Trabajo Práctico Nº2

Programación III

**Integrantes**

* Luraschi, Roberto Gino

Contenido

[Contenido 2](#_Toc514159476)

[Decisiones tomadas 3](#_Toc514159477)

[Decisiones tomadas a nivel de código de controladores 3](#_Toc514159478)

[Decisiones tomadas en cuanto a algoritmo de Dijkstra 3](#_Toc514159479)

[Código de las clases 4](#_Toc514159480)

# Decisiones tomadas

A lo largo del desarrollo del trabajo práctico, se han tomado distintas decisiones, tanto a nivel de código funcional como del código visual.

A continuación se detallarán las decisiones tomadas en cada caso.

## Decisiones tomadas a nivel de código de controladores

En cuanto a nivel general de la aplicación se decidió tomar un diseño de MVC.

En el que el modelo serán los puntos de interés, estos puntos de interés tienen sus IDs para poder ser representados en el grafo fácilmente(se opto por esto ya que es mas fácil su interpretación y funcionamiento).

La vista se utilizará la consola(fácilmente se puede mudar a un formulario hecho con Java Swing). La vista esta hecha en 2 partes: Ingreso de puntos de interés e ingreso de túneles, por el simple hecho de que, a la hora de ingresar los túneles(que son aristas del grafo) se necesita inicializar el grafo, que no esta inicializado por el hecho de no saber cuantas minas puede llegar a tener.

La parte compleja se encuentra en el controlador, ya que este tiene los métodos para tomar los puntos de interés con sus respectivos tiempos y sus conexiones. Tiene métodos para poder agregar por nombre las minas, también los tiempos entre puntos de interés y un método encargado de devolver el listados de las minas.

## Decisiones tomadas en cuanto a algoritmo de Dijkstra

Para diseñar el algoritmo de búsqueda se hizo una investigación en cuanto al funcionamiento, quedando como resultado un algoritmo que consta de la creación de un vector con las distancias a cada uno de los vértices desde el que se comienza (el cual es elegido por el usuario) en el que la distancia será 0 y en los restantes un valor infinito.

Además se creará un lista con los vértices visitados, y se agregará el vértice inicial. Al comenzar a recorrer el grafo se hará una búsqueda que mientras no sean visitados todos los vértices, se buscara desde los vértices visitados, el que menor distancia tenga desde el vértice inicial.

Al tener el vértice con menor distancia se va a marcar y se va a tomar desde el comienzo se va a tomar su arista desde el camino visitado hasta el vértice con menor distancia. Finalmente se va a sumar la distancia al vector con las distancias y se agregará la arista al árbol resultado.

# Código de las clases

**Algoritmo de Controlador.java**

**public** **class** Controlador {

ArbolConTiempo arbol;

**private** List<PuntoInteres> listaMinas;

**public** String CAMINOS\_SIN\_CARBON = "No existen caminos con carbon";

**public** Controlador() {

listaMinas = **new** ArrayList<PuntoInteres>();

}

**public** **void** agregarArista(PuntoInteres minaA, PuntoInteres minaB, **int** tiempo) {

**if** (tiempo < 0)

**throw** **new** RuntimeException("El tiempo no puede ser negativo.");

**if** (arbol == **null**)// Aca ya que no se sabe cuantas minas se ingresaran

arbol = **new** ArbolConTiempo(listaMinas.size());

arbol.agregarArista(minaA.getId(), minaB.getId(), tiempo);

}

**public** **void** agregarMina(**int** id, String nombre, **double** cantCarbon)

**throws** Throwable {

**if** (cantCarbon < 0)

**throw** **new** RuntimeException("No se puede agregar Carbon Negativo");

PuntoInteres pi = **new** PuntoInteres(id, nombre, cantCarbon);

listaMinas.add(pi);

}

**public** PuntoInteres getMina(String nombre) {

**for** (**int** x = 0; x < listaMinas.size(); x++)

**if** (listaMinas.get(x).getNombre().equals(nombre))

**return** listaMinas.get(x);

**return** **null**;

}

**public** String getListaMinas() {

StringBuilder sb = **new** StringBuilder();

**for** (PuntoInteres pi : listaMinas)

sb.append(pi.getId() + " - " + pi.getNombre() + "\n");

**return** sb.toString();

}

**public** **void** agregarArista(String minaA, String minaB, **int** tiempo)

**throws** Throwable {

PuntoInteres puntoInteresA = getMina(minaA);

**if** (puntoInteresA == **null**)

**throw** **new** RuntimeException("La mina " + minaA + " no existe.");

PuntoInteres puntoInteresB = getMina(minaB);

**if** (puntoInteresB == **null**)

**throw** **new** RuntimeException("La mina " + minaB + " no existe.");

**if** (tiempo < 0)

**throw** **new** RuntimeException("El tiempo ingresado es incorrecto");

agregarArista(puntoInteresA, puntoInteresB, tiempo);

}

StringBuilder sb;

**public** String recorrer(**int** ini) {

ArbolConTiempo agm = Dijkstra.*resolver*(arbol, ini);

String ret = "";

**double** mejorRecorrido = Double.***MAX\_VALUE***;

**for** (**int** x = 0; x < agm.getCantidadVertices(); x++){

sb = **new** StringBuilder();

sb.append(ini).append("--").append(agm.peso(ini, x))

.append("-->").append(x);;

**if** (agm.existeArista(ini, x)) {

**double** p = buscarMejorRecorrido(agm, ini, x);

**if** (p < mejorRecorrido) {

mejorRecorrido = p;

ret = sb.toString();

}

}

}

**return** ret.equals("") ? CAMINOS\_SIN\_CARBON : ret;

}

**private** **double** buscarMejorRecorrido(ArbolConTiempo agm, **int** ini, **int** x) {

**int** actual = x;

**if** (listaMinas.get(x).getCantCarbon() > 0)

**return** agm.peso(ini, x);

**for** (**int** y = 0; y < agm.getCantidadVertices(); y++) {

**if** (agm.existeArista(actual, y)) {

sb.append("--").append(agm.peso(actual, y))

.append("-->").append(y);

**if** (listaMinas.get(y).getCantCarbon() > 0)

**return** agm.peso(actual, y);

actual = y;

}

}

**return** Double.***MAX\_VALUE***;

}

}

Algoritmo de Dijkstra

**public** **class** Dijkstra {

**public** **static** ArbolConTiempo resolver(ArbolConTiempo grafo, **int** comienzo) {

**int** n = grafo.vertices();

ArbolConTiempo agmRet = **new** ArbolConTiempo(n);

Double[] distancias = **new** Double[n];

List<Integer> visitados = **new** ArrayList<Integer>();

distancias[comienzo] = (**double**) 0;

visitados.add(comienzo);

**while** (visitados.size() < grafo.getCantidadVertices()) {

Arista aristaMinima = *obtenerMinimoVisitado*(visitados, distancias, grafo);

visitados.add(aristaMinima.y);

distancias[aristaMinima.y] = aristaMinima.peso;

agmRet.agregarArista(aristaMinima.x, aristaMinima.y, aristaMinima.peso);

}

**return** agmRet;

}

**private** **static** Arista obtenerMinimoVisitado(List<Integer> visitados, Double[] distancias, ArbolConTiempo grafo) {

**double** min = Double.***MAX\_VALUE***;

Arista ret = **new** Arista();

**for** (**int** x : visitados) {

**for** (**int** y = 0; y < grafo.vertices(); y++)

**if** (!visitados.contains(y) && grafo.existeArista(x, y))

**if** (distancias[x] + grafo.peso(x, y) < min) {

min = distancias[x] + grafo.peso(x, y);

ret.x = x;

ret.y = y;

ret.peso = min;

}

}

**return** ret;

}

**static** **class** Arista {

**int** x;

**int** y;

**double** peso;

}