cual tiene una fácil implementación, pero de difícil diseño.

En un algoritmo ávido, su enfoque para solucionar los problemas es seleccionando la mejor opción disponible en ese momento de forma local, sin tener en cuenta el impacto que esta decisión pueda tener en las etapas futuras. El algoritmo nunca revierte la decisión anterior, incluso si la elección es incorrecta. Funciona en un enfoque de arriba hacia abajo. Por lo tanto, un algoritmo ávido no garantiza una solución óptima en todos los casos, pero suele ser eficiente y rápido en muchos problemas prácticos.

Suelen ser indiciados para los problemas de optimización y un ejemplo común de algoritmo ávido es el algoritmo de selección de la moneda, donde se desea dar cambio utilizando la menor cantidad posible de monedas. En este algoritmo, se selecciona siempre la moneda más grande posible en cada etapa hasta llegar al monto deseado, lo cual en muchos casos conduce a una solución óptima.

En resumen, los algoritmos ávidos son muy rápidos dentro del coste computacional asintótico natural del problema. Tienen un enfoque de resolución de problemas en el cual se toma la mejor decisión local en cada etapa sin tener en cuenta las consecuencias a largo plazo. puede conducir a soluciones subóptimas en algunos casos, pero puede ser muy eficiente y rápido en otros. Y cabe destacar que una vez diseñados habrá que comprobar la corrección y estado de las soluciones que éste encuentra.

V.I ¿CUÁNDO PODEMOS USAR LOS ALGORITMOS ÁVIDOS?

Podemos determinar si el algoritmo se puede usar con cualquier problema si el problema tiene las siguientes propiedades:

1. Propiedad de elección ávida: si podemos encontrar una solución óptima al problema eligiendo la mejor opción en cada momento sin reconsiderar los pasos anteriores una vez seleccionados, en ese caso, podemos hacer uso de los algoritmos ávidos.

2. Subestructura óptima: si la solución general óptima del problema corresponde a la solución óptima de sus subproblemas, entonces podemos hacer uso de los algoritmos ávidos.

3. El problema tiene una estructura matemática que permite determinar cuál es la mejor opción local en cada etapa.

Algunos ejemplos de problemas que pueden resolverse mediante algoritmos ávidos y de los cuáles uno de ellos pertenece al ejercicio práctico de esta práctica son:

- Problemas de búsqueda de caminos más cortos en un grafo, como el algoritmo de Dijkstra. Este lo explicaremos más en detalle al explicar nuestra implementación de la práctica.
- Problemas de selección de objetos, como el problema de la mochila fraccionaria.

En resumen, los algoritmos ávidos son útiles para resolver problemas de optimización que tienen una estructura matemática adecuada y se pueden descomponer en subproblemas más pequeños, donde la selección de la mejor opción local en cada etapa lleva a una solución óptima global.

V.II ESQUEMA GENERAL

El esquema general de un algoritmo ávido consta de los siguientes pasos:

1. Definir el problema: Se debe identificar el problema como un problema de optimización en el que debemos encontrar la mejor solución respecto a otro tipo de posibles soluciones.
2. Seleccionar una solución inicial: Para empezar, el conjunto de soluciones (que contiene las respuestas) está vacío. El algoritmo ávido comienza con una solución inicial que puede ser construida de forma aleatoria o siguiendo un criterio determinado.
3. Seleccionar la mejor opción local: En cada etapa, se examina todas las opciones posibles y se selecciona la mejor opción local según un criterio de selección definido previamente. Se agrega el elemento al conjunto de soluciones.
4. Actualizar la solución global: Una vez que se ha seleccionado la mejor opción local, se actualiza la solución global y se continúa con la siguiente etapa.
5. Verificar la solución: Si el conjunto de soluciones es factible, se mantiene el elemento actual. De lo contrario, el artículo se rechaza y no se vuelve a considerar nunca más.
6. Refinar la solución: Si la solución no es óptima, se pueden aplicar técnicas de refinamiento para mejorarla.

V.III DEMOSTRACIÓN DE PORQUE NO SIEMPRE PRODUCE LA SOLUCIÓN ÓPTIMA

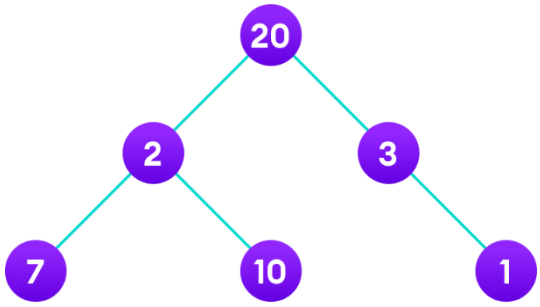

Figura 3: Grafo demostración

Como mencionamos previamente los algoritmos ávidos no siempre producen la solución óptima y esto es su mayor desventaja. [5]

Para ello veremos un ejemplo en el cual queremos encontrar el camino más largo del grafo de la figura 3 desde la raíz hasta las hojas haciendo uso de un algoritmo ávido.

- Primero comencemos con el nodo raíz 20. El peso del hijo derecho es 3 y el del izquierdo es 2.

- Como lo que buscamos es el camino más largo el algoritmo elegirá el 3.

- Finalmente, como solo tiene un hijo y su peso es 1, elegimos este y el resultado final será: 20 + 3 + 1 = 24.

Sin embargo, la solución que se obtiene no es la solución óptima. Hay otro camino que tiene más peso como se muestra en la Figura 4.

