****

**Universidad Nacional de General Sarmiento**

**Programación III**

**Trabajo Práctico:**

**Clustering con AGM**

**Profesores:**

* Patricia Bagnes.
* Javier Marenco.

**Grupo:**

* Andrés Geiser
  + 2018/42239150
  + geiserandres00@gmail.com
* Hernán Rodrigo Fernández
  + 2017/41584220
  + fernandezhr\_1998@hotmail.com

**Objetivo**

El objetivo que nos planteamos resolver en este trabajo practico consiste en implementar una aplicación que muestre la formación de distintos clusters a partir de un conjunto de coordenadas en el mapa, separando el código responsable de la lógica del programa de aquel que se encarga de modelar la interfaz (implementada con Swing y JMapMaker).

**Responsabilidad de clases**

El proyecto cuenta con 5 paquetes en los que están distribuidas las clases según su funcionalidad:

* **Controlador:** dentro se encuentran las clases encargadas de comunicar la interfaz gráfica con el modelo, implementando el ActionListener de los distintos componentes de la interfaz visual.
* **Modelo:** dentro se encuentran las clases encargadas de poner en funcionamiento la lógica interna del programa.
* **Vista:** en él se encuentran las clases visuales, que contienen el código de interfaz correspondiente al MainFrame y los distintos paneles.
* **Test:** dentro se encuentra la suite de test correspondiente al funcionamiento interno.
* **Iconos:** contiene imágenes que son utilizadas por las clases visuales como iconos para los botones.
* **Principal:** contiene la clase principal que se encarga de inicializar las clases visuales y desencadenar el funcionamiento del programa.

A continuación, se prestará detalle cada una de las clases de los paquetes más importantes, describiendo sus variables de clase, sus métodos y su rol dentro del programa.

PAQUETE

**MODELO**

**Clase Grafo**

**Variables de clase**

* *double[][] D*: representa la matriz de distancias interna del grafo.

**Métodos de clase**

A continuación, veremos la descripción de los métodos más importantes de esta clase:

* *void agregarArista(int i, int j, double distancia)* : se encarga de agregar una arista entre los vértices **i** y **j** colocando en esa posición de la matriz la distancia entre ambos.
* *void borrarArista(int i, int j):* se encarga de eliminar la arista entre los vértices **i** y **j** volviendo nula la distancia entre ellos.
* *boolean existeArista(int i, int j):* verifica si existe una arista entre los vértices **i** y **j** es decir si la posición de la matriz es mayor a cero.
* *set<Integer> vecinos(int i)*: retorna una colección con aquellos vértices que tengan una arista con el vértice **i**.

**Clase AGM**

Esta clase implementa una versión del famoso algoritmo de Prim para obtener un AGM.

**Variables de clase**

* *Grafo grafoAGM:* variable que contendrá el árbol generador mínimo del grafo pasado como parámetro en el constructor.
* *boolean marcados[]:* arreglo que usaremos para ver que vértices ya han sido agregado en el *grafoAGM* para luego buscar la arista mínima entre todos ellos.
* *ArrayList<Arista> aristas:* contendrá las aristas finales del árbol generador mínimo.

**Métodos de clase**

* *AGM (Grafo grafo):* constructor que recibe como parámetro un grafo con el cual se trabajara para armar su árbol generador mínimo.
* *void verificarTamanio():* comprueba que el grafo no este vacío, es decir que contenga al menos un vértice.
* *boolean todosMarcados():* chequea si todos los vértices están marcados es decir si fueron agregados al árbol generador mínimo.
* *void generarAGM (Grafo grafo):* método que construye el árbol generador mínimo mediante una versión del algoritmo de Prim. Lo que hace su algoritmo es lo siguiente:

Supongamos que tenemos este grafo, lo que hacemos en primer lugar es marcar un vértice cualquiera, nosotros marcamos el vértice 0. Luego por cada vértice marcado recorremos sus vecinos para encontrar la arista con menor peso entre todas.

Grafo

10

12

9

2

30

7

15

En este caso la arista con menos peso fue la del vértice 0 y el 1. Agregamos esa arista al *grafoAGM* y marcamos en el arreglo el vértice 1. Ahora repitiendo el mismo proceso buscara la arista con menor peso entre los vértices marcados.

Grafo

10

12

9

2

30

7

15

Se agrega la arista (1,5) al grafoAGM porque fue la menor y marcamos el vértice 5 en el arreglo.

Este proceso se repite hasta tener todos los vértices marcados.

Grafo

10

12

9

2

30

7

15

Finalmente, el grafoAGM quedaría conformado por estas aristas.

Grafo

12

9

2

7

15

Aristas de las cuales se elegirá la menor para agregar al grafoAGM.

Vértices ya marcados y aristas agregadas al grafoAGM.

Vértices no marcados.

**Clase Arista**

**Variables de clase**

* *int vertice1, vertice2:* representan los vértices que se unen mediante una arista.
* *double peso:* representa el peso de la arista entre ambos vértices.

Los métodos de esta clase no son relevantes ya que devuelven información de la Arista en sí misma.

**Clase Modelo**

Esta clase es la más importante de este paquete, utiliza todas las clases mencionadas anteriormente para llevar a cabo la lógica de la aplicación.

**Variables de clase**

* *ArrayList<Coordinate> coordenadas*: conjunto de coordenadas con las cuales se armará el grafo.
* *Grafo grafo*: grafo conformado por vértices que representaran cada una de las coordenadas del conjunto y sobre el cual se aplicará el clustering.
* *ArrayList<Arista> aristasGrafoOriginal:* conjunto de aristas del grafo que usaremos para no perder las aristas del grafo original después de hacer clustering.
* *double pesoTotal, desviacionEstandar;*

*int cantClusters:* variables que solo guardan datos estadísticos del grafo.

**Métodos de clase**

* *agregarCoordenada(Coordiante coordenada):* método que recibe una coordenada y la agrega al conjunto.
* *armarGrafo():* método que pone en marcha la creación del grafo utilizando las siguientes funciones:
  + *armarGrafoCompleto():* arma el grafo uniendo cada uno de los vértices con los demás, calculando la distancia euclídea entre las coordenadas que representara el peso de la arista entre los vértices.
  + *grafoArbolMinimo():* hace uso de la clase AGM para obtener el árbol generador mínimo del grafo completo creado en la función anterior.
* *clustering(int n):* función que se encarga de borrar

algunas aristas del grafo para obtener una cantidad de clusters igual al número **‘n’** pasado como parámetro. Para hacerlo hace uso de las funciones:

