



POLITÉCNICA

"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL



Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

TRABAJO FIN DE GRADO

Optimización de caminos de coste mínimo en grafos
de decisión a través de la Aplicación de algoritmos de
búsqueda

Autor: Francisco Jiménez González

Director: Vicente Martínez Orga

MADRID, JUNIO 2019

Índice

1.- INTRODUCCIÓN	1
1.1.- OBJETIVO	1
2.- TRABAJOS PREVIOS	2
3.- EL ALGORITMO A*	5
3.1.- DEFINICIÓN	5
3.2.- DESARROLLO	5
3.3.- CÓDIGO	6
3.3.1.- CLASE STATION	6
3.3.2.- CLASE DATACONTROLLER	9
3.3.3.- CLASE ASTARCONTROLLER	15
4.- DISEÑO	23
4.1.- DISEÑO DE ALTO NIVEL	23
4.2.- DISEÑO DE BAJO NIVEL	24
4.3.- ESPECIFICACIÓN DE REQUISITOS	25
5.- PRUEBAS UNITARIAS	26
6.- BIBLIOGRAFÍA	30

1.- INTRODUCCIÓN

1.1.- Objetivo

El propósito general del presente Trabajo de Fin de Grado será el de desarrollar un sistema de toma de decisiones en un grafo, concretamente un sistema que encuentre el camino más corto entre dos estaciones de una red de metro de una ciudad. Esta aplicación encontrará el trayecto de menor tiempo entre dos puntos sirviéndose del algoritmo de búsqueda A* estudiado en la asignatura de Inteligencia Artificial. Así mismo, durante el Trabajo de Fin de Grado, se aplicaron las técnicas de desarrollo, control y gestión de proyectos estudiadas en las asignaturas de Interacción Persona-Ordenador, e Ingeniería de Software para mejora y optimización del tiempo de desarrollo.

La red de Metro elegida para el proyecto es la del Metro de Madrid, con motivo del centenario de su inauguración, y la representación gráfica de la plataforma será ejecutada a través de una aplicación de escritorio en Java, a través de la cual el usuario selecciona dos estaciones de la red, una de comienzo del trayecto y una de final, y mediante el algoritmo de búsqueda descrito anteriormente el programa calculará y mostrará el trayecto más corto, contando con los posibles transbordos asociados a las estaciones. Además, no solo calculará el trayecto que menor tiempo emplearía, sino que además estimaremos el tiempo que tomaría realizar dicho trayecto, ya que utilizaremos herramientas de geoposicionamiento y, ayudados de datos de velocidades medias de los trenes, intentaremos ofrecer un tiempo aproximado de trayecto para la distancia que separa ambas estaciones.

2.- TRABAJOS PREVIOS

Para el desarrollo del trabajo planteado, uno de los puntos críticos es una correcta definición de la base de datos de las estaciones que conformarán el plano de Metro. Para el funcionamiento del algoritmo, es necesario identificar cada una de las estaciones en su localización real para poder calcular la distancia tanto entre estaciones adyacentes como la distancia hasta la estación objetivo.

En primer lugar se optó por buscar un dataset que facilitara la mayor información posible, como de localización como de transbordos, por lo que se partió del dataset que proporciona la Comunidad de Madrid en su portal. Sin embargo, a pesar de la cantidad de información de la que disponía (accesos para minusválidos, aparcamientos, etc), este dataset no reunía la información de los transbordos ni sus coordenadas, al menos de una manera clara y concisa, por lo que se optó por la generación de un dataset propio que contuviese la información que el algoritmo A* necesita para su correcto funcionamiento.

Así pues, el dataset, para el que se escogió el CSV como tipo de archivo por su comodidad tanto para leer como para escribir datos en él, debía estar estructurado de la siguiente manera: cada una de las líneas del documento representará un nodo del algoritmo, teniendo las estaciones con transbordo tantos nodos como líneas confluyan en dicha estación. Así pues, para cada nodo, se define la línea a la que pertenece el nodo, su nombre, y sus coordenadas X e Y. De esta manera, el algoritmo iterará entre los nodos más cercanos de una misma línea, y podrá realizar transbordos de línea entre nodos cuya localización sea la misma.

Para la obtención de las coordenadas X e Y, se ha utilizado la herramienta de localización de Google Maps para cada una de las estaciones de la red de metro, mientras que para la información de los transbordos se ha optado por transcribirla a mano desde la página web del Metro de Madrid (www.metrodemadrid.com).

Por último, uno de los problemas para la implementación del algoritmo fue la necesidad de calcular el tiempo empleado en los transbordos para una estimación más ajustada a la realidad del camino mínimo. Desafortunadamente, no existe información acerca de las distancias de los transbordos puesto que dependen de la arquitectura de cada estación. Por tanto, se optó por una estimación que diera una solución a este problema: se ha definido una jerarquía entre las distintas líneas, simulando una distancia entre ellas estándar en función de la línea a la que pertenecen. De esta manera, se ha definido que cada línea tiene una profundidad bajo tierra uniforme, siendo la línea 1 la que menor profundidad tiene y, la 12, la que más. Así, podemos calcular fácilmente la distancia entre una línea y otra dentro de la misma estación.



El fragmento inferior corresponde a un tramo de la línea 1 de metro. En él, podemos apreciar la estructuración de los datos descrita anteriormente. La primera columna corresponde a la línea a la que pertenece cada uno de los nodos. La segunda, el nombre de la estación. Las columnas tres y cuatro hacen referencia a las coordenadas X e Y de la estación con las que calcularemos la distancia entre nodos. Por último, la quinta columna solo tendrá contenido en los casos en los que su nodo correspondiente tenga un enlace con otra línea. Dado el caso anterior, la manera de definir las líneas con las que hay enlaces es mediante una sucesión de números, indicando la distancia que hay a cada una de las líneas (de 1 a 12), indicando 0 que no hay enlace entre las líneas. Por ejemplo, en el caso de la primera estación (Pinar de Chamartín), se indica que tiene enlace con la línea 4, y que hay 3 niveles de profundidad en su enlace.

1	Pinar de Chamartín	40.480105	-3.666938	0-0-0-3-0-0-0-0-0-0-0-0
1	Bambú	40.476784	-3.676374	
1	Chamartín	40.471842	-3.682621	0-0-0-0-0-0-0-0-0-9-0-0
1	Plaza de Castilla	40.466910	-3.689213	0-0-0-0-0-0-0-0-8-9-0-0
1	Valdeacederas	40.464459	-3.695077	
1	Tetuán	40.460765	-3.698299	
1	Estrecho	40.454555	-3.702847	
1	Alvarado	40.450218	-3.703507	
1	Cuatro Caminos	40.446961	-3.703892	0-1-0-0-0-5-0-0-0-0-0-0
1	Ríos Rosas	40.442078	-3.701376	
1	Iglesia	40.434929	-3.698962	
1	Bilbao	40.429045	-3.702099	0-0-0-3-0-0-0-0-0-0-0-0
1	Tribunal	40.426239	-3.701227	0-0-0-0-0-0-0-0-0-9-0-0
1	Gran Vía	40.419993	-3.702138	0-0-0-0-4-0-0-0-0-0-0-0
1	Sol	40.417116	-3.703230	0-1-2-0-0-0-0-0-0-0-0-0
1	Tirso de Molina	40.412486	-3.704303	
1	Antón Martín	40.412507	-3.699419	
1	Estación del Arte	40.408672	-3.691647	
1	Atocha Renfe	40.407460	-3.688995	
1	Menéndez Pelayo	40.404449	-3.680988	
1	Pacífico	40.401249	-3.674910	0-0-0-0-0-5-0-0-0-0-0-0
1	Puente de Vallecas	40.398298	-3.669311	
1	Nueva Numancia	40.395787	-3.664461	
1	Portazgo	40.392867	-3.658970	
1	Buenos Aires	40.391549	-3.653924	

3.- EL ALGORITMO A*

3.1.- Definición

Un algoritmo de búsqueda es aquel encargado de identificar un elemento con unas características concretas dentro de un grupo de datos. El algoritmo A* es un tipo de algoritmo de búsqueda empleado para el cálculo de caminos mínimos en una red. Es un algoritmo heurístico; es decir, se vale de una función de evaluación heurística para realizar los cálculos, mediante los cuales etiquetará los diferentes nodos de la red y determinará la probabilidad de dichos nodos de pertenecer al camino óptimo.

Para el algoritmo A* disponemos de dos funciones para dicha evaluación, una de ellas para calcular la distancia al siguiente nodo, y una segunda para calcular la distancia hasta el nodo destino. Por tanto, si se desea encontrar el camino más corto desde el nodo de origen S hasta el nodo destino T, un nodo intermedio N de la red queda definido por la siguiente función:

$$f(N) = g(N) + h(N)$$

Siendo $g(N)$ la distancia del camino desde el nodo de origen S hasta el nodo intermedio N, y $h(N)$ la distancia desde el nodo intermedio N hasta el nodo de destino T.

Así pues, cuando menor sea el valor de la función f de un nodo concreto, mayor probabilidad tendrá de encontrarse en el camino mínimo que se busca. El algoritmo debe mantener una lista ordenada por valor creciente de mérito de los nodos que pueden ser explorados, y de ahí seleccionará el de menor valor, que será el primero de la lista. El algoritmo empezará analizando el nodo que se toma como origen para el problema del camino más corto. Calculará su mérito, y a continuación pasará a explorar sus nodos sucesores, es decir, los nodos con los que esté unido por un enlace este nodo de origen. De esos nodos con los que tenga nexo se calculará su mérito, y se continuará con la ejecución del algoritmo evaluando los adyacentes al de mayor mérito. Cuando el algoritmo evalúa el nodo destino del problema será el momento en que el algoritmo termine y ya se dispondría de la solución óptima.

3.2.- Desarrollo

Una vez consolidada la lógica y la estructura de datos que define cada estación de metro, se procede a plasmar en código el funcionamiento del algoritmo A*. La lógica del algoritmo permite dos maneras distintas de afrontar los cálculos: ponderando los nodos por distancia o por tiempo empleado en recorrerla, siendo esta última la empleada para el proyecto, sabiendo que el Metro de Madrid tiene una velocidad media de 31km/h. Para estimar el tiempo empleado en recorrer la distancia entre dos estaciones, damos

por hecho que el recorrido se hace en línea recta, puesto que las vías por las que circulan los trenes no suelen tener curvas significativas.

Para el cálculo del camino mínimo entre ambas estaciones se ha utilizado una lista cerrada y una lista abierta. La lista abierta la hemos implementado con una PriorityQueue (biblioteca propia de la asignatura de Algoritmos y Estructura de Datos). Esta cola con prioridad nos permite, con un simple método como es removeMin(), obtener y eliminar el elemento de menor valor de esta. En cuanto a la lista cerrada con un simple array de Stations nos sirve, puesto que lo único que nos interesa es saber las paradas que contiene.

Más específicamente para el cálculo de la función heurística, lo primero a realizar es calcular la distancia del nodo inicial al nodo destino. Dicha distancia se obtiene desde el anterior nodo consolidado hasta el nodo actual en línea recta. Posteriormente, obtenemos la distancia en línea recta entre el nodo actual y el destino, por lo que el valor de f es la suma de ambas distancias, como quedó definido anteriormente.