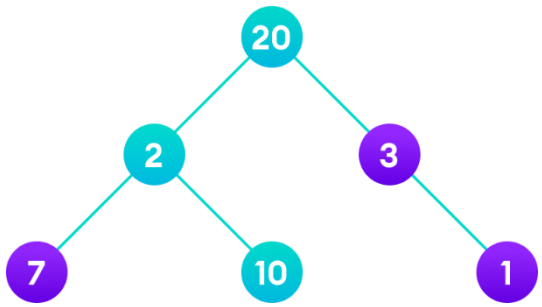

Figura 4: Solución más larga (20 + 2 + 10 = 32)

VI. IMPLEMENTACIÓN MVC

Como ya hemos comentado anteriormente en este informe existen tres tipos de variaciones según respecta a la implementación del patrón "Modelo Vista Controlador". En esta práctica, debido a la facilidad que ofrece a la hora de gestionar los distintos módulos, así como para facilitar la compresión de los distintos componentes sin mezclar funcionalidades, se ha optado por desarrollar el programa siguiendo un esquema centralizado.

Para refrescar cabe comentar que este esquema persigue que la gestión de la comunicación entre las partes implicadas pase por un intermediario común. Este centro de control se responsabiliza también del ciclo de vida de los componentes.

Este "manager" central se ha implementado en la clase **Main,** a partir de la cual empieza a ejecutarse el software una vez se inicia el ejecutable.

El elemento que permite que la vista, el controlador y el modelo se puedan comunicar entre sí es una interfaz llamada **EventListener** que obliga a implementar la función "**notify**". Mediante el uso de la clase abstracta **Event**, el **main** podrá distinguir cuál de los componentes ha notificado el evento y redireccionarlo según corresponda. Esta solución es posible ya que cada componente implementa la misma interfaz, por lo que también deben implementar su propia versión de "**notify**", la cuál será llamada desde el **main** cuando éste decida que un evento debe ser tratado por la correspondiente parte. Por tanto, en el **main** se tendrá una instancia de cada componente de la arquitectura.

La arquitectura descrita permite una gran versatilidad a la hora de implementar los distintos elementos por separado debido a que existe un gran desacople en el sistema de comunicación. Esto ayuda a la hora de desarrollar un módulo sin tener que pensar en cómo afectarán las nuevas funcionalidades en la lógica global.

Se debe mencionar que se han creado tres ficheros '.ltim' para generar los mapas disponibles. Para ello, se hace uso de la clase enumerada Map que contiene los valores:

- PITIUSES
- MENORCA
- MALLORCA

Siendo 'MALLORCA' el mapa con mayor número de nodos y de arestas, 'MENORCA' el que menos y 'PITIUSES' como término medio. Para todos ellos, las arestas son bidireccionales, en representación al usual tipo de carreteras entre poblaciones, las cuáles normalmente son de doble sentido, a pesar de que para ambas direcciones el peso de la ruta no tiene por qué ser el mismo.

VI.I MODEL

El modelo es una parte fundamental de cualquier aplicación, ya que es el componente encargado de almacenar y gestionar todos los datos que serán escritos o leídos por los usuarios, tanto de manera directa como indirecta. En otras palabras, el modelo es el lugar donde se guardan y organizan los datos que serán utilizados para realizar diferentes operaciones y análisis en la aplicación.

En el caso específico de nuestra aplicación, el modelo sería un mapa, definido en un fichero en dónde aparecen las distintas poblaciones contenidas en dicho mapa junto con su ubicación en el mismo, así como las rutas que las unen y las características propias de cada ruta. El modelo entonces estará compuesto por la lista de poblaciones y por la lista de rutas que conforman el mapa.

ModelEvent

La clase ModelEvent hereda de **Event** y se utiliza para que el **main** notifique al Model cuando se produzca un evento que este deba gestionar. Además, en él aparece un enumerado ModelEventType para diferenciar el tipo de ModelEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro del modelo.

El ModelEventType compone los eventos de modelo:

- UPDATE_ORIGIN
- UPDATE_MIDDLE
- UPDATE_DESTINATION
- START
- RESET

Los atributos de ModelEvent son el tipo para representar el

ModelEvent que se haya notificado, una variable entera para especificar el índice de la población que se quiere seleccionar y finalmente, un arreglo de booleanos que permite saber qué criterios se deben de tener en cuenta para el cálculo del peso de las rutas.

Model

El modelo almacena en un ArrayList llamado "poblations" las poblaciones y en otro ArrayList llamado "routes" las rutas, estas dos estructuras contienen las referencias a objectos Poblation y Route respectivamente. Cabe destacar, que las referencias a estos objetos solo se almacenarán en estos dos arreglos, es decir serán únicas en todo el programa y cuando se quiera referenciar el objeto en cuestión, ya sea una población o una ruta, se usará el índice numérico de la posición que ocupa en su arreglo respectivamente. Por otro lado, para indicar el tipo de grafo, puesto que este podría ser dirigido o no dirigido, se usa un String llamado "type". Dentro de esta clase también se define un arreglo llamado "pobSelected" que contiene los pueblos seleccionados para modelar el camino, es decir en la primera posición se guarda el índice del origen, en la segunda el de la población intermedia y en la tercera el del destino. Por último, existe una variable para almacenar la referencia al **main** y un arreglo de tipo *double* llamado "weights" el cual define por defecto los valores de los criterios siendo 0.6 para la distancia, 0.3 para el tiempo y 0.1 para el coste monetario que se usarán para calcular los pesos de las rutas.

Los atributos "poblations" y "routes" utilizan una estructura ArrayList puesto que mientras se genera el grafo del mapa, no se saben cuántos elementos deben ser almacenados en cada lista, por lo que la adaptabilidad frente a variaciones en la dimensión que ofrece el ArrayList nos resulta de gran utilidad para llevar a cabo esta labor. El ArrayList proporciona la misma ventaja que el array al uso de java añadiendo esta capacidad de incrementar o decrementar su dimensión.

