Pràctica 4: Recorregut de camins mínim

Jaume Adrover Fernandez, Diego Bermejo Cabañas i Joan Balaguer Llagostera

***Resum*—En aquest document s'explicarà el contingut de la pràctica 4 sobre trobar el camí més curt entre dos pobles d'Eivissa i Formentera, passant per un altre obligatòriament. Explicarem com hem afrontat el problema, quines solucions hem implementat, el codi desenvolupat i finalment unes conclusions amb els problemes que ens hem trobat i algunes reflexions que hem fet.**

1 Introducció

En aquest article explicarem el funcionament i la implementació de la nostra pràctica, estructurant el document de tal forma que es cobreixi la totalitat de aspectes rellevants d'aquesta.

Per una part explicarem l'estructura del MVC que hem relitzat i mostrarem analíticament com funciona aquesta estructura durant l'execució del programa.

A continuació, explicarem breument el problema que ens han proposat resoldre. Mostrarem analíticament les dues solucions que hem implementat per resoldre el problema, juntament amb explicacions en pseudocodi perquè posteriorment es pugui entendre millor el codi que hem implementat

Després, mostrarem el codi implemementat per a resoldre el problema tal i com l'hem explicat a l'apartat anterior.

Finalment, explicarem com funciona la nostra aplicació una vegada s'executa, a més del problemes que ens hem trobat i com els hem acabat resolguent.

2 Implementació del MVC

Una estructura MVC (Model Vista Controlador), consta de 3 grans parts:

- **Model**: és l'encarregat d'emmagatzemar les dades del programa.
- **Vista**: s'encarrega de mostrar la interfície d'usuari amb les dades del model.
- **Controlador**: encarregat de notificar canvis de la vista entre el programa principal i el model de dades.

Una vegada sabem el que simbolitza cada part de l'estructura, podem entrar més en detall en el que fa cada una d'aquestes a la nostra pràctica. Per entendre-ho millor, exemplificarem el paper que jugarà cada una de les parts quan executem el programa.

- Una vegada es posa en marxa el programa principal, aquest, crear el model de dades i la vista. D'aquesta forma, es mostra al panell la imatge del mapa d'Eivissa i Formentera, sense cap tipus de nodes ni arestes. El controlador s'inicialitza a null, ja que de moment no s'hauran produït canvis a la vista.
- L'estructura, esdevindrà d'aquesta forma fins al moment en què l'usuari cliqui un dels 3 botons de la finestra (ja que és l'única forma que té d'interactuar amb l'aplicació i que es produeixin canvis a aquesta). A l'hora de clicar un botó, la vista avisa al programa principal, que es la central responsable dels canvis.
- Una vegada el programa principal sap quin dels botons s'ha clicat, aquest se li notificarà i actuarà en conseqüència (els casos específics s'explicaran quan expliquem el programa principal).
- Quan el programa principal rep la instrucció de lectura de fitxer, aquest s'encarrega d'inserir al model l'estructura d'un graf. Això es fa mitjançant un arxiu *XML*, que conté una enumeració de tots els pobles i les carreteres pertanyents a aquest mapa. Tota aquesta informació es troba a una instància de la classe *Mapa*.
- A continuació, l'usuari ha de seleccionar 3 pobles diferents, per a poder calcular una ruta entre el primer i el darrer, passant per l'intermedi. Aquest camí sirà el més curt possible. En haver seleccionat els 3 pobles diferents, clicam el botó de calcular.
- El controlador notifica a la classe *Dijkstra*, que ha d'intentar trobar una solució amb les dades que hi ha al model. D'aquesta manera, s'anirà recorrent el graf fins a trobar un vector amb les distàncies mínimes des d'un origen determinat.
- Una vegada acabi l'execució de l'algorisme, les dades del model ja estaran actualitzades i des del Controlador notificarem a la Vista que torni a repintar el panell on es mostra el mapa.
- Finalment, en cas de voler tornar a calcular una altra ruta des de pobles diferents, clicam el botó de Reset i podrem tornar a seleccionar 3 pobles diferents

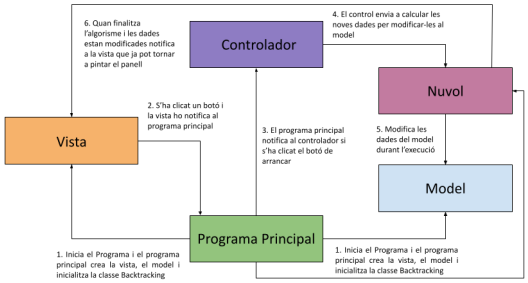

Figura 1. Esquema de com funciona el nostra MVC

3 Implementació de la solució

3.1 Problemes a resoldre

Durant el desenvolupament d'aquesta pràctica, hem tingut diferents inconvenients/dubtes. Vegeu la llista:

- **Obtenció del graf**: un dels nostres dubtes principals a l'hora de obtenir el graf era com podíem obtenir tota la informació d'alguna manera eficient.
- **Conversió a estructura adient**: hi ha moltes variacions de l'algoritme Dijkstra però no sabíem quina estructura havíem d'utilitzar per al recorregut d'aquest.
- **Representació visual**: una altra sospita que vàrem tenir va ser a l'hora de representar visualment el camí, ja que no sabíem com afrontar un cas bastant concret: quan es passa per un poble dos pics. De la manera inicial només es pintaria la carretera damunt la pintada anterior.