3.3.- Código

3.3.1.- Clase Station

```
package models;

import java.util.Map;

public class Station {

    private static double SPEED = 0.5;

    private String lineStation;
    private double xStation;
    private double yStation;
    private String nameStation;
    private Map<String, Integer> transferStation;
    private Station prevStation;
    private Station nextStation;

    private Station parentStation;

    public Station(String linea, String nombre) {
        lineStation = linea;
        nameStation = nombre;
    }
}
```

```
        transferStation = null;
    }
```

```
    public String getLineStation() {
        return lineStation;
    }
```

```
    public void setLineStation(String lineStation) {
        this.lineStation = lineStation;
    }
```

```
    public double getXStation() {
        return xStation;
    }
```

```
    public void setXStation(double xStation) {
        this.xStation = xStation;
    }
```

```
    public double getYStation() {
        return yStation;
    }
```

```
    public void setYStation(double yStation) {
        this.yStation = yStation;
    }
```

```
    public String getNameStation() {
        return nameStation;
    }
```

```
    public void setNameStation(String nameStation) {
        this.nameStation = nameStation;
    }
```

```
    public Map<String, Integer> getTransferStation() {
        return transferStation;
    }
```

```
    public void setTransferStation(Map<String, Integer>
transferStation) {
        this.transferStation = transferStation;
    }
```

```
    public Station getPrevStation() {
        return prevStation;
    }
```

```
    public void setPrevStation(Station prevStation) {
        this.prevStation = prevStation;
    }
```

```
    public Station getNextStation() {
        return nextStation;
    }
```

```
    public void setNextStation(Station nextStation) {
        this.nextStation = nextStation;
    }
```

```
    public Station getParentStation() {
        return parentStation;
    }
```

```
    public void setParentStation(Station parentStation) {
        this.parentStation = parentStation;
    }
```

```
    public double time(Station dstStation) {
        double time = 0;
        if (this.transferStation != null && !
this.equals(dstStation) &&
this.nameStation.equals(dstStation.getNameStation())){
            time = 90 *
this.transferStation.get(dstStation.getLineStation());
        } else if(!
this.nameStation.equals(dstStation.getNameStation())) {
            time = 6000 *
(Math.sqrt(Math.pow((dstStation.getxStation() -
```

```

        this.getxStation()), 2) +
        Math.pow((dstStation.getyStation() - this.getyStation()),
        2))) / SPEED;
    }
    return time;
}

    public boolean equals(Station station) {
        return
this.lineStation.equals(station.getLineStation()) &&
this.nameStation.equals(station.getNameStation());
    }

    @Override
    public String toString() {
        return "<" + this.lineStation + ":" +
this.nameStation + ">";
    }

}

```

Cada nodo estación se define como un objeto de la clase **Station**. En dicha clase se definen los parámetros que representan las estaciones, como sus transbordos, su nombre, o la línea a la que pertenecen.. Es especialmente remarcaba la definición de una constante llamada *speed*, que es un factor de corrección que se utilizará, en el método *time()*, para estimar la velocidad a la que se desplaza el tren. Dicho método realiza la estimación de forma distinta para el tiempo que transcurre entre t los transbordos y para los trayectos en tren entre estaciones. Además, en esa clase, se definen tanto los *getters* como los *setters* necesarios para el correcto funcionamiento de la aplicación.

3.3.2.- Clase Datacontroller

```

package controllers;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;

```

```
import java.util.Map;

import launcher.Launcher;
import models.Station;

public class DataController {

    private static String FILENAME = "/resources/
estaciones.csv";

    private Map<String, Station> mapLines;
    private Map<String, ArrayList<String>> mapNameStation;
    private ArrayList<String> listNameStation;

    public DataController() {
        try (BufferedReader br = new BufferedReader(new
InputStreamReader(Launcher.class.getResourceAsStream(FILENA
ME)))) {

            this.mapLines = new HashMap<String,
Station>();
            this.mapNameStation = new HashMap<String,
ArrayList<String>>();
            this.listNameStation = new
ArrayList<String>();

            String metro_line = "-";
            Station current_station = null;
            Station prev_station = null;

            String line = br.readLine();
            while (line != null) {
                current_station =
this.createStation(line);
                this.updateDataLists(current_station);

                if ( !
metro_line.equals(current_station.getLineStation())){
                    prev_station = null;
```

```

current_station.setPrevStation(prev_station);

current_station.setNextStation(null);
    prev_station = current_station;

    metro_line =
current_station.getLineStation();
    this.mapLines.put(metro_line,
current_station);

    line = br.readLine();
    current_station =
this.createStation(line);

this.updateDataLists(current_station);
    }

prev_station.setNextStation(current_station);

current_station.setPrevStation(prev_station);

    prev_station = current_station;

    line = br.readLine();
    }
    Collections.sort(this.listNameStation);

    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public Map<String, Station> getMapLines() {
    return mapLines;
}