Por parte del atributo "pobSelected", en un primer momento pensamos en definirlo con una estructura HashMap que utilizase como clave un tipo enumerado para representar los tres tipos de poblaciones que constituyen el camino, pero en última instancia se ha decidido utilizar un arreglo para almacenar esta información ya que la posición en el arreglo basta para identificar el tipo de población que queremos manejar y esto conlleva a un ahorro de memoria respecto a la otra implementación.

El constructor de la clase recibe el **main,** que referencia al programa principal para realizar las comunicaciones entre componentes; y se inicializan las estructuras de la clase se seleccionando que el tipo de grafo será dirigido. A continuación, se llama a la función para inicializar el camino initializePobSelected(), la cual es la encargada de inicializar "pobSelected" con el origen, el pueblo intermedio por el que pasará y el destino a -1 (sin valor), indicando así que aún no se han seleccionado las poblaciones.

Volviendo a las estructuras para almacenar los elementos del grafo se han declarado dos métodos para la inicialización: addPoblation(), el cual añade un objeto "Poblation", que contiene el nombre y las coordenadas que se han recibido como parámetros, al ArrayList correspondiente; y addRoute(), que hace lo mismo con un objeto "Route" pero no sin antes almacenar en él el índice de las dos poblaciones que han sido pasadas por parámetro mediante sus respectivos nombres en String, representando el origen y el destino, así como almacenando los valores en distancia, tiempo y dinero que supone tomas esa ruta o carretera.

Por otro lado, se han implementado una serie de métodos *getters* necesarios para que el controlador pueda obtener información sobre el estado del modelo.

Los métodos son:

- getPoblation(): Se coge el índice que ocupa el pueblo pasado por parámetro como String en el ArrayList.
- getNPoblations(): Devuelve el tamaño del ArrayList de poblaciones.
- getNExitRoutes(int index): Devuelve la cantidad de rutas que salen desde el pueblo pasado por parámetro como índice.
- getNEntryRoutes(int index): Devuelve la cantidad de rutas que entran al pueblo pasado por parámetro como índice.
- getDestPoblation(int indexp, int indexr): Devuelve el pueblo destino de la ruta. Para saber de qué ruta hablamos se le pasa por parámetro indexp como el índice del pueblo desde donde sale y indexr la posición que ocupa el índice de la ruta dentro de la lista de rutas salientes de esa población en concreto.
- getExitRouteValue(int indexp, int indexr): Devuelve el peso de la ruta mediante el método getter getValue() de la clase ruta. Para saber de qué ruta hablamos se le pasa por parámetro indexp como el índice del pueblo desde donde sale y indexr la posición que ocupa el índice de la ruta dentro de la lista de rutas salientes de esa población en concreto.
- getOriginPoblation(int indexp, int indexr): Devuelve el pueblo origen de la ruta. Para saber de qué ruta hablamos se le pasa por parámetro indexp como el índice del pueblo al que llega y indexr la posición que ocupa el índice de la ruta dentro de la lista de rutas entrantes de esa población en concreto.

- getEntryRouteValue(int indexp, int indexr): Devuelve el peso de la ruta mediante el método getter getValue() de la clase ruta. Para saber de qué ruta hablamos se le pasa por parámetro indexp como el índice del pueblo al que llega y indexr la posición que ocupa el índice de la ruta dentro de la lista de rutas entrantes de esa población en concreto.

- getOrigin(): Devuelve el pueblo de origen.

- getDest(): Devuelve el pueblo de destino.

- getMiddle(): Devuelve el pueblo intermedio.

- getPobName(int index): Devuelve el nombre del pueblo cuyo índice es representado por el valor index pasado por parámetro.

- getPoblationX(int index): Devuelve la coordenada X del pueblo cuyo índice es representado por el valor index pasado por parámetro.

- getPoblationY(int index): Devuelve la coordenada Y del pueblo cuyo índice es representado por el valor index pasado por parámetro.

Otros métodos son: isSelected(int index) y selectionCompleted() dos métodos booleanos, el primero es el encargado de notificar si el pueblo elegido esta seleccionado por el usuario y por lo tanto se representará su nodo de color negro en el mapa mientras que el otro es el encargado de verificar que haya tres pueblos seleccionados antes de poder lanzar el algoritmo.

La clase Model implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realizan unas operaciones u otras en función del ModelEventType recibido. UPDATE_ORIGIN, UPDATE_MIDDLE, UPDATE_DESTINATION actualizan la posición del array del origen, punto medio o destino dependiendo del evento recibido; START itera sobre la lista de rutas para calcular el peso de cada una en función de los criterios que haya seleccionado el usuario; y RESET vuelve a realizar la inicialización.

Poblation

Las poblaciones del mapa son los nodos del grafo, por lo que esta clase sirve como entidad de representación para los nodos del modelo.

De cada población se almacena el nombre y las coordenadas con las que aparece en el mapa, junto con una lista con los índices numéricos de las rutas que salen y otra para los de las rutas que llegan a la población. Como se ha explicado antes estos índices representan las posiciones que ocupan las rutas correspondientes en el ArrayList “routes” del modelo. Esto es necesario ya que en un mapa las carreteras y rutas, a pesar de que muchas veces son bidireccionales, no tienen por qué serlo siempre, lo que implica que un mapa presenta la estructura de un grafo dirigido, en el que cada ruta tiene una población origen y una población destino. Por esta razón se ha comentado anteriormente que el atributo “type” que indica si se trata de un gráfico dirigido o no dirigido siempre almacenará el primer valor. Los atributos mencionados aparecen en el código con la siguiente definición:

- String name
- int coordx, coordy
- ArrayList<Integer> exitRoutes
- ArrayList<Integer> entryRoutes

Se ha decidido utilizar nuevamente la estructura ArrayList por los mismos motivos que han sido explicados para el caso del Model.