3.2 Solucions adoptades

- **Redacció arxiu XML**: tot i haver cercat per tot internet maneres més eficients de generar un graf a partir d'una secció de Google Maps, vàrem haver de fer una redacció d'un arxiu XML, manualment.La nostra metodologia de feina va ser la següent:
  - **Joan**: ell s'encarregava de tenir una imatge del mapa i anar dictant els pobles i les carreteres, pintant les que ja haviem afegit.
  - **Diego**: encarregat de cercar la distància en km donats dos pobles al Google Maps.
  - **Jaume**: encarregat d'escriure tota la informació dictada pels anteriors dedins l'arxiu *poples.ltim*.

  És important considerar que si s'haguès trobat una metodologia més simple per obtenir aquests fitxers, sense haver de redactar-los, es podrien implementar mapes més complexos i grans.

  Cal mencionar un factor molt important i es que per poder implementar la solució de Dijkstra haurem de tenir un **graf dirigit**, encara que a l'XML tinguem una única carretera amb el seun valor per a cada parella de pobles connectats a l'hora de carregar cada aresta afegim DOS arestes per a cada carretera llegida de l'arxiu. Amb aquesta solució representam que totes les carreteres tenen doble sentit.

- **Conversió de dues llistes a matriu**: Posterior a la lectura de l'arxiu amb la informació del graf, obteníem dues llistes amb tots els nodes i totes les arestes.Després, vàrem considerar que la millor manera de recórrer el graf era mitjançant una matriu d'adjacència. Aquí l'algoritme que ens permet crear una matriu d'adjacència a partir de dues llistes:

Algorithm 1 Algorisme per crear matriu adj.

```
1: int dim ← numPobles
2: for i ← 0 to dim do
3:     for j ← 0 to dim do
4:         matrix_{i,j} ← ∞
5:     end for
6:     matrix_{i,i} ← 0
7: end for
8:
9: int i← 0
10: for ∀ poble ∈ pobles do
11:     for ∀ carretera ∈ poble.carreteres do
12:         int idx ← carretera.apunta().getIndex()
13:         double valor ← carretera.dist
14:         matriu_{i,idx} ← valor
15:     end for
16:     i++
17: end for
```

Com podem apreciar, primer s'inicialitza la matriu amb 0's a les diagonals i ∞ a la resta. Finalment, actualitzam dins la fila del poble actual la posició del veinat corresponent. Per exemple:

Estam observant el poble de Sóller, és a dir, estam al segon bucle. Allà podem trobar com a veinat Deià, a distància d. De cada poble tenim emmagatzemat la seva posició dins un atribut i així, tenguent la variable i(número de files) i la anterior, podem trobar la posició de la carretera entre ambdós pobles. Si Sóller està en la 3a fila i Deià a la 4a, la variable $i \leftarrow 2$ i $idx \leftarrow 3$. Recordam que els indexos sempre comencen per 0 i, per tant, tenen un valor inferior en 1. El valor de $matriu_{i,idx} \leftarrow valor$, on valor és la distància d de la carretera de Sóller-Deià.

- **Enumeració d'ordre de pobles**: per a solucionar la problemàtica de la representació, dibuixam devora cada poble l'ordre d'arribada en la ruta final.

  Per exemple:

  $v_1 = \{x_i, x_{i+1}, ..., x_n | x_i \in mapa.pobles\}$

  $v_2 = \{x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_1 \rightarrow x_4\}$

Podem observar que, en la ruta(vector 2), passam dos pics per x_1. Per tant, aquest poble, tendrà dibuixat en el mapa els nombres 1 i 4.

3.3 Resolució general

Després d'haver evaluat tots els problemes sorgits i les seves solucions, anem a comentar la nostra proposta d'algoritme per a la resolució de l'enunciat. Per a trobar la ruta óptima entre dos pobles passant per un tercer, podem aplicar la tècnica de "dividir i vèncer". Primerament, executarem dijkstra per a arribar del primer poble fins al segon, i, posteriorment, del segon fins al tercer. Suposem el següent predicat:

$$\exists \quad ruta \quad A \to C \quad | \quad B \in ruta$$

D'aquesta manera juntarem dos camins, el més curt de **A**→**B**, i el més curt de **B**→**C**. Per tant, en conclusió, hem d'aplicar l'algoritme àvid de Dijkstra dos cops.

Posant-mos en context, després d'haver clicat el botó de calcula, el programa principal rep la notificació de calcular i avisa al control. El control fa aquesta funció:

Algorithm 2 Mètode Run del fil Control

```
1: if msg is 'Calcula' then
2:     model.trobarCami(A,B,1); //Trobar primer camí
3:     model.trobarCami(B,C,2) //Trobar segon camí
4:     model.fusionaCamins(); //Juntam els dos anteriors
5:     model.setSolucionat(true);
6:     model.notifica("Repintar");
7:     model.setSolucionat(false);
8: end if
```

Primer de tot, es comprova que el missatge rebut correspon a l'ordre de càlcul. Si és així, cridam al model per a que trobi el primer camí(línia 2) i el segon(línia 3). Aquests s'emmagatzemen en dos llistes dins model(veure explicació posterior). El nombre enter que es passa com a darrer paràmetre correspon a quin camí és. Si és un 1, correspon al primer camí.

Cal destacar que el mètode trobarCamí executa Dijkstra i posteriorment fa el recorregut des de el destí fins a l'origen per obtenir el camí amb el cost corresponent. Es fa un .reverse de la llista ja que sinos tendríem el camí des de el final fins l'origen. El resultat queda així:

$$cami_i = \{x_0, x_1, ..., x_n | origen = x_0 \quad \wedge \quad desti = x_n\}$$

Aquesta llista podria quedar en ordre contrari però preferíem iterar sobre ella ascendentment. A continuació, concatenam les dues llistes amb els mètodes *.addAll* de Java i ens queda un vector amb la següent estructura:

$$camiFinal = \{cami_1 \cdot cami_2\}$$

Aquesta llista ens servirà per a representar posteriorment la solució damunt el mapa.

3.4 Algoritme de Dijkstra

Finalment, anem a centrar-nos el l'algoritme de Dijkstra. La nostra implementació ve donada en la classe *Dijkstra.java*, aïllada de la resta del patró de MVC. Anem a veure els atributs que conté:

```
private Model mod;
private int numPobles;
```

Necessitem tenir comunicació amb el model per a poder obtenir la informació corresponent dels pobles i poder "enviar-li·la solució trobada. Anem a centrar-nos ara en els mètodes:

```
private int minDist(double dist[],
    bool sptSet[]);
public void trobarPesos(int src,
    int desti,int c);
public void trobarCami(int src,
    int desti,int c);
```

- **minDist(double dist[], Boolean sptSet[])**: aquest mètode, donat l'array dist de solucions i un array de pobles visitats, ens retorna l'índex del poble amb menor distància i no visitat. Es va iterant sobre l'array dist, que conté les distàncies als pobles corresponents des de l'origen. Si es troba un poble no visitat amb distància menor que totes les trobades, actualitzam la variable que conté l'índex a retornar.

- **trobarPesos(int src,int desti,int c)**:aquest mètode és l'algoritme de Dijkstra, que explicarem amb més detall posteriorment.

- **trobarCami(int src,int desti,int c)**:un cop trobada la solució, que vendrà representada en un vector de nombres *double*, miram d'iterar des del destí fins l'origen, emmagatzemant tots els pobles per on hem passat.

3.4.1 Trobar pesos

Anem a veure com hem implementat l'algoritme de Dijkstra mitjançant aquest mètode:

Algorithm 3 Algoritme Dijkstra

```
1: procedure TROBARPESOS(src, desti, c)
2:     dist[] ← double array[numPobles]
3:     sptSet[] ← boolean array[numPobles]
4:
5:     for i ← 0 to numPobles − 1 do
6:         dist[i] ← ∞;//init distancies
7:         sptSet[i] ← false;//init pobles visitats
8:     end for
9:
10:    dist[src] ← 0 //distància origen a origen és 0
11:    for count ← 0 to numPobles − 1 do
12:        u ←minDist(dist,sptSet); //index poble mínim
13:        sptSet[u] ← true; //poble visitat
14:        for v ← 0 to numPobles − 1 do
15:            if poble no visitat i dist menor then
16:                dist[v] ← dist[u] + mod.getPes(u, v);
17:            end if
18:        end for
19:    end for
20:
21:    mod.setSol(dist); //Actualitzam solució a model
22:    trobarCami(src, desti, c);//trobam camí
23: end procedure
```

Com podem apreciar, primer inicilitzam els dos vectors, els de distàncies a infinit excepte el de l'origen, que té cost inicial 0. En canvi, l'altre és tot false ja que no sha visitat cap poble encara.

