# Práctica Metro Monterrey

Inteligencia Artificial. Grupo 1

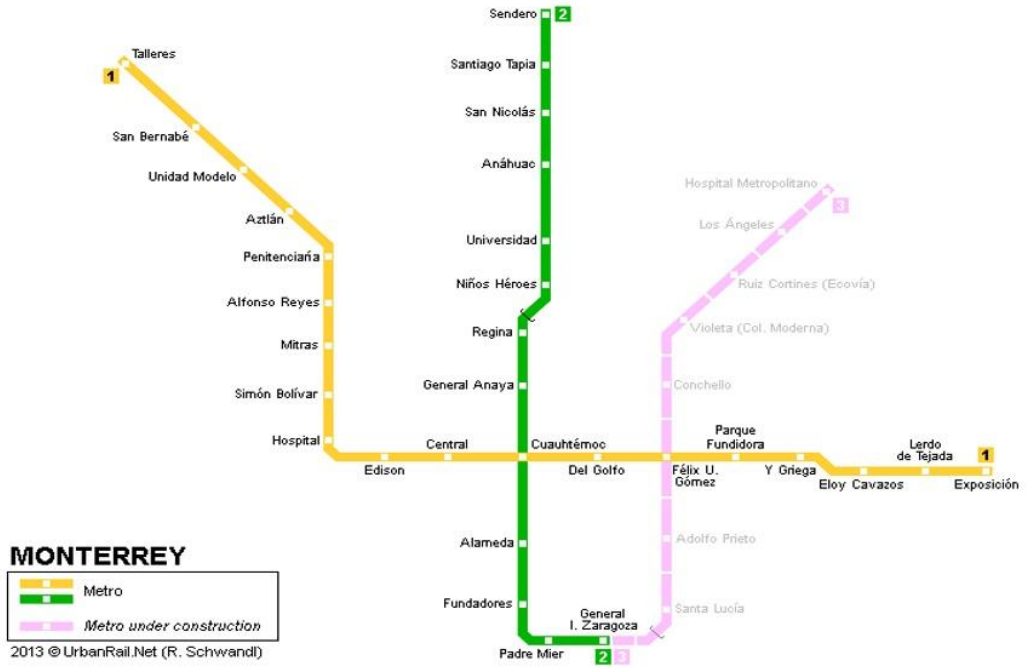
Geneviève Cirera
Juan Francisco Salamanca Carmona
Jaime Labiaga Ferrer

**03/06/2014**

# *Índice*

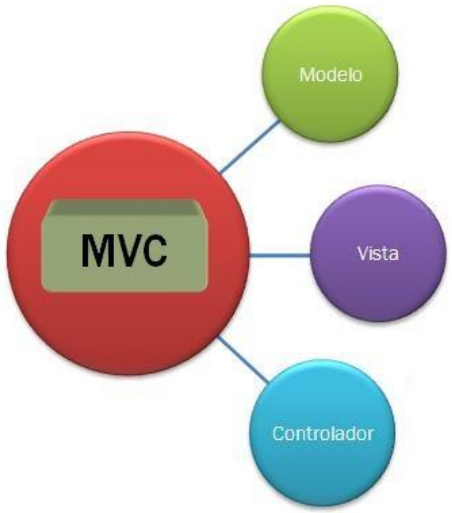
# *Objetivo*

Esta práctica consiste en el desarrollo de una aplicación que calcule el trayecto óptimo entre dos estaciones del metro de Monterrey. En nuestro caso, en lo que nos hemos centrado, ha sido en recorres la menor distancia entre estaciones.



# *Modelo de implementación elegido*

Hemos elegido implementar un modelo MVC (Modelo Vista Controlador).




- Modelo: trabaja con los datos y los controla. No necesita conocer cuál es el controlador o la vista.
- Vista: es la presentación al usuario a través del modelo. Lo que es la interfaz gráfica.
- Controlador: Depende de las órdenes del usuario para actuar sobre los datos, interactuando con el modelo considerando él si se realiza un cambio en la vista o en el modelo.

Este modelo lo hemos elegido porque se adecua muy bien con nuestra visión del proyecto. Una de sus principales ventajas es que permite independizar la vista del modelo, por lo que se puede cambiar fácilmente y sin tener que hacer modificaciones en el código del modelo o del controlador. Por tanto, si quisiéramos añadir una vista en el futuro también resultaría fácil.

Otra de sus ventajas es que este modelo permite trabajar en grupo de manera totalmente libre, ya que como comentábamos antes, sus tres módulos están separados y son completamente independientes los unos de los otros.

# *Desarrollo de la aplicación*

Para la **recogida de datos** hemos usado la web http://calculardistancias.info/aereas.html . Con esto hemos medido las distancias que hay entre cada estación respectivamente. Y las hemos volcado en el archivo data.txt, que se encuentra en src/resources.

- Estructura de data.txt:
  - Nombre de todas las estaciones del plano.

```
*Stations
Talleres:San Bernabé:Unidad Modelo:Aztlán:Penitenciaria
```

  - Distancias de cada estación con cada una de las estaciones.

```
%Talleres
0:690:1680:3000:4080:4800:5810:6500:7200:8090:8490:8980:9660:10380:10990
```

  - Vínculos.

```
;10:12:26:27
```

Respecto a la **implementación del algoritmo** para el cálculo del trayecto más óptimo hemos optado por el algoritmo A*.

Hemos desarrollado el proyecto usando eclipse, en lenguaje java. Y para la **implementación de la interfaz** hemos usado Swing.

# *División de clases*

- **Paquete controller**. Formada por todo lo relativo al controlador.
  - **Controller.java**: Es el controlador en el MVC. Se encarga de hacer el vínculo entre el modelo y la vista y poner los listeners. Contiene el método que se encarga empezar el buscar el trayecto óptimo. Se activa cuando pulsamos ok. Aquí se encuentra todo lo relativo al controlador
- **Paquete environment**. Contiene las siguientes clases:
  - **AlgoA.java:** Aquí está implementado el algoritmo A*. Se detalla su implementación más adelante.
  - **Main.java:** Este es el index, desde aquí se lanza la aplicación.

- **Model.java:** Se recogen el origen y destino y se llama al algoritmo. Se crean todos los objetos necesarios. Tiene toda la estructura y funcionalidad del proyecto.

  Limpia todas las estaciones de los posibles datos de anteriores búsquedas.

```
private void clearAllStation() {
    for (int i = 0; i < this.stations.length; i++) {
        this.stations[i].clear();
    }
}
```

  Con el método ok comprueba que las estaciones sean correctas y empieza la búsqueda. Luego pinta el camino.

```
public void ok(String origin, String destination) {
    // Encontrar las estaciones que tienen estos nombres
    clearAllStation();
    Station si = null, sf = null;
    for (int i = 0; i < stations.length; i++) {
        if (stations[i].getName().equals(origin)) {
            si = stations[i];
        }
        if (stations[i].getName().equals(destination)) {
            sf = stations[i];
        }
        if (sf != null && si != null)
            break;
    }
    if (sf == null || si == null) {
        System.out.println("Una de las dos estaciones no existen");
    }
    this.way = algo.findWay(si, sf);
    fireWayChanged();// Draw way
}
```

- **ParserData.java:** Esta clase se ocupa de volcar los datos guardados en data.txt (nombres, distancias y vínculos).

```
private final String FILENAME ="/resources/data.txt";
```

  Y agrega las estaciones la lista de estaciones.

```
case "*Stations":
    s = br.readLine();
    parseStations(s);
    break;


private void parseStations(String s) {
    String[] station = s.split(":");
    int n = station.length;
    this.distances = new int[n][n];
    this.stations = new Station[n];
    for (int i = 0; i < n; i++) {
        stations[i] = new Station(station[i], i);
    }
}
```

- **Station.java:** Clase que crea objeto “estación”, con su correspondiente id, nombre, vecinos, g y f.

- **Paquete graphicInterface**. Este paquete contiene todo lo referente al diseño gráfico de la aplicación. Consta de las siguientes clases:

  - **Map.java:** Es la vista en nuestro modelo MVC. Pinta el mapa, también el camino y da la posibilidad de elegir la estación de inicio y final. Si se da click en ok, da una señal al controlador con las estaciones origen y destino, el controlador llama al modelo para que ejecute los métodos necesarios para encontrar el camino, y una vez encontrado, da señal a la vista, pasándole el camino. La vista Map ya sólo tiene que mostrar este camino.

    Añade las posiciones al mapa.

    ```
    add(new Position(358, 451));// Edison
    ```

    Llama al controlador con origen y destino. Dibuja la ruta devuelta por él.

    ```
    this.drawCombo();
    this.draw(new ArrayList<Station>(), 0, 0);
    ```

    Draw carga todo lo relacionado con el mapa, y llama a drawCamino

    ```
    drawCamino(s);

    private void drawCamino(ArrayList<Station> stations) {
    ```

  - **Position.java:** Crea el objeto “position”, con valores x e y para representar las coordenadas de la imagen.

    ```
    public Position(int x, int y)
    ```

# *Implementación del algoritmo A**

Implementamos el algoritmo A* para realizar la búsqueda del camino más óptimo.

Consta de los siguientes métodos:

- **FindWay:**. Sus parámetros son origen y destino. Y sigue estos pasos:

  Se inicializan listas abierta y cerrada, se añade la estación origen a la lista abierta y se llama a método find con el origen como parámetro.

```
this.openedList.clear();
this.closedList.clear();
this.way.clear();

this.openedList.add(si);
find(si);
```

- **Find:** Su parámetro es la estación padre. Y sigue estos pasos:
  - Se calcula g y f.

```
for (int i = 0; i < padre.getNeighbours().size(); i++) {
    // Encuentra distancia entre padre e hijo
    int gPH = this.distances[padre.getId()][padre.getNeighbours().get(i).getId()];
    // Calcula g, la distancia hasta el padre + la distance padre-hijo
    int g = gPH + padre.getG();
    // Calcula f, g+ la distancia hasta el destino sf
    int f = g + this.distances[padre.getNeighbours().get(i).getId()][this.sf.getId()];
```

  - Si se encuentra un f menor, recalcula g, f y el camino.

```
if ((f < padre.getNeighbours().get(i).getF() && padre.getNeighbours().get(i).getF() != 0)
        || padre.getNeighbours().get(i).getF() == 0) {
    // Actualiza g y f
    padre.getNeighbours().get(i).setG(g);
    padre.getNeighbours().get(i).setF(f);
    // Actualiza el camino
    padre.getNeighbours().get(i).addWayToHere(padre.getWayToHere());
    padre.getNeighbours().get(i).addWayToHere(padre);
```

  - Se añaden los vecinos a la lista abierta.

```
almacenar(padre.getNeighbours());
```

  - Se reordena la lista abierta por orden de f.

```
ArrayList<Station> stationClone = (ArrayList<Station>) this.openedList.clone();
this.openedList.clear();
reordenar(stationClone);
```

  - Se pone el primer nodo de la lista abierta en la lista cerrada.

```
this.closedList.add(this.openedList.get(0));
this.openedList.remove(0);
```

- Se calcula f por cada hijo respectivamente. Si el último nodo puesto en la lista cerrada no es el nodo final, se llama de nuevo a find.

```
Station padre1 = this.closedList.get(this.closedList.size() - 1);
// Si el último nodo puesto en la lista cerrada no es el nodo final
if (!padre1.equals(sf)) {
    // Llamar de nuevo find
    find(padre1);
```

- Sino, significa que se ha llegado al destino y se añaden las estaciones del camino recorrido al camino (incluyendo el destino).

```
} else {// Si el último nodo de la lista cerrada es el destino
    this.way.addAll(padre1.getWayToHere());
    this.way.add(sf);
}
```

- **Almacenar:** Su parámetro es una lista de estaciones 's'.

  Pone en la lista abierta la estación siempre que no se encuentre ya en alguna lista.

```
public void almacenar(ArrayList<Station> s) {
    for (int i = 0; i < s.size(); i++) {
        if (!this.openedList.contains(s.get(i)) && !this.closedList.contains(s.get(i))) {
            this.openedList.add(s.get(i));
        }
    }
}
```

- **Reordenar:** Su parámetro es una lista de estaciones que llamamos 'sationClone'.

  Reordena la lista de estación por mínimo de f, y almacena la nueva lista en la lista abierta.

```
public void reordenar(ArrayList<Station> stationClone)
```

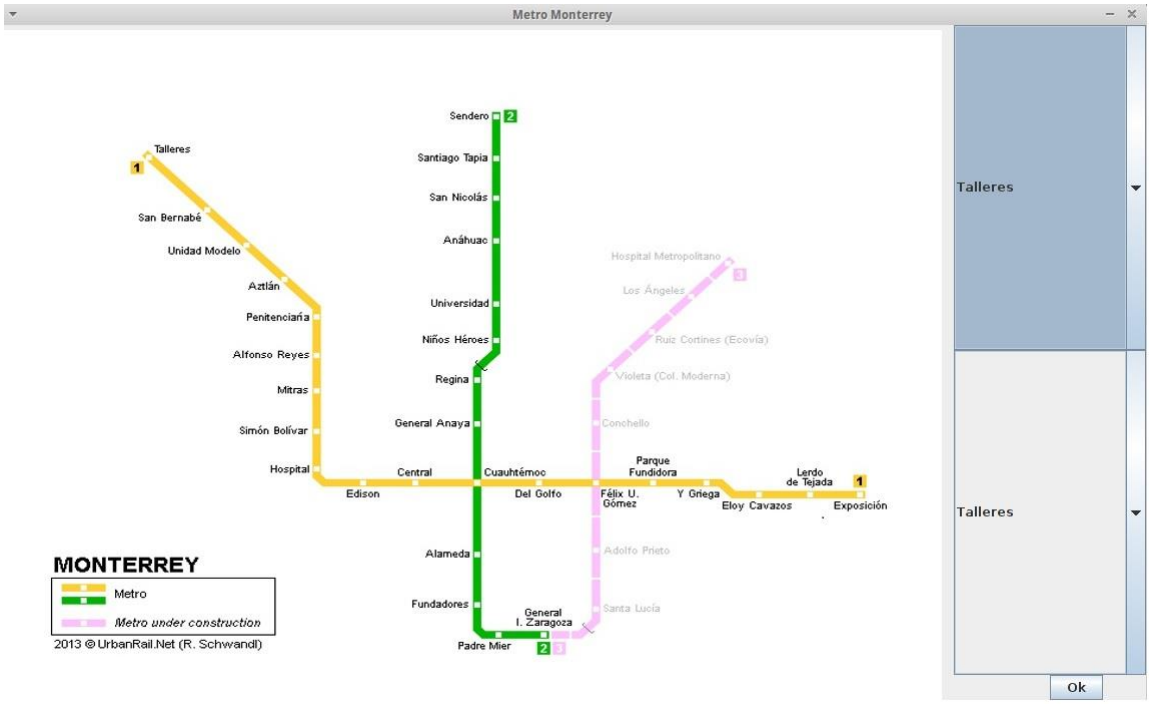
- **FindMin:** Su parámetro es una lista de estaciones 's'. Y sigue estos pasos:

  Devuelve el id de la estación que tiene el f mínimo en toda la lista

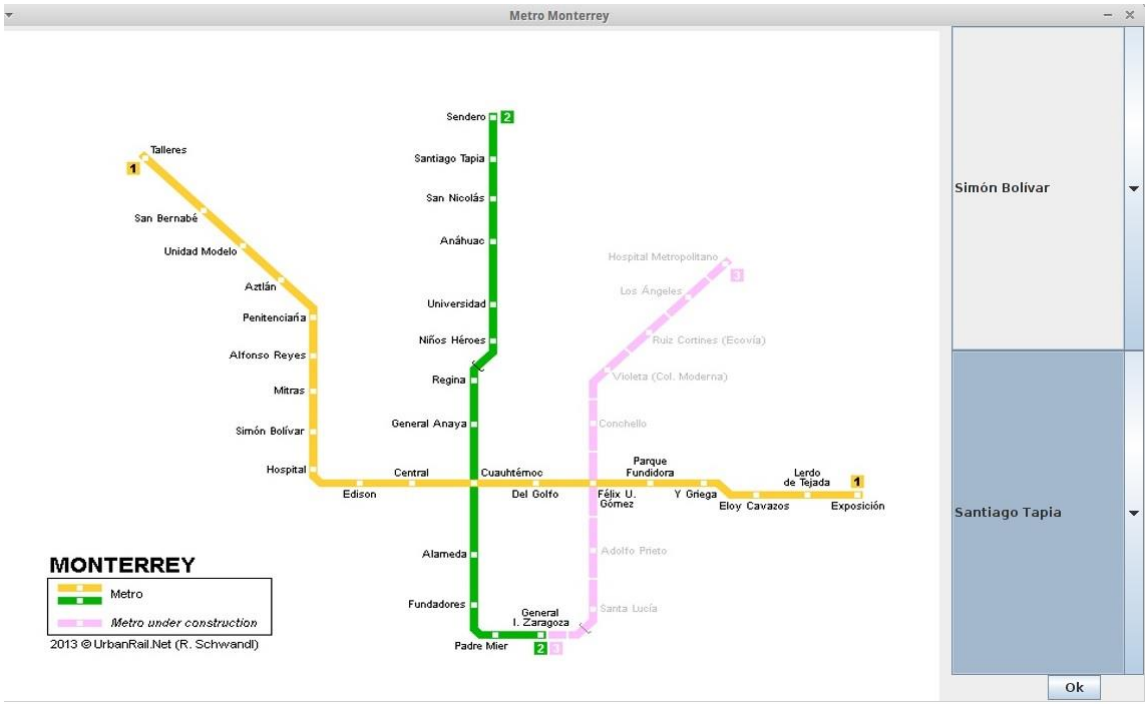
```
public int findMin(ArrayList<Station> s)
```

# *Pasos para la ejecución*

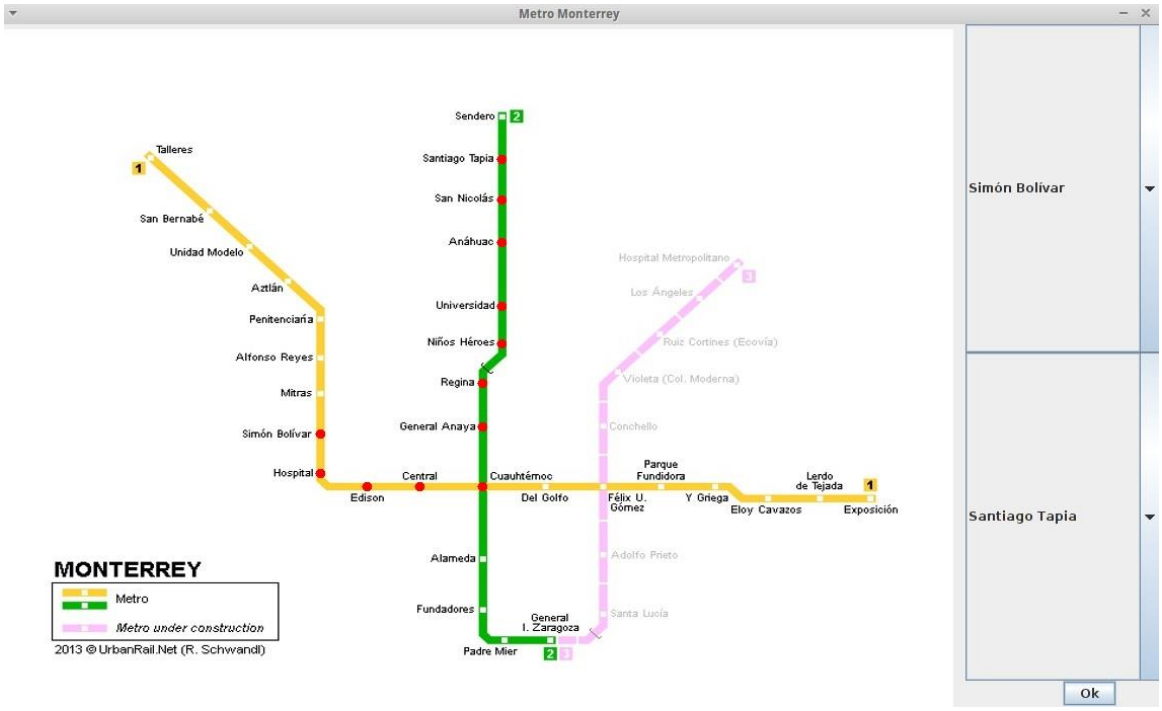
Al lanzar el programa, nos aparece la siguiente ventana:



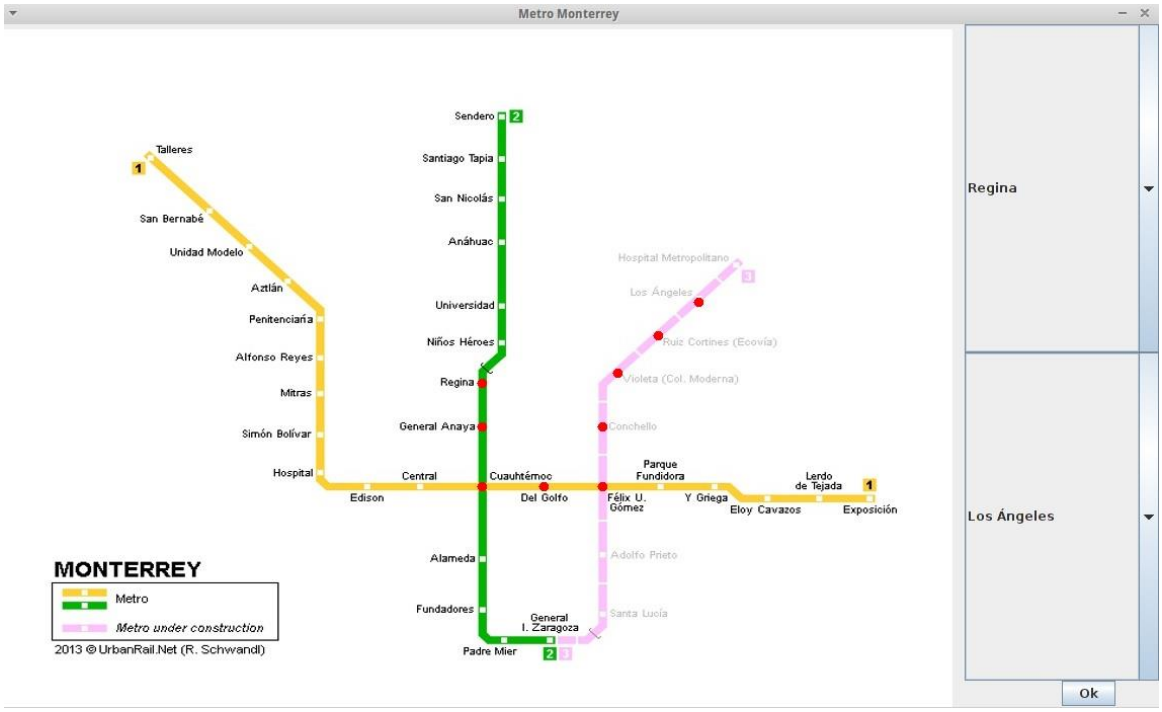

Disponemos de dos menús a la derecha (origen y destino). Vamos a elegir Simón Bolivar como origen y Santiago Tapia como destino.



Una vez elegidos origen y destino, si pulsamos el botón ok, nos trazará la ruta del trayecto más óptimo entre las dos estaciones.




En este otro ejemplo vemos el camino devuelto por el programa, pero en este caso involucrando a una de las estaciones que se encuentran en construcción.



# *Dificultades encontradas*

A la hora de medir las distancias de la línea en construcción, hemos tenido que tomarlas de manera aproximada.

Realizar el diagrama de clases para independizar las partes con el objetivo de poder trabajar más fácilmente juntos.

Aprendizaje de la realización del diseño de la interfaz gráfica.

Al crear el ejecutable tuvimos un problema con un fallo producido por la ruta de la carpeta resource.

Repartición de las tareas entre los miembros del equipo.

# *Bibliografía*


- http://calculardistancias.info/aereas.html