Por lo que respecta a los métodos de la clase, se trata de una serie de métodos ‘getters’ y ‘setters’ *protected* para acceder y actualizar los elementos de la clase respetando el nivel de abstracción. Estos métodos son:

- getName(): Devuelve la cadena de caracteres que representa el nombre de la población.

- addExitRoute(Integer r): Añade el índice que representa a una ruta a la lista ExitRoutes.

- addEntryRoute(Integer r): Añade el índice que representa a una ruta a la lista EntryRoutes.

- getNExitRoutes(): Devuelve el número de rutas almacenadas en la lista ExitRoute.

- getNEntryRoutes(): Devuelve el número de rutas almacenadas en la lista EntryRoute.

- getExitRoute(int i): Devuelve el índice de la ruta almacenada en la posición indicada por el parámetro ‘i’ de la lista ExitRoute.

- getEntryRoute(int i): Devuelve el índice de la ruta almacenada en la posición indicada por el parámetro ‘i’ de la lista EntryRoute.

- getCoordx(): Devuelve la coordenada x en la que se ubica la población en el mapa.

- getCoordy(): Devuelve la coordenada y en la que se ubica la población en el mapa.

Route

Esta clase sirve para representar las aristas de nuestro grafo o, como bien indica su nombre, las rutas entre los diferentes pueblos (nodos).

Esta clase tiene tres atributos privados:

1. Poblations: Se trata de un arreglo de enteros de dos posiciones que almacena los índices de las dos poblaciones que conecta una ruta, siendo el índice de la primera posición el que se corresponde con la población de origen de la ruta y el índice de la segunda posición el que se corresponde con la población de destino.

2. Criterias: Se trata de un arreglo de tipo “double” que representa los criterios de evaluación de la ruta. Hay tres criterios: distancia, tiempo y coste monetario. El arreglo se inicializa en el constructor al pasar los valores para estos criterios por parámetro.

3. Weight: Se trata de un valor de tipo “double” que representa el peso de la ruta el cual de calcula con un método llamado setWeight(). Este peso representa el coste que los criterios dan a la ruta en su conjunto y es por ello por lo que para obtenerlo se hace uso de unas ponderaciones previamente definidas en el modelo.

La clase tiene un constructor público que acepta dos enteros "pueblo1" y "pueblo2", y tres *doubles* "distancia", "tiempo" y "monetario". Este constructor inicializa los atributos de la clase con los valores pasados como argumentos.

La clase también implementa 4 métodos públicos: "getOrigin()", que devuelve el primer elemento del arreglo "poblations", es decir, la población origen; "getDestination()", que devuelve el segundo elemento del arreglo "poblations", o lo que es lo mismo, la población a la que se dirige; "getValue()", que devuelve el valor del atributo "weight"; y setWeight(), al cual se le pasa por parámetro un arreglo de booleanos para saber qué criterios se aplican para el cálculo, ya que estos pueden variar según indique el usuario a través de la vista, y otro array que contiene las ponderaciones de cada criterio que como se ha comentado anteriormente se definen desde el modelo.

VI.II CONTROLLER

El controlador es el componente con el rol de bisagra situada entre la vista y el modelo de datos previamente explicado. Es decir, es el encargado de gestionar el flujo de datos entre estos dos módulos. Este flujo de datos es transformado mediante operaciones ejecutadas por el mismo controlador y visualizado en la vista o almacenado en el modelo.

En nuestra aplicación el controlador se responsabiliza de ejecutar el algoritmo ávido siguiendo la técnica Dijkstra para encontrar la ruta con menor peso según el criterio seleccionado por el usuario desde una población de origen hasta llegar a una población de destino, pasando obligatoriamente por una tercera población; todas ellas habiendo sido seleccionadas por el usuario y almacenadas en el modelo como paso previo a la ejecución del algoritmo. El resultado de la ejecución será una lista de índices como representación de las poblaciones almacenadas en el modelo.

ControllerEvent

La clase ControllerEvent hereda de **Event** y se utiliza para que el **main** notifique al Controller cuando se produzca un evento que este deba gestionar. Además, en ella aparece un enumerado ControllerEventType para diferenciar el tipo de ControllerEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro del Controller.

El ControllerEventType compone los eventos de controlador:

- START
- STOP

Los atributos de ControllerEvent son el tipo para representar el ControllerEvent que se haya notificado y un tipo enumerado para representar la estructura que se va a utilizar en la ejecución del algoritmo.

Controller

El controlador consta de una serie de atributos de clase para gestionar su funcionamiento. En primer lugar, contiene dos variables para almacenar referencias a instancias de otras clases, una que referencia al **main** que se inicializa al pasársela como parámetro en el constructor y una para Model que se inicializa justo antes de la ejecución del algoritmo. Esta segunda referencia se usa debido a que el modelo no se modifica durante el transcurso de la ejecución del algoritmo y resulta de utilidad ya que se accede a sus atributos a lo largo de toda la clase Controller. También son atributos de esta clase un objeto de tipo Thread que almacena la meta-data sobre el hilo de la ejecución del algoritmo y un objeto de tipo StructureTechnique que es de utilidad a la hora identificar qué tipo de estructura se va a querer utilizar durante el proceso de etiquetado de Dijkstra. Este último tipo de dato se explica en detalle en el siguiente subapartado. Por último, se tiene un ArrayList para ir almacenando de forma dinámica los índices de las poblaciones que forman parte del camino solución y un arreglo “minimumDistance” de tipo *double* para guardar el peso de cada población visitada durante el transcurso

algoritmo. Las posiciones de este arreglo están mapeadas sobre la estructura que almacena las poblaciones en Model y es por ello por lo que cada índice en el arreglo representa a su respectiva población.

Cabe destacar que, como primera aproximación, en lugar de tener un *array* usábamos un *HashMap* para mapear cada población con su correspondiente peso. Sin embargo, vimos que la opción de usar esa estructura no tenía sentido ya que al hacerlo con un arreglo tradicional también obteníamos un coste *O(1)* y hacía el diseño más sencillo y liviano.

Por otro lado, otra puntualización es que seguimos usando la misma metodología comentada en el apartado de Model para referenciar a las poblaciones y sus rutas. Es decir, usamos sus respectivos índices numéricos sobre los ArrayLists que los almacenan.

Volviendo a la implementación, el controlador hace uso de una función para que al lanzar el *thread* este pueda ejecutar el código del algoritmo. La función en cuestión se llama "run()" y realiza un *Override* ya que se implementa mediante la interfaz "Thread". Lo primero que hace es obtener la referencia del modelo, como se ha explicado anteriormente, y todo seguido instancia el ArrayList de la variable solución. Como se ha implementado el algoritmo de Djikstra usando tres tipos de etiquetado (explicados posteriormente) la función usa la variable "StructureTechnique" para distinguir y ejecuta la siguiente lógica: Etiqueta el grafo usando el nodo intermedio como nodo origen; calcula el camino mínimo desde el nodo origen, de momento el nodo intermedio, hasta al nodo destino; vuelve a etiquetar el grafo pero partiendo desde el verdadero nodo origen; y calcula de nuevo el camino mínimo desde el origen hasta el nodo intermedio, es decir, este último es interpretado como nodo destino esta vez. En el proceso descrito se obtiene el listado de índices de las poblaciones que comprenden la solución global, pero ordenados de destino a origen, por tanto bastaría con llamar a la función "reverse()" de la librería *Collections* para pasarle a la vista la solución con el orden adecuado.

La función que calcula el resultado a partir del grafo etiquetado es la función "calculate()". La lógica que sigue está basada en un recorrido invertido que parte desde el nodo destino y que siempre va a ir al nodo cuyo peso puede conseguir restándole a su peso el de la ruta que los conecta. Es decir, el algoritmo obtiene el peso del nodo actual *pD* así como las rutas que le llegan desde otros, entonces para cada una de ellas se consulta el peso del nodo origen correspondiente *pO* así como el peso de la arista *pR*, si al realizar la resta *pD – pR* da el valor de *pO* se cambia el índice de nodo actual y se vuelve a realizar la iteración hasta llegar al nodo origen de la solución global. Un punto a destacar es que en cada iteración el método mete en el *ArrayList* solución el índice del nodo en el que se encuentra, de esta manera se obtiene una lista con el camino solución invertida.

Por otro lado, en cuanto al etiquetado se refiere, se ha desarrollado un algoritmo iterativo ávido que sigue la metodología de Djikstra. Es decir, en el algoritmo clásico se usa una cola para almacenar las poblaciones a medida que se van viendo y se van eliminando las visitadas, sin embargo, esto puede suponer un coste elevado de $O(n^2 + a)$, siendo n igual al número de poblaciones y a al número de rutas, este último valor a tiende a n^2 debido a que como las carreteras son en su mayoría bidireccionales hemos especificado los grafos de tal manera que cada dos nodos que estén conectados lo harán con dos rutas, una en cada sentido. Sin embargo, también hemos implementado dos variaciones que ordenan las rutas mediante su peso y permiten realizar un etiquetado siguiendo esa propiedad, estas son el etiquetado usando una cola de prioridad y otro usando el montículo de Fibonacci. A continuación, se describen las tres funciones que implementan Dijkstra:

- tagGraf(): Esta función es la que implementa el algoritmo tradicional con una cola con el coste definido anteriormente.

- binaryTagging(): El etiquetado binario se realiza con una cola de prioridad, usando el objeto *PriorityQueue* de Java, en la cual los elementos se ordenan en forma de *Heap.*

  Esta implementación tiene un coste $O((a + n) * \log(n))$ ya que la operación de *Decrease-Key* para actualizar un peso, que se realiza a veces en el peor de los casos, en una cola de prioridad es $O(\log(n))$ y la de *Insert-Key* lo mismo, pero realizándose n veces. Cabe destacar que este método es bueno cuando el número de arestas no es muy elevado, es decir, en nuestro caso donde el número de rutas tiende a n^2 obtenemos un coste $O\big((n^2 + n) * \log(n)\big)$ que es peor que el que se tiene con el uso de la cola.

- fibonacciTagging(): El etiquetado de Fibonacci sigue la misma idea que el método anterior pero, sin embargo, es más eficiente a la hora de realizar la actualización del peso de un nodo con un O(1) y es por ello que su coste asintótico es de $O(a + n * \log(n))$.

  > "A pesar de esto, sin embargo, los montones de Fibonacci a menudo se consideran lentos en la práctica debido a su mayor consumo de memoria y los altos factores constantes contenidos en sus operadores." [5] .

  Es decir, en la teoría, esta metodología supera a la anterior en eficiencia si nos basamos en la representación del coste asintótico, sin embargo, en la práctica esto no suele ocurrir hasta cierto valor muy

elevado de n al que no es común llegar y por ende usar una de las técnicas anteriores es mejor en el caso general.

Cabe destacar que, al Java no tener ninguna librería que implemente esta estructura de datos, hemos optado por importar una librería externa llamada *FibonacciHeap* [6].

En las siguientes tres figuras se pueden observar los tiempos medios obtenidos por cada tipo de estructura con Dijkstra. Se ha realizado la misma prueba, es decir mismos nodos origen, de paso y destino; para los diferentes mapas disponibles. Modificando el número de iteraciones totales, se ha intentado obtener una media del valor tiempo para comprobar el comportamiento de la ejecución del algoritmo haciendo uso de las distintas estructuras.

Tras observar los gráficos, se puede corroborar que el montículo de Fibonacci, pese a tener un buen coste asintótico teórico, llevado a la práctica no se obtienen buenos resultados a menos que no se tenga un número mucho mayor de poblaciones que las que se tienen en los mencionados mapas. Por este motivo, este tipo de estructura resulta ser para la que normalmente se obtiene de media un coste temporal superior a las del resto. Para lo que respectan, el montículo binario y la cola normal, vemos como para estas dos se obtienen unos valores similares en las distintas ejecuciones, siendo a veces uno mejor que otro y viceversa. Esto puede ser respondido por lo que ya se ha mencionado anteriormente, dónde se exponía cómo debido al tipo de mapas que se han desarrollado, el montículo binario no podía llegar a valores mejores por causa del número de arestas presentes en el mapa.

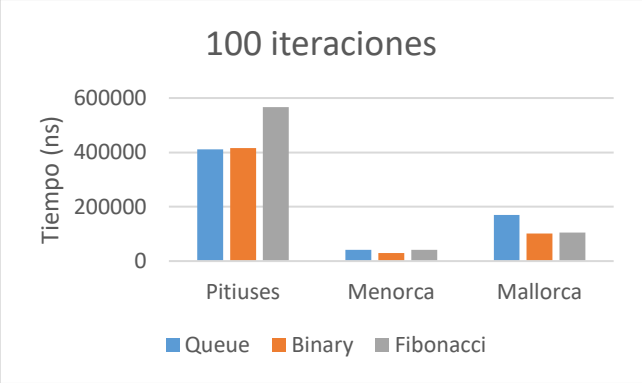

Figura 5: Gráfico comparativo haciendo 100 iteraciones

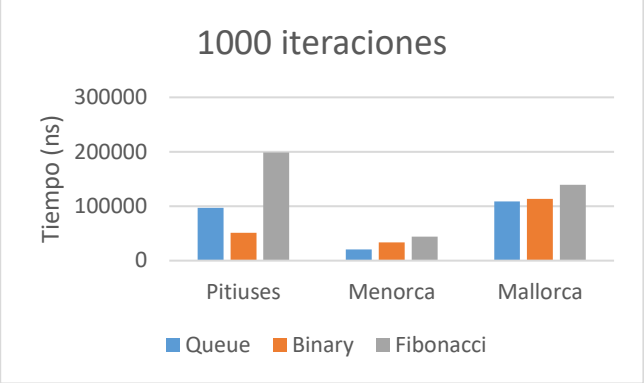

Figura 6: Gráfico comparativo haciendo 1000 iteraciones

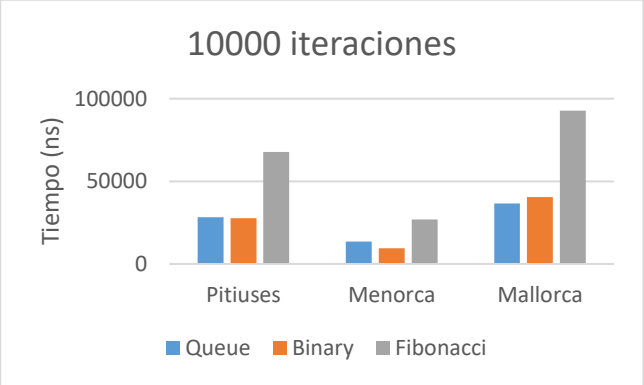

Figura 7: Gráfico comparativo haciendo 10000 iteraciones

Tanto en el etiquetado binario como en el de Fibonacci, a diferencia del normal en el cual los elementos de la cola son enteros que representan los índices de las poblaciones, se usa un objeto aparte llamado *QueueElement* que se describe posteriormente en su respectivo subapartado.

Falta por definir cómo funciona la lógica de Djikstra en nuestro programa. Entonces, el algoritmo empieza inicializando el arreglo "minimumDistance" que contiene los etiquetados y la estructura de datos que permite realizar el recorrido, así como se inserta en la estructura la población origen y se etiqueta con un peso de 0. A partir de aquí se itera sobre la estructura y mientras no esté vacía se extrae un nodo de ella obteniendo las rutas que salen de él. Por cada una de estas rutas se obtiene la población a la que llega y se le calcula un peso acumulado *pAcc* mediante el peso del nodo actual y el de la ruta en cuestión, entonces se consulta si el valor que tenía esa población en "minimumDistance" era mayor al nuevo *pAcc* i se actualiza su etiquetado si se da el caso. Este proceso iterativo parará cuando la estructura se encuentre vacía queriendo decir que ya no hay más poblaciones a evaluar y por lo tanto el etiquetado es el definitivo.

Para realizar la suma de los valores *doubles* se utiliza la

función auxiliar “roundDouble()” consiguiendo así facilitar la comparativa entre los valores *double* utilizados en el algoritmo.

Por último, la clase Controller implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realizan unas operaciones u otras en función del ControllerEventType recibido. START inicializa la variable “selectedTechnique” para que el programa sepa cual ejecutar, así como lanza al *Thread* y STOP interrumpe el *Thread* parando la ejecución.

StructureTechnique

Esta clase se trata de un enumerado que contiene las diferentes técnicas ya descritas previamente:

- QUEUE
- BINARY_HEAP
- FIBONACCI_HEAP

QueueElement

Esta clase Java se utiliza para crear un objeto que representa un elemento de una cola de prioridad o de un montículo de Fibonacci. Estas estructuras se utilizan para almacenar objetos en un orden específico siguiendo unas prioridades previamente establecidas.

La clase QueueElement tiene dos variables de instancia: "node" y "weight". "node" es un entero que representa un nodo en una estructura de datos y "weight" es un número de punto flotante que representa el peso o la prioridad de ese nodo.

El constructor de la clase QueueElement toma dos argumentos los cuales se utilizan para inicializar las variables de instancia.

La clase QueueElement implementa la interfaz Comparable, que se utiliza para comparar objetos y determinar su orden. El método "compareTo" se utiliza para comparar la prioridad de dos objetos QueueElement. El método devuelve un entero que indica si el objeto actual es mayor, menor o igual al objeto que se pasa como parámetro.

En el método "compareTo", se compara el peso del objeto actual con el peso del objeto que se pasa como parámetro. Si el peso del objeto actual es mayor que el peso del objeto que se pasa como parámetro, el método devuelve 1. Si en cambio es menor, devuelve -1. Si son iguales devuelve 0.

VI.III VIEW

La vista representa la interfaz de usuario de la aplicación y se encarga de presentar los datos al usuario final. La vista no interactúa directamente con el modelo ni con el controlador, sino que se comunica con ellos a través del gestor de eventos.

MapDisplay

El constructor de esta clase recibe como parámetro una referencia al Model para facilitar la consulta del modelo y poderlo pintar de manera adecuada, también un string con el nombre de la imagen y un entero ‘pobDimension’ para definir la dimensión de los círculos que dan representación a las poblaciones. También se inicializa una ArrayList de valores enteros que almacenará los índices de las poblaciones que forman parte de la futura solución. Así cuando se tenga una solución, una vez pintados todos los pueblos, se iterará sobre la ArrayList para ir marcando sobre el mapa el recorrido del camino mínimo.

El método paint() de la clase utiliza la técnica de doble *buffering* para visualizar el mapa, los nodos y las rutas. Los nodos se representan como circunferencias negras sin rellenar y aquellos que se seleccionen como origen, paso intermedio y destino se rellenan de negro. Finalmente, una vez ejecutado el algoritmo muestra el camino más corto con líneas rojas e indicando el orden en qué se visita cada población. En los casos en los que una población se visite hasta dos veces podía ocurrir que se solaparan dichos números. La solución ha sido colocar los números del primer tramo de la ruta a la izquierda del nodo y los números del segundo tramo a la derecha.

También contamos con métodos auxiliares como repaint() para volver a pintar la pantalla y reset() para eliminar la solución encontrada. A su vez, el método pobSolution(ArrayList<Integer> indexs) el cual actualiza la lista de soluciones con la lista pasada por parámetro y llama al método repaint() para mostrarlo en el mapa, mientras que el método updateImage(String img) se usa para cambiar la imagen del mapa que se visualiza.

ViewEvent

La clase ViewEvent hereda de **Event** y se utiliza para que el **main** notifique a la View cuando se produzca un evento que esta deba gestionar. Además, aparece un enumerado ViewEventType para diferenciar el tipo de ViewEvent que se reciba. De esta manera, se simplifica la identificación de tipo de evento dentro de la View.

El ViewEventType compone los eventos de modelo:

- SHOW_RESULT

ViewEvent únicamente necesita como atributos uno para almacenar el tipo de evento de vista y otro para contener los índices de los nodos que forman parte del camino solución del algoritmo.

View

La componente View envía tanto al Controller como al Model. Notifica al Controller únicamente cuando se pulse el botón para iniciar la ejecución de este y cuando se pulse el botón para detener su ejecución. Al Model se le notifica cuando se seleccione alguna población, cuando se modifique el mapa a visualizar, cuando se cambie el tipo de estructura a utilizar con Dijkstra o cuando se quiera especificar el tipo de criterios por los que se realizará el cálculo de Dijkstra en el momento de la ejecución.

La vista es responsable de presentar los datos de manera clara y coherente para el usuario. Esto incluye la disposición de la información, la selección de la paleta de colores, la presentación de los datos en diferentes formatos y la gestión de la interacción del usuario, además de poder apreciar el resultado del algoritmo.

En el constructor de la clase, solo se recibe la instancia del **main**. Además, se inicializan los atributos privados de la clase: se obtiene la instancia del Model, ya que se consulta constantemente en la View; la dimensión de los nodos del mapa se inicializa con valor entero '8' y el arreglo de *boolean* para representar con un valor True qué criterios se han seleccionado para ser utilizados con la ejecución.

Podemos apreciar todos estos elementos en la interfaz:

1. mapComboBox: Recibe como lista de elementos los diferentes mapas disponibles que se encuentra presentes en la clase enumerada Map. Cuando se seleccione uno, se identificará cuál ha sido seleccionado utilizando el índice correspondiente de la ComboBox.

2. buttonGroup: Incluye tres componentes RadioButton en representación a los nodos origen, de paso y destino; de esta manera al seleccionar un RadioButton y pulsar sobre un nodo del mapa, se puede identificar dicho nodo en función del RadioButton. Justo debajo del grupo de botones, aparecen unos JLabels para mostrar los nombres de los nodos que han sido seleccionados.

3. structureComboBox: Recibe como lista de elementos los diferentes tipos de estructuras que se pueden utilizar para llevar a cabo la ejecución del algoritmo. El listado viene dado a partir de la clase enumerada StructureTechnic. Cuando se seleccione uno, se identificará cuál ha sido seleccionado utilizando el índice correspondiente de la ComboBox.

4. distanceBox: JCheckBox para indicar si se quiere utilizar la distancia de las rutas para calcular el camino mínimo. Si no se selecciona ninguna de las JCheckBox, este criterio se utiliza como predeterminado.

5. timeBox: JCheckBox para indicar si se quiere utilizar el tiempo de las rutas para calcular el camino mínimo.

6. moneyBox: JCheckBox para indicar si se quiere utilizar el coste monetario de circular a través de las rutas para calcular el camino mínimo.

7. map: Visualiza el mapa que se procesa de un fichero. Es una instancia de la clase MapDisplay, la cuál extiende de la clase JPanel de Swing.

8. progressBar: Debajo del mapa aparece la barra de progreso, una JProgressBar, que se mantiene inactiva hasta que el algoritmo se pone en ejecución. Una vez se detiene el algoritmo, también se detiene la barra.

9. solutionList: Inicialmente se encuentra vacía. Cuando se recibe el evento correspondiente en la View, se crea un arreglo para contener los nombres de las poblaciones que forman parte de la solución y se le pasa a la instancia de la JList para que se puedan visualizar.

10. JButton (Start): Se permite pulsar el botón mientras no haya ninguna ejecución del algoritmo iniciada siempre y cuando se haya seleccionado un nodo origen, de paso y destino. Para conocer si se cumple esta segunda condición, se consulta a la instancia de Model. Cuando el botón sea pulsado, envía al Model el arreglo de los criterios para que se tengan en consideración para el cálculo de los pesos de las arestas del modelo. Al Controller envía el índice en la 'structureComboBox' para que se pueda identificar el tipo de estructura que se debe utilizar.

11. JButton (Reset): Limpia todos los valores seleccionados en la vista y notifica al Model que debe hacer el *reset* así como también al Controller. No solo eso, otros componentes que serán actualizados son la 'solutionList' para eliminar la posible lista de nombres y al 'mapDisplay' para que inicialice la lista de puntos de la solución y haga el correspondiente *refresh*.

Los elementos de la vista previamente descritos se pueden

observar en la Figura 5.

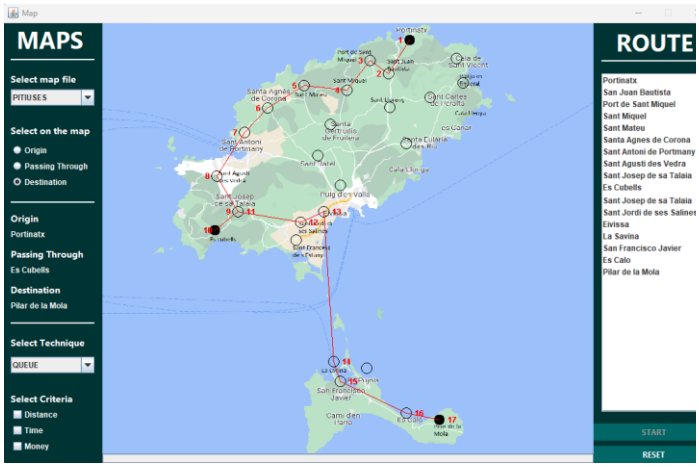

Figura 8: Vista de la aplicación

La clase View implementa la interfaz **EventListener,** por lo que se debe implementar el método **notify()** donde se realiza la operación para mostrar los resultados de la ejecución del algoritmo. Se muestra la lista de poblaciones que conforman el camino solución y se le pasa al panel del mapa para que se pueda visualizar el camino. También se detiene la 'progressBar'.

VII. CONCLUSIÓN

El equipo considera que la implementación del algoritmo ávido de *Dijkstra* de la práctica utilizando el patrón de diseño MVC ha sido exitosa en términos de estructura y eficiencia. El uso del MVC junto a su consecuente abstracción ha sido bastante buena ya que nos ha permitido separar la lógica del algoritmo de la interfaz de usuario sin interferir unas con otras, lo que ha resultado en un código más organizado y fácil de mantener. Todo lo anterior junto a la implementación de las tres estructuras para el uso del algoritmo de Dijkstra: la cola simple, el montículo binario y montículo de Fibonacci; ha permitido realizar un estudio de costes muchos más amplio. Además, pensamos que se ha implementar un programa robusto y escalable en el que se podrían introducir más mapas.

Conseguimos acabar nuestro cuarto trabajo juntos, en el que hemos podido demostrar ya nuestra comprensión de la implementación del MVC, y donde el equipo ha funcionado bastante bien. El control de versiones mediante GitHub ha sido determinante, mejorando nuestros conocimientos de la plataforma y del comando *git* sobre todo. El Trello ha sido también una buena herramienta para poder ir notificando que cosas quedaban por hacer al equipo y que cosas se estaban haciendo o ya habían sido terminadas. Por los problemas que respectan a esta práctica, no ha habido muchos, simplemente nos quedamos con el cambio que hicimos al final de muchas estructuras que habíamos implementado inicialmente como Hash, finalmente las pasamos a array por tema de eficiencia en memoria.

En resumen, creemos que el resultado final conseguido por el equipo ha sido bastante satisfactorio sobre todo a la hora de realizar el análisis de los algoritmos ávidos y de las estructuras que podemos usar para implementarlos.

VIII. ENLACES

1. Teoría del patrón MVC: https://developer.mozilla.org/en-US/docs/Glossary/MVC

2. Teoría del patrón por eventos: https://www.tibco.com/es/reference-center/what-is-event-driven-architecture

3. Ventajas y desventajas del patrón por eventos: https://es.wikipedia.org/wiki/Arquitectura_dirigida_por_eventos

4. Algoritmos Ávidos: https://www.programiz.com/dsa/greedy-algorithm

5. R. Lewis. *A Comparison of Dijkstra's Algorithm Using Fibonacci Heaps, Binary Heaps, and Self-Balancing Binary Trees*. School of Mathematics, Cardiff University, Cardiff, Wales. March 2023 https://arxiv.org/pdf/2303.10034.pdf

6. Librería que implementa la estructura de datos del Montículo de Fibonacci: https://github.com/trudeau/fibonacci-heap/tree/master/src/main/java/org/nnsoft/trudeau/collections/fibonacciheap