A continuació, anam recorreguent en un bucle i obtenint tots els pobles amb índex mínim. Dedins el segon bucle interior, miram si podem actualitzar distàncies menors a les que tenim des de les carreteres del poble actual. Si un poble no és visitat i té una distància acceptable(diferent de 0, infinit i menor a l'actual), actualitzam el vector de distàncies amb el cost actual més el cost d'arribar al poble. Aquest cost per arribar al poble es mira a través del model, a una matriu d'adjacència on miram l'element i,j (mod.getPes(i,j). Això obtindrà l'element de la fila i i columna j. La solució vendrà donada en un vector com el següent:

$$v_i = \{x_0, x_1, ...x_n| \quad \exists! x_i = 0 \wedge x_k \in \mathbb{R} > 0\}$$

És a dir, existeix una posició que conté el valor 0, que vendria a ser l'origen, i les altres siran nombres reals majors a 0. Aquests darrers representaran les distàncies del poble origen cap al poble final.

4 Model

A la classe model tenim diferentes estructures que explicarem més endavant per a poder emmmagatzemar les dades que necesitarem. Aquestes són:

```
private ArrayList<String> seleccionats;
private final ibizaDjikstra prog;
private final Mapa map;
private boolean solucionat;
private boolean llegit;
private double matriu[][];
private double sol[];
private ArrayList<Integer> cami1;
private ArrayList<Integer> cami2;
private ArrayList<Integer> camiFinal;
```

- `seleccionats`: En aquest arraylist contenim el nom dels tres pobles que l'usuari ha seleccionat, essent la segona posició el poble per el que hem de passar i el primer i tercer representen l'inici i el destí respectivament.
- `prog`: Aquesta es una instància de la classe principal.
- `map`: aquest es l'objecte que conté el graf.
- `solucionat`: valor booleà que ens indica si ja s'ha trobat la solució, sirem notificats.
- `llegit`: valor booleà que igualments ens indica si el fitxer ja ha estat llegit.
- `matriu`: representa la matriu d'adjacéncia aquesta és una representació estructurada que mostra les connexions entre els diferents pobles a través de les carreteres existents, juntament amb les distàncies de les carreteres. Es tracta d'una matriu bidimensional on les files i les columnes representen els pobles, i els valors de la matriu indiquen la distància en quilòmetres entre els pobles conectats per les carreteres. Si hi ha una carretera entre dos pobles, el valor serà la distància en quilòmetres de la carretera. En cas contrari, el valor serà un valor elevat com `Double.MAX_VALUE` per indicar que no hi ha cap carretera directa entre els pobles.
- `sol`: En aquest array emmagatzemam els diferents costs de la solució trobada, es a dir, les diferentes distàncies entre els pobles.
- `cami1, cami2 y camiFinal`: el primer dels camins representa el camí mínim del poble inicial al poble intermig seleccionat per el que hem de passa, mentre que el segon camí ens mostra el que queda fins al poble de destí. Aquestes estructures obtenen les dades del pobles a partir d'un index que explicarem més endavant a l'apartat de la classe poble. Com es pot intuir el darrer atribut conté el camí mínim final que despres el convertirem per obtenir la solució sense indexos. Per a poder obtenir aques camí final, tenim un métode anomenat **FusionaCamins()** que junta cami1 i cami2.

Passem ara a explicar els métodes i funcions del Model són principalment *getters i setters*, com per exemple `getPes(int i, int j)`, que obté una posició de la matriu d'adjacéncia que representa el pes d'aquells dos pobles. A més també contenim en aquesta secció la imatge "mapa.jpg"que representarem.

4.1 Poble

La classe Poble.java emmagatzema les dades individuals d'un poble, per a poder ser utilitzades al model:

- `String nom`: Com el propi nom indica aquest atribut conté el nom del poble
- `ArrayList <Carretera> salientes`: aquesta es una estructura clau, ja que emmagatzema totes les carreteres que connecten al poble per a que pugui ser accesible
- `int X`: aquesta es la coordenan X del poble al mapa.
- `int Y`: aquesta es la coordenan Y del poble al mapa.
- `int index`: Amb aquest atribut podem identificar al poble a les diferentes estructures de dades que utilitzem al model.

Els métodes d'aquesta classe son principalment `getters i setters`, també trobam el de afegir aresta o un `equals` que comproba si dos pobels són iguals a partir del string que representa el nom.

4.2 Carretera

La classe carretera representa únicament les dades que han de ser contenides a una aresta, els atributs són:

- `Poble apunta`: representa l'únic poble al que esta conectat, recordam que el graf es dirigit i que tindrem dues carreteres per cada aresta.
- `double valor`: com el propi nom indica representa el cost pertinent d'aquesta carretera, que simula la distància en quilòmetres

Cal mencionar que les dues carreteres que tindrem per a cada aresta tindran el mateix cost.

4.3 Mapa

Aquesta classe conté totes les dades per a emmagatzemar el graf. Els atributs són:

- `BufferedImage img`: Aquesta es la imatge de les illes que mostrarem i sobre la que representam el graf.
- `ArrayList<Poble> pobles`: Aquests son els pobles que representen el nodes del graf.
- `ArrayList<Carretera> carreteras`: Com es pot imaginar les carreteres representes les arestes del graf dirigt, es a dir, mostram dues carreteres per a acada aresta.

Els métodes d'aquesta classe són els que ens permeten crear i manipular el graf dirigit:

5 Vista

La Vista, té com a funció, dintre de l'estructura del programa, mostrar a l'usuari mitjançant un GUI les dades que actualment es troben al model. Els atributs que pertanyen a la classe principal de la `Vista` són el següents:

```
private final ibizaDjikstra prog;
private final Panell panell;
private String fitxer;
```

- `prog`: atribut que és una instancia del programa principal. Amb ell, podrem obtenir l'atribut `mod` amb el que podrem obtenir la informació continguda dins aquest.
- `panell`: atribut que representa l'objecte `Panell` on dibuixarem tots els punts que generem.
- `fitxer`: atribut per guardar el nom del fitxer on contenim les dades que mostrarem sobre el mapa.

Pel que fa als mètodes de la classe simplement són 5:

```
public void mostrar();
public void afegeixComponennts();
public void actionPerformed(ActionEvent e);
public void notificar(String s);
public void mouseClicked(MouseEvent e);
```

- `mostrar()`: aquest mètode bàsicament crida a totes les funcions encarregades de generar i mostrar les components a pintar
- `afegeixComponents()`: en aquesta funció cream els botons a les posicions dels pobles per a que l'usuari els pugui seleccionar. Cal mencionar que aquesta funció s'ha de cridar una vegada hem llegit el fitxer i el model conté ja l'estructura amb tots els pobles i les seves coordenades que es troben a la classe.
- `actionPerformed(ActionEvent e)` : mètode que s'executa quan algun dels botons de la GUI es clica. Emmagatzema el text que està escrit al botó i l'envia com a comanda al programa principal (que el tenim emmagatzemat a l'atribut `prog`).
- `notificar(String s)`: mètode que utilitzam per notificar a la vista que ha de repintar el panell on es troben els punts pitats. Si es crida aquest mètode, voldrà dir que les dades del model han estat actualitzades.
- `mouseClicked(MouseEvent e)`: En aquesta funció agafarem els botons seleccionats de model per a que es puguin pintar de manera que es distingueixin de la resta.

5.1 Classe Panell

La classe `Panell` és a on es pintaran els punts que es mostraran a la GUI. Dins aquesta classe és on es produïran

els canvis de color dels punts que pertoquin y del camí que contengui el model, una vegada estigui solucionat. Els atributs que defineixen el `Panell` són els següents:

```
private final Model mod;
private ArrayList<JButton> botonsPobles;
```

- `mod`: atribuit que és una instancia del model del dades. Amb ell, podem extreure la informació del model de dades amb la informació dels punts i el camí que hem de mostrar.

- `botonsPobles`: arraylist amb el contingut del botó de tots els pobles que es troben a la `Vista`

Pel que fa als mètodes que gestionen el `Panell`, són els següents:

```
public Panell(int w, int h, Model m);
public void paint(Graphics g);
public void setBotonsPobles(JButton b)
```

- `Panell(int w, int h, Model m)`: mètode constructor de la classe. Per paràmetre s'indica l'amplada (`w`) i l'altura (`h`) de la finestra. A más, també es passa l'instancia del model de dades amb el qual inicialitzarem l'atribut `mod` i s'inicialitza l'arrraylist de botons vuit

- `paint(Graphics g)`: la primera acció que realitzam en aquesta funció es dibuixar la imatge que conté el Model sobre la finestra que mostrarem.

  Una vegada hem fet això el que hem de realitzar a continuació es pintar els pobles i les carreteres inicials amb els seus costs. Per poder fer això haurem de iterar sobre els pobles que conté el model comprovant cada vegada si aquest poble ha estat seleccionat o no, a més a més, obtindrem les seves aristes contingudes al poble y les pintarem com a linies a partir de les coordenades.

  El següent que hem de fer es pintar la solució en cas de que aquesta hagi estat obtinguda. Primerament obtindrem dos arraylist paralels, un amb els indexos dels pobles de la solució, i l'altre contindrá el número de passada d'aquell poble que es trobi a l'index proposat. Aquesta estructura emmagatzemara els nombres en tipus String degut a que ha de estar formatetjat en cas de que passem dues vegades per el mateix poble no es mostri un nombre d'amunt l'altre, sino de la forma "3,8", així mostram que aquest es el tercer poble i despres hem donat la volta i també ha estat el vuité del camí. Per implementar això hem realitzar un algorisme de complexitat tribial, simplement consultam l'array de pobles i afegim el número de vegades que apareix amb l'string i eliminam en el moment en que es repeteix.

  Una vegada que ja tenim l'informació hem de mostrar la solució, de la mateixa forma que mostram les carreteres totals peró unicament les dels pobles del camins de la solució i cambiant el color, també pintam l'string amb el nombre abans calculat. Per últim mostram un JOptionPane on mostram el cost total del camí seleccionat per l'úsuari.

- `setBotonsPobles(JButton b)`: Aquest métode simplement inicialitza els pobles

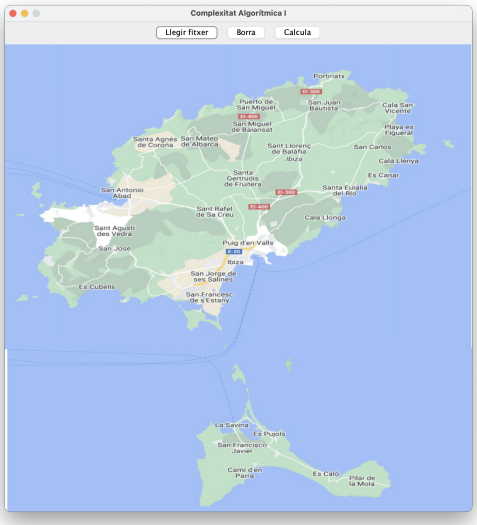

Figura 2. Vista general de la nostra GUI

6 Controlador

El controlador és l'encarregat de rebre l'instruccio del programa principal i posar en funcionament la classe Nuvol explicada en apartats anteriors. Depenent de la comanda d'entrada, s'executara un o un altre mètode de la classe. Els atributs de la classe són els següents:

```
private final ibizaDjikstra prog;
private String msg;
private Dijkstra d;
```

- `prog`: es tracta de una instancia del programa principal la qual utilitzarem per obtenir les dades que es troben al model de dades.

- `msg`: variable de tipus `String` que conté el missatge que rep el mètode `notifica()` del Controlador.

- `d`: instancia de la classe `Dijkstra` que ens serveix per començar a executar el calculs referents a trobar el camí mínim segons les dades que hi hagin introduïdes en aquell moment en el Model.

Pel que fa als mètodes del Controlador, tenim els següents:

```
public Control(ibizaDjikstra p);
public void run();
public void notificar(String s);
```

- `Control()`: mètode constructor de la classe amb el qual instanciam l'atribut del programa principal i inicialitzam l'objecte de la classe Dijkstra.

- `run()`: mètode que s'executa quan es crida al mètode `start()`. Aquest, actua en conceqüencia quan li arriba l'string amb la paraula *Calcula*. En aquest cas, voldrà dir que l'usuari ha carregat el contingut del XML i ha seleccionat els 3 pobles amb els quals executar l'algorisme. D'aquesta forma, executarà el mètode `trobarPesos()` de la classe Dijkstra i modificarà les dades del model mitjançant aquest.

- `notificar()`: mètode que simplement inicialitza l'atribut `msg` amb l'string que rep per paràmetre i posa en marxa el mètode `run()`.

7 Programa Principal

En aquesta secció s'explicarà la estructura que segueix el nostre programa principal, a més de les accions a realitzar. El nostre programa principal conté com a atributs tres diferents instàncies que implementen el MVC:

```
private Model mod;
private Vista vis;
private Control con;
```

Amb aquests punters podem accedir a l'informació del model i notificar el diferents events enregistrats. En quant als métodes, el més important i l'únic digne d'explicació es el de `notificar(String s)` Aquest métode implementa les accions que deriven de l'String rebut per paràmetre, cada posibilitat del switch que utilitzam representa un dels botons de l'interfície les diferentes opcions són:

- `case 'Llegir fitxer': `: aquest botó s'encarrega de carregar el fitxer i mostrar el graf pertinent. per fer això feim servir una instància de la clase `sax`, aquesta classe li pasam per paràmetre el fitxer i cridam al métode llegir(), vist a classe. Una vegada fet això notificam a la vista que ja hem llegit el fitxer.

- `case 'Borra': `, aqui reseteam el model i notificam a la vista per a que borri tot el que havia pintat.

- `case 'Calcula': `, en aquest botó notificam al control per a que calculi amb el parametre indicat.

8 Joc de proves

En aquesta secció ens encarregarem de realitzar 2 jocs de proves diferents, per a comprovar el correcte funcionament de l'aplicació. Aquests testeigs consistiran en escollir 3 diferents pobles per a analitzar les possibles rutes i veure si és la mínima. Cal remarcar, com hem dit anteriorment, que si es passa per un poble més d'un cop, apareix l'ordre dels nombres concatenat amb el caràcter ',' coma.

8.1 Primer cas de prova

En aquest cas de prova, calcularem la ruta entre aquests 3 pobles:

$$ruta = \{x_0 \rightarrow x_1 \rightarrow x_2\}$$

$$x_0 \leftarrow \text{Cala Llenya}$$

$$x_1 \leftarrow \text{St Antoni Abad}$$

$$x_2 \leftarrow \text{La Savina}$$

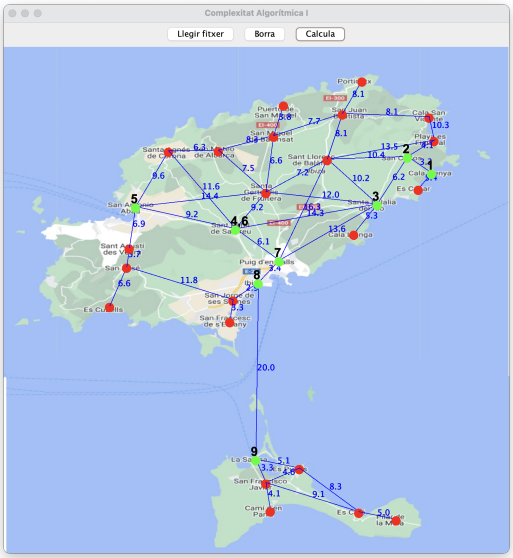

Figura 3. Recorregut del primer cas de prova

Com es pot apreciar en la imatge, seguim aquesta ruta:

Cala Llenya → Sant Carlos → Santa Eulàlia del Riu → Santa Rafel de Sa Creu → Sant Antoni Abad → Sant Rafel de Sa Creu → Puig d'en Valls → Ibiza → La Savina

El cost de la ruta és el següent:

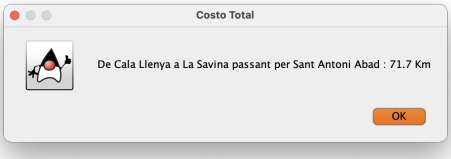

Figura 4. Cost del primer cas de prova

8.2 Segon cas de prova

En aquest cas de prova, calcularem la ruta entre aquests 3 pobles:

$$ruta = \{x_0 \rightarrow x_1 \rightarrow x_2\}$$

$x_0 \leftarrow$ Santa Gertrudis de Fruitera

$x_1 \leftarrow$ St Francesc Xavier

$x_2 \leftarrow$ Pilar de la Mola

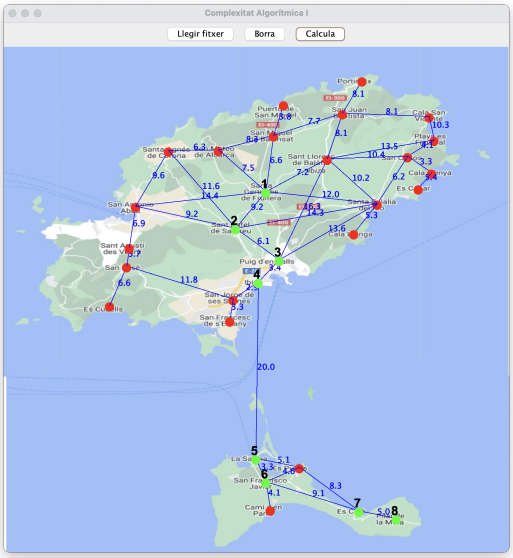

Figura 5. Recorregut del segon cas de prova

Com es pot apreciar en la imatge, seguim aquesta ruta:

Santa Gertrudis de Fruitera → Sant Rafel de Sa Creu → Puig d'en Valls → Ibiza → La Savina → Sant Francesc Xavier → Puig d'en Valls → Ibiza → La Savina → Sant Francesc Xavier → Es Caló → Pilar de la Mola

El cost de la ruta és el següent:

Figura 6. Cost del segon cas de prova

8.3 Tercer cas de prova

En aquest cas de prova, calcularem la ruta entre aquests 3 pobles:

$$ruta = \{x_0 \rightarrow x_1 \rightarrow x_2\}$$

$x_0 \leftarrow$ Santa Agnés de Corona

$x_1 \leftarrow$ San José

$x_2 \leftarrow$ Camí den Parra

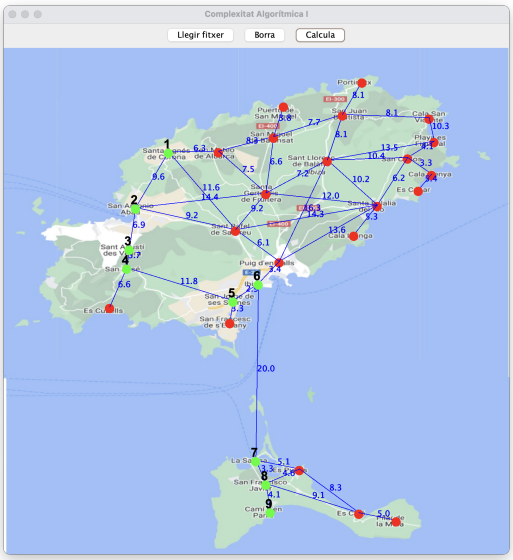

Figura 7. Recorregut del cost cas de prova

Com es pot apreciar en la imatge, seguim aquesta ruta:

Santa Agnés de Corona → Sant Antoni Abad → San Agustí des Vedrà → San José → San Jordi de ses Salines → Ibiza → La Savina → Camí den Parra

El cost de la ruta és el següent:

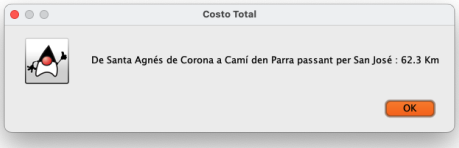

Figura 8. Cost del tercer cas de prova

9 Conclusions Finals

Aquesta pràctica, tot i no haver comparació de rendiments entre algoritmes, podem extreure unes conclusions finals molt interessants.

En primer lloc, cal destacar que aquesta pràctica compleix el següent principi:

$$\exists S^* \quad | \quad S^* \in opt$$

Existeix un conjunt que és òptim, S^*, que pertany al grup opt, representant la optimalitat.

$$S^* = \{S_0, S_1, ...S_n | \sum_{i=0}^{n} S_i = S^*\}$$

A continuació, podem observar que aquest conjunt és pot dividir en subconjunts més petits S_i, de tal manera que afegits tots, formin un altre cop S^*. Això vendria a ser com dividir la solució en conjunts més petits.

$$\forall S_i \in S^* \rightarrow S_i \in opt$$

Per a qualsevol subconjunt que pertanyi a S^*, és demostrat que aquest subconjunt també és òptim. Hem fet la demostració general, però anem a veure el problema des de una perspectiva més enfocada al nostre problema:

$$S^* = \{\mathbf{A} \rightarrow \mathbf{B} \rightarrow \mathbf{C}\}$$

Suposam que tenim la solució òptima de la ruta que va de **A** a **C**, passant per **B**. Destacam que aquestes tres lletres poden esser 3 pobles qualssevol, no afecta en el principi postulat anteriorment.

$$S^* = \{S_0 \cdot S_1 \quad | \quad S_0 = \{\mathbf{A} \rightarrow \mathbf{B}\} \wedge S_1 = \{\mathbf{B} \rightarrow \mathbf{C}\}\}$$

Com podem apreciar, es pot dividir el problema en anar de **A** fins a **B** i després de **B** fins a **C**. D'aquesta manera, ambdós subconjunts de la solució són òptims, és a dir, que la solució ve donada per la concatenació dels camins més curts anteriors.

Es pot això complir per a qualsevol ruta que hagui de passar per n pobles? Sí i no. Si suposam que enlloc de tenir una ruta amb 3 pobles en tenim entre 4, existeixen dues possibilitats:

$$\exists \mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$$

$$S_0^* = \{\mathbf{A} \rightarrow \mathbf{B} \rightarrow \mathbf{C} \rightarrow \mathbf{D}\}$$

$$S_1^* = \{\mathbf{A} \rightarrow \mathbf{C} \rightarrow \mathbf{B} \rightarrow \mathbf{D}\}$$

Això ens fa pensar en un factor molt important, l'ordre a seguir. El fet de posar aquesta restricció ens fa disminuir el nombre de possibilitats de manera dràstica. Si tenim n pobles i volem fer una ruta del primer al darrer seguint un ordre establert amb els intermigs, podem aplicar el principi anterior. Si és al contrari, ens trobam amb un problema molt més difícil computacionalment. El problema passar a ser del viatjant de comerç.

En aquest cas, quan ens trobam amb el problema del viatjant de comerç, ens estam enfrontant a un problema que pertany a "NP-Hard", que ens diu que no es pot resoldre amb un temps polinomial, sinó que és exponencial. Tot i haver implementacions que arriben a trobar solucions molt pròximes a la òptima, no es garanteix el fet de trobar la solució òptima en temps polinomial. Si no és troba la solució òptima, no garanteix que qualsevol conjunt més petit sigui òptim també.

En conclusió, podem extreure dos idees molt importants:

- **Divisió de solucions òptimes**: el nostre estudi ens ha demostrat que donat qualsevol conjunt S^* amb la solució òptima compleix aquest principi:

  $$\forall S_i \in S^* \rightarrow S_i \in opt$$

  Per tant, si hem trobat el camí més curt de Palma cap a Sóller passant per Inca,per posar un exemple, haurem trobat dos camins més curts. El primer és el de Palma cap a Inca i l'altre el de Inca cap a Sóller, evitant així possibles càlculs en un futur.

- **L'ordre és un factor molt important**: per a qualsevol nombre $n > 3$ que tenguem, s'haurà de establir una ruta si volem tenir un problema que es pugui resoldre en temps polinòmic. Remarcam que això comença a passar quan $n = 4$.

10 Bibliografia

- GeeksForGeeks
- Video Explicatiu de la Pràctica amb execucions