```

```
    public Map<String, ArrayList<String>>
getMapNameStation() {
        return mapNameStation;
    }

    public ArrayList<String> getListNameStation() {
        return listNameStation;
    }

    private Station createStation(String line) {
        String[] line_parts = line.split(";");

        Station current_station = new
Station(line_parts[0], line_parts[1]);

        current_station.setxStation(Double.parseDouble(line_parts[2
]));
        current_station.setyStation(Double.parseDouble(line_parts[3
]));

        if (line_parts.length > 4) {
            String[] transfers =
line_parts[4].split("-");
            Map<String, Integer> transfer_station = new
HashMap<String, Integer>();
            for (int i = 0; i < transfers.length; i++) {
                if (!transfers[i].equals("0")) {

transfer_station.put(Character.toString((char) (65 + i)),
Integer.parseInt(transfers[i]));
                }
            }

            current_station.setTransferStation(transfer_station);
        }

        return current_station;
    }
}
```

```
        private void updateDataLists(Station current_station){
            if (!
this.listNameStation.contains(current_station.getNameStation())){
                ArrayList<String> station_lines = new
ArrayList<String>();
station_lines.add(current_station.getLineStation());
this.mapNameStation.put(current_station.getNameStation(),
station_lines);
this.listNameStation.add(current_station.getNameStation());
            } else {
                ArrayList<String> station_lines =
this.mapNameStation.get(current_station.getNameStation());
station_lines.add(current_station.getLineStation());
this.mapNameStation.put(current_station.getNameStation(),
station_lines);
            }
        }

        public ArrayList<Station> getStationsByName(String
station_name){
            ArrayList<Station> station_list = new
ArrayList<Station>();

            ArrayList<String> station_lines =
this.mapNameStation.get(station_name);
            for (String line_station : station_lines) {
                Station current_station =
this.mapLines.get(line_station);
                while(!
current_station.getNameStation().equals(station_name)){
                    current_station =
current_station.getNextStation();
                }
                station_list.add(current_station);
            }
        }
    }
}
```

```

        }
        return station_list;
    }

    public Station getStation(String line, String
station_name){
        Station current_station =
this.mapLines.get(line);
        while(!
current_station.getNameStation().equals(station_name)){
            current_station =
current_station.getNextStation();
        }
        return current_station;
    }
}

```

En la clase **DataController** se hace uso del CSV de estaciones descrito anteriormente. En esta clase se instancia cada uno de los nodos necesarios para encontrar el camino mínimo en un objeto **Station** mediante el método *createStation()*, para posteriormente ser utilizadas en la lógica del algoritmo A* (3.2.2.3), que recibirá un objeto **DataController** con la lista de estaciones obtenida desde el CSV.

Un objeto **Station** consta de tres parámetros clave para su correcto funcionamiento:

```

private Map<String, Station> mapLines;
private Map<String, ArrayList<String>> mapNameStation;
private ArrayList<String> listNameStation;

```

Por un lado, disponemos de un objeto Map **mapLines** que define las estaciones cabeceras de las líneas de las que consta nuestro sistema. definiendo pares línea - estación para cada una de ellas. El segundo de esos elementos es otro mapa llamado **mapNameStation**, que ayudado del CSV crea una estructura de datos en la que, dado un nombre de estación, se dispone de las líneas que pasan por la estación mencionada. Dichas estaciones están definidas en forma del tercer parámetro importante, **listNameStation**, que es una estructura similar a los array típicos de Java, pero que posee unas características que optimizan la búsqueda de valores en esos array.

3.3.3.- Clase AStarController

```
package controllers;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Map;

import models.Station;
import net.datastructures.Entry;
import net.datastructures.PriorityQueue;
import net.datastructures.SortedListPriorityQueue;

public class AStarController {

    private DataController dataMap;
    private PriorityQueue<Double, Station> openList;
    private ArrayList<Station> closeList;
    private Station currentStation;

    public AStarController(DataController map){
        this.dataMap = map;
    }

    public ArrayList<String[]> calculate(String
src_station, String dst_station){

        ArrayList<Station> src_stations =
this.dataMap.getStationsByName(src_station);
        ArrayList<Station> dst_stations =
this.dataMap.getStationsByName(dst_station);

        AStarResult final_a_star_result = new
AStarResult(Math.pow(10.0, 6), null);
        for (Station source_station : src_stations) {
            for (Station destination_station :
dst_stations) {
                AStarResult a_star_result =
this.executeAStar(source_station, destination_station);
```

```
        if(a_star_result.time <
final_a_star_result.time){
            final_a_star_result =
a_star_result;
        }
    }
}

    ArrayList<String[]> route = new
ArrayList<String[]>();

    Integer total_time = 0;
    for (Station station : final_a_star_result.route)
{
        Double time = 0.0, min = 0.0, seg = 0.0;
        if (station.getParentStation() != null){
            time =
station.time(station.getParentStation());
        }
        min = time/60; seg = time%60; total_time +=
min.intValue()*60 + seg.intValue();
        String estacion = station.getLineStation();
        switch (estacion) {
            case "A":
                estacion = "1";
                break;
            case "B":
                estacion = "2";
                break;
            case "C":
                estacion = "3";
                break;
            case "D":
                estacion = "4";
                break;
            case "E":
                estacion = "5";
                break;
            case "F":
                estacion = "6";
```

```
        break;
    case "G":
        estacion = "7";
        break;
    case "H":
        estacion = "8";
        break;
    case "I":
        estacion = "9";
        break;
    case "J":
        estacion = "10";
        break;
    case "K":
        estacion = "11";
        break;
    case "L":
        estacion = "12";
        break;
    default:
        estacion = "-1";
        break;
    }
    route.add(new String[]{estacion,
station.getNameStation(), String.format("%02d min %02d
seg", min.intValue(), seg.intValue())});
    station.setParentStation(null);
}

    route.add(new String[]{String.format("%02d horas
%02d minutos %02d segundos", total_time/3600,
(total_time%3600)/60, (total_time%3600)%60)});

    return route;
}

    private AStarResult executeAStar(Station src_station,
Station dst_station){
        src_station.setParentStation(null);
        dst_station.setParentStation(null);
```

```
        this.openList = new
SortedListPriorityQueue<Double, Station>();
        this.closeList = new ArrayList<Station>();
        this.currentStation = src_station;

        Station last_station = null;
        double current_f = 0;

        double timeGx = 0;
        double timeG =
this.currentStation.time(src_station);
        double timeH =
this.currentStation.time(dst_station);

        this.openList.insert(timeG+timeH,
this.currentStation);

        while (!this.openList.isEmpty()) {
            last_station = this.currentStation;

            Entry<Double, Station> f_station =
this.openList.removeMin();
            this.currentStation = f_station.getValue();
            current_f = f_station.getKey();

while(this.closeList.contains(this.currentStation)){
                f_station = this.openList.removeMin();
                this.currentStation =
f_station.getValue();
                current_f = f_station.getKey();
            }

            this.closeList.add(this.currentStation);
            if
(this.currentStation.getNameStation().equals(dst_station.ge
tNameStation())){
                break;
            }
        }
```

```

        timeG = current_f -
this.currentStation.time(dst_station);

        Station prev_station =
this.currentStation.getPrevStation();
        if (prev_station != null && !
prev_station.equals(last_station) && !
this.closeList.contains(prev_station)) {
            timeGx = timeG +
this.currentStation.time(prev_station);
            timeH = prev_station.time(dst_station);
            this.updateOpenList(timeGx + timeH,
prev_station);
        }

        Station next_station =
this.currentStation.getNextStation();
        if (next_station != null && !
next_station.equals(last_station) && !
this.closeList.contains(next_station)) {
            timeGx = timeG +
this.currentStation.time(next_station);
            timeH = next_station.time(dst_station);
            this.updateOpenList(timeGx + timeH,
next_station);
        }

        Map<String, Integer>
current_transfer_station =
this.currentStation.getTransferStation();
        if(current_transfer_station != null){
            for (String line :
this.dataMap.getMapNameStation().get(this.currentStation.ge
tNameStation())) {
                Station transfer_station =
this.dataMap.getStation(line,
this.currentStation.getNameStation());
                if (!
this.currentStation.equals(transfer_station) && !

```

```

last_station.equals(transfer_station) && !
this.closeList.contains(transfer_station)){
    timeGx = timeG +
this.currentStation.time(transfer_station);
    timeH =
this.currentStation.time(dst_station);
    this.updateOpenList(timeGx +
timeH, transfer_station);
}
}
}
}

ArrayList<Station> route = new
ArrayList<Station>();
do {
    route.add(this.currentStation);
    this.currentStation =
this.currentStation.getParentStation();
} while (this.currentStation != null);
Collections.reverse(route);

return new AStartResult(timeG, route);
}

```

```

private void updateOpenList(Double f, Station station)
{
    PriorityQueue<Double, Station> aux_open_list =
new SortedListPriorityQueue<Double, Station>();
    boolean inserted = false;

    while(!this.openList.isEmpty()){
        Entry<Double, Station> f_station =
this.openList.removeMin();
        if(f_station.getValue().equals(station)){
            inserted = true;
            if (f_station.getKey() > f){
station.setParentStation(this.currentStation);
                aux_open_list.insert(f, station);
            }
        }
    }
}

```

```

        } else{
aux_open_list.insert(f_station.getKey(),
f_station.getValue());
        }
    }else{
aux_open_list.insert(f_station.getKey(),
f_station.getValue());
    }
}
    if (!inserted){
station.setParentStation(this.currentStation);
aux_open_list.insert(f, station);
    }
    this.openList = aux_open_list;
}
}

```

```

class AStarResult {
    public Double time;
    public ArrayList<Station> route;

    public AStarResult(Double time, ArrayList<Station>
route){
        this.time = time;
        this.route = route;
    }
}

```

La clase que recoge todo lo comentado anteriormente y en la que se realizan los cálculos del algoritmo es la clase **AStarController**. En esta clase se realiza la gestión de las listas abierta y cerrada necesarias para el correcto cálculo del camino mínimo entre las estaciones de origen y de destino.

Esta clase posee cuatro parámetros:

```
private DataController dataMap;  
private PriorityQueue<Double, Station> openList;  
private ArrayList<Station> closeList;  
private Station currentStation;
```

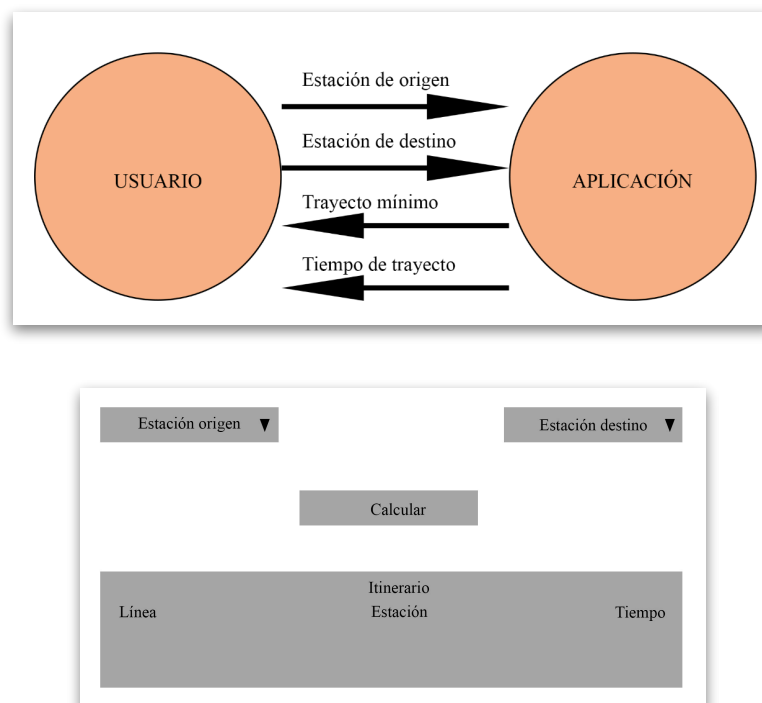
El objeto **dataMap** se corresponde con la descripción realizada en el apartado anterior. Por otro lado, se define la lista abierta como un objeto `PriorityQueue`, que relaciona la estación con el tiempo que emplea cada estación para llegar al destino. El tercer parámetro característico es la lista cerrada, definida como un objeto `ArrayList` de estaciones. En este caso no es necesario almacenar el tiempo que emplea al destino, puesto que una vez que una estación entra a la lista cerrada no debería salir de ella. Además, necesitamos conocer la estación actual sobre la que se está iterando, por lo que se ha utilizado un objeto **Station** auxiliar para gestionar ese apartado.

La lista abierta se actualiza mediante el método ***updateOpenList***, donde se realizan los cálculos pertinentes para comparar los méritos de cada estación. Por su parte, es el método ***executeAStar*** el encargado de llevar a cabo los cálculos necesarios para obtener el camino mínimo entre las dos estaciones que recibe por parámetro. Este método hace uso del método ***updateOpenList*** descrito anteriormente, además de los métodos de cálculo de distancias propios de la clase *Station*. Así pues, una vez se ha obtenido el camino mínimo, este se devuelve en forma de un objeto **AStarResult**, que contiene dos parámetros: el **tiempo total del trayecto** y la **lista de estaciones** que lo componen.

4.- DISEÑO

Debido a la gran cantidad de trabajo imprevisto derivada de los trabajos previos descritos en el punto 2, las tareas definidas en el plan de trabajo sufrieron una gran modificación. Dicha modificación consistió en la eliminación de uno de los dos ciclos de desarrollo previstos en el plan inicial. Es por esto que el diseño final satisface únicamente las funcionalidades básicas de la aplicación.

4.1.- Diseño de alto nivel



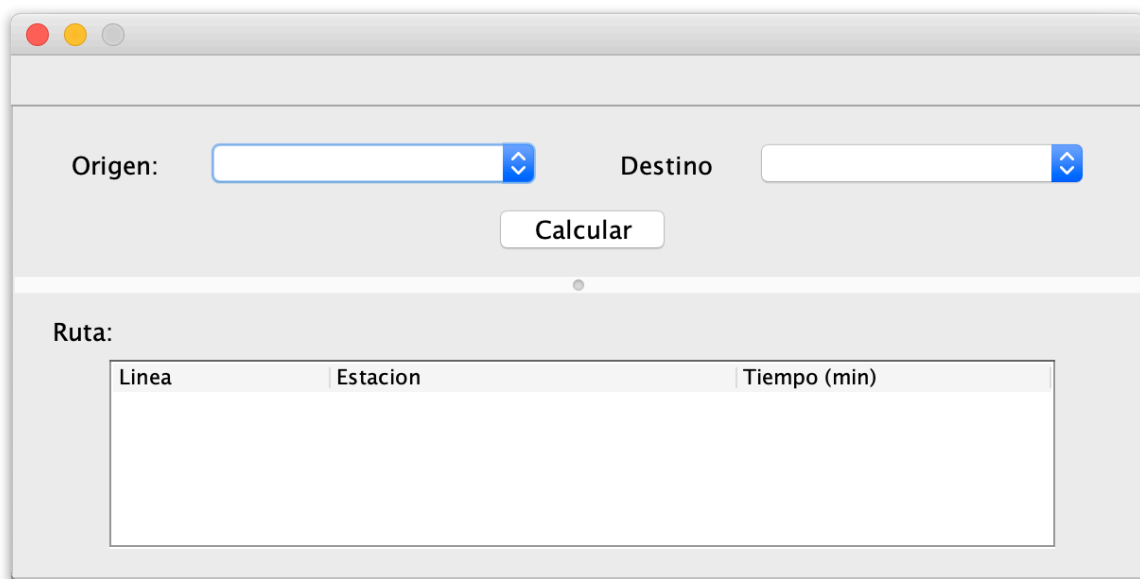
El diseño de bajo nivel define las funcionalidades básicas de la aplicación, pues elementos como un mapa 2D del trayecto formaban parte del planteamiento para la segunda fase de desarrollo. Estos elementos básicos son:

- **Estación origen:** Un desplegable con todas las estaciones de la red de Metro, en el cual el usuario de la aplicación debe seleccionar cuál de ellas se corresponde con el punto de origen desde el cual se emprende el camino.
- **Estación destino:** Un desplegable con todas las estaciones de la red de Metro, en el cual el usuario de la aplicación debe seleccionar cuál de ellas se corresponde con el punto de destino al cual debe dirigirse.

- **Calcular:** Un botón de calcular que, una vez seleccionadas las estaciones en el punto anterior, ejecute el algoritmo con las estaciones de origen y destino definidas en los despleguables
- **Itinerario:** Al pulsar el botón calcular, la sección de itinerario debe rellenarse con los datos que facilita el algoritmo. Estos datos arrojan por cada línea la estación por la que se pasa, así como la línea a la que pertenece y el tiempo empleado en recorrerla.

4.2.- Diseño de bajo nivel

El diseño de alto nivel se realizó directamente con la herramienta Eclipse, ya que en ella se encuentran todos los elementos que se podían utilizar posteriormente.



The screenshot shows a Java Swing window with a light gray background. At the top, there are two dropdown menus labeled 'Origen:' and 'Destino:'. Below these is a button labeled 'Calcular'. Underneath the button is a section labeled 'Ruta:' which contains a table. The table has three columns: 'Linea', 'Estacion', and 'Tiempo (min)'. The table is currently empty.

Linea	Estacion	Tiempo (min)
-------	----------	--------------

Para las listas desplegables de estaciones se utilizó la herramienta **JComboBox**, para el botón se definió como un **JButton**, mientras que los textos estáticos están definidos como **JLabel**. Además, la tabla en la que se mostrarán los trayectos es un objeto **JTable** dentro de un **JScrollPane** para que, en caso de que haya suficientes estaciones como para que no quepan a primera vista, el usuario se pueda desplazar hacia abajo y ver las estaciones que quedan en un principio fuera de la ventana, como se puede apreciar en la siguiente figura, donde, en la barra lateral, se aprecia la barra de desplazamiento que permite el funcionamiento anteriormente descrito.

Línea	Estación	Tiempo (min)
4	Alfonso XIII	00 min 00 seg
4	Prosperidad	01 min 36 seg
4	Avenida de América	01 min 14 seg
4	Diego de León	00 min 55 seg
5	Diego de León	01 min 30 seg

4.3.- Especificación de requisitos

FASE	REQUISITO
1	R01 - Introducir las estaciones de origen y destino
1	R02 - Rellenar los desplegables con las estaciones que contiene el CSV
1	R03 - Listar el camino óptimo dentro de una misma línea
1	R04 - Listar el camino óptimo cuando se realiza un transbordo entre estaciones
2	R05 - Dibujar el camino más rápido en el mapa
2	R06 - Seleccionar las estaciones de origen y destino en el mapa

La tabla anterior muestra los requisitos que se plantearon en un principio. Los requisitos son tareas unitarias que debe realizar la aplicación con el fin de llevar un control detallado de qué hay que hacer. Como se ha comentado en alguna ocasión durante la presente memoria, los imprevistos sufridos al generar los datos que alimentan la aplicación, solo se pudieron satisfacer los previstos en la fase 1.

5.- PRUEBAS UNITARIAS

Una vez finalizada la fase de desarrollo de la aplicación, se llevaron a cabo una serie de pruebas para corroborar que se cumplieron los requisitos de etapa planteados:

- **Prueba 1:** Las estaciones de origen se muestran correctamente por orden alfabético en el desplegable correspondiente.

The screenshot shows the 'Calculador de Trayectos - Metro de Madrid' application. The 'Origen:' dropdown menu is open, displaying a list of stations in alphabetical order: Abrantes (checked), Acacias, Aeropuerto T1-T2-T3, Aeropuerto T4, Alameda de Osuna (highlighted), Alcorcón Central, Alfonso XIII, and Almendrales. The 'Destino' dropdown is closed. Below the dropdowns is a 'Calcular' button. The 'Ruta:' section contains a table with columns 'Linea' and 'Tiempo (min)'. The table is currently empty.

- **Prueba 2:** Las estaciones de origen se muestran correctamente por orden alfabético en el desplegable correspondiente.

The screenshot shows the 'Calculador de Trayectos - Metro de Madrid' application. The 'Origen:' dropdown menu is closed and shows 'Abrantes'. The 'Destino' dropdown menu is open, displaying a list of stations in alphabetical order: Barajas, Barrio de la Concepción, Barrio del Pilar, Barrio del Puerto, Batán, Baunatal, Begoña (checked and highlighted), and Bilbao. The 'Calcular' button is visible between the dropdowns. The 'Ruta:' section contains a table with columns 'Linea', 'Estacion', and 'Tiempo (min)'. The table is currently empty.

- **Prueba 3:** Las estaciones se leen correctamente desde el CSV donde están definidas y se rellenan correctamente las cabeceras de línea.

```

<terminated> Launcher (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Contents/Home/bin/java
Estaciones:
[Abrantes, Acacias, Aeropuerto T1-T2-T3, Aeropuerto T4, Alameda de Osuna, Alcorcón Central, Alfonso XIII, Almendrales, Alonso
Cano, Alonso Martínez, Alonso de Mendoza, Alsacia, Alto de Extremadura, Alto del Arenal, Aluche, Alvarado, Antonio Machado,
Antón Martín, Arganda del Rey, Arganzuela - Planetario, Argüelles, Arroyo Culebro, Arroyofresno, Artilleros, Arturo Soria,
Ascao, Atocha Renfe, Avenida de América, Avenida de Guadalajara, Avenida de la Ilustración, Avenida de la Paz, Aviación
Española, Bambú, Banco de España, Barajas, Barrio de la Concepción, Barrio del Pilar, Barrio del Puerto, Batán, Baunatal,
Begoña, Bilbao, Buenos Aires, Callao, Campamento, Canal, Canillas, Canillejas, Carabanchel, Carabanchel Alto, Carpetana,
Cartagena, Casa de Campo, Casa del Reloj, Chamartín, Chueca, Ciudad Lineal, Ciudad Universitaria, Ciudad de los Ángeles,
Colombia, Colonia Jardín, Colón, Concha Espina, Conde de Casal, Congosto, Conservatorio, Coslada Central, Cruz del Rayo,
Cuatro Caminos, Cuatro Vientos, Cuzco, Delicias, Diego de León, Duque de Pastrana, El Bercial, El Capricho, El Carmen, El
Carrascal, El Casar, Embajadores, Empalme, Esperanza, Estación del Arte, Estadio Metropolitano, Estrecho, Estrella, Eugenia de
Montijo, Feria de Madrid, Francos Rodríguez, Fuencarral, Fuenlabrada Central, García Noblejas, Getafe Central, Goya, Gran Vía,
Gregorio Marañón, Guzmán el Bueno, Henares, Herrera Oria, Hortaleza, Hospital 12 de Octubre, Hospital Infanta Sofía, Hospital
Severo Ochoa, Hospital de Fuenlabrada, Hospital de Móstoles, Hospital del Henares, Ibiza, Iglesia, Islas Filipinas, Jarama,
Joaquín Vilumbrales, Juan de la Cierva, Julián Besteiro, La Almudena, La Elipa, La Fortuna, La Gavia, La Granja, La Latina, La
Moraleja, La Peseta, La Poveda, La Rambla, Locomo, Lago, Laguna, Las Musas, Las Rosas, Las Suertes, Las Tablas, Lavapiés,
Leganes Central, Legazpi, Lista, Loranca, Los Espartales, Lucero, Manoteras, Manuel Becerra, Manuel de Falla, Manuela
Malasaña, Mar de Cristal, Marqués de Vadillo, Marqués de Valdavia, Menéndez Pelayo, Miguel Hernández, Mirasierra, Moncloa,
Montecarmelo, Méndez Álvaro, Móstoles Central, Noviciado, Nueva Numancia, Nuevos Ministerios, Núñez de Balboa, Opañel, Oporto,
O'Donnell, Paco de Lucía, Pacífico, Palos de la Frontera, Pan Bendito, Parque Europa, Parque Lisboa, Parque Oeste, Parque de
Santa María, Parque de las Avenidas, Parque de los Estados, Pavones, Peñagrande, Pinar de Chamartín, Pinar del Rey, Pirámides,
Pitis, Plaza Elíptica, Plaza de Castilla, Plaza de España, Portazgo, Pradillo, Prosperidad, Príncipe Pío, Príncipe de Vergara,
Pueblo Nuevo, Puente de Vallecas, Puerta de Arganda, Puerta de Toledo, Puerta del Sur, Puerta del Ángel, Pío XII, Quevedo,
Quintana, República Argentina, Retiro, Reyes Católicos, Rivas Futura, Rivas Urbanizaciones, Rivas Vaciamadrid, Ronda de la
Comunicación, Rubén Darío, Ríos Rosas, Sainz de Baranda, San Bernardo, San Blas, San Cipriano, San Cristóbal, San Fermín -
Orcasur, San Fernando, San Francisco, San Lorenzo, San Nicasio, Santiago Bernabéu, Santo Domingo, Serrano, Sevilla, Sierra de
Guadalupe, Simancas, Sol, Suanzes, Tetuán, Tirso de Molina, Torre Arias, Tres Olivos, Tribunal, Universidad Rey Juan Carlos,
Urgel, Usera, Valdeacederas, Valdebernardo, Valdecarros, Valdezarza, Velázquez, Ventas, Ventilla, Ventura Rodríguez, Vicente
Aleixandre, Vicalvaro, Villa de Vallecas, Villaverde Alto, Villaverde Bajo, Vinateros, Vista Alegre, Ópera]
Estaciones cabeceras por línea:
{A=<A:Pinar de Chamartín>, B=<B:Las Rosas>, C=<C:Villaverde Alto>, D=<D:Argüelles>, E=<E:Alameda de Osuna>, F=<F:Laguna>,
G=<G:Hospital del Henares>, H=<H:Nuevos Ministerios>, I=<I:Paco de Lucía>, J=<J:Hospital Infanta Sofía>, K=<K:Plaza Elíptica>,
L=<L:Puerta del Sur>}

```

- **Prueba 4:** Se rellena correctamente la tabla de ruta, aunque su resultado no sea correcto.

Ruta: 00 horas 52 minutos 09 segundos

Línea	Estación	Tiempo (min)
5	Alonso Martínez	01 min 42 seg
5	Chueca	01 min 07 seg
5	Gran Vía	01 min 02 seg
5	Callao	00 min 45 seg
5	Ópera	00 min 49 seg

- **Prueba 5:** Se calcula correctamente un camino entre estaciones de la misma línea con estaciones de origen y destino **sin** transbordos.

Calculador de Trayectos – Metro de Madrid

Origen: Sevilla Destino: La Elipa

Calcular

Ruta: 00 horas 21 minutos 20 segundos

Línea	Estacion	Tiempo (min)
2	Sevilla	00 min 00 seg
2	Banco de España	00 min 55 seg
2	Retiro	01 min 51 seg
2	Príncipe de Vergara	01 min 17 seg
2	Goya	00 min 43 seg

- **Prueba 6:** Se calcula correctamente un camino entre estaciones de la misma línea con estaciones de origen y destino **con** transbordos.

Calculador de Trayectos – Metro de Madrid

Origen: Sol Destino: Tribunal

Calcular

Ruta: 00 horas 04 minutos 51 segundos

Línea	Estacion	Tiempo (min)
1	Sol	03 min 00 seg
1	Gran Vía	00 min 36 seg
1	Tribunal	01 min 15 seg

- **Prueba 7:** Se calcula correctamente un camino entre estaciones de distintas líneas separadas por un transbordo.

Calculador de Trayectos – Metro de Madrid

Origen: Destino:

Ruta: 00 horas 11 minutos 41 segundos

Línea	Estación	Tiempo (min)
12	San Nicasio	00 min 00 seg
12	Puerta del Sur	07 min 24 seg
10	Puerta del Sur	03 min 00 seg
10	Joaquín Vilumbrales	01 min 17 seg

- **Prueba 8:** Se calcula correctamente un camino entre estaciones de distintas líneas separadas por varios transbordos.

Calculador de Trayectos – Metro de Madrid

Origen: Destino:

Ruta: 00 horas 15 minutos 17 segundos

Línea	Estación	Tiempo (min)
11	Plaza Elíptica	02 min 05 seg
6	Plaza Elíptica	07 min 30 seg
6	Opañel	01 min 02 seg
6	Oporto	01 min 39 seg
5	Oporto	01 min 30 seg

6.- BIBLIOGRAFÍA

Definición y uso del algoritmo A*:

- Laboratorio de Infraestructuras de Datos Espaciales Universidad de Valladolid
<http://idelab.uva.es/algoritmo>
- Documentación de la asignatura Inteligencia Artificial FI UPM

Obtención de datos de las estaciones de Metro:

- Consorcio de Transportes de la Comunidad de Madrid
<https://data-crtm.opendata.arcgis.com/items/datos-abiertos-elementos-de-la-red-de-metro>
- Google Maps
maps.google.es
- Metro de Madrid
metromadrid.es

Estructuras de datos:

- Documentación de la asignatura Algoritmos y Estructura de Datos FI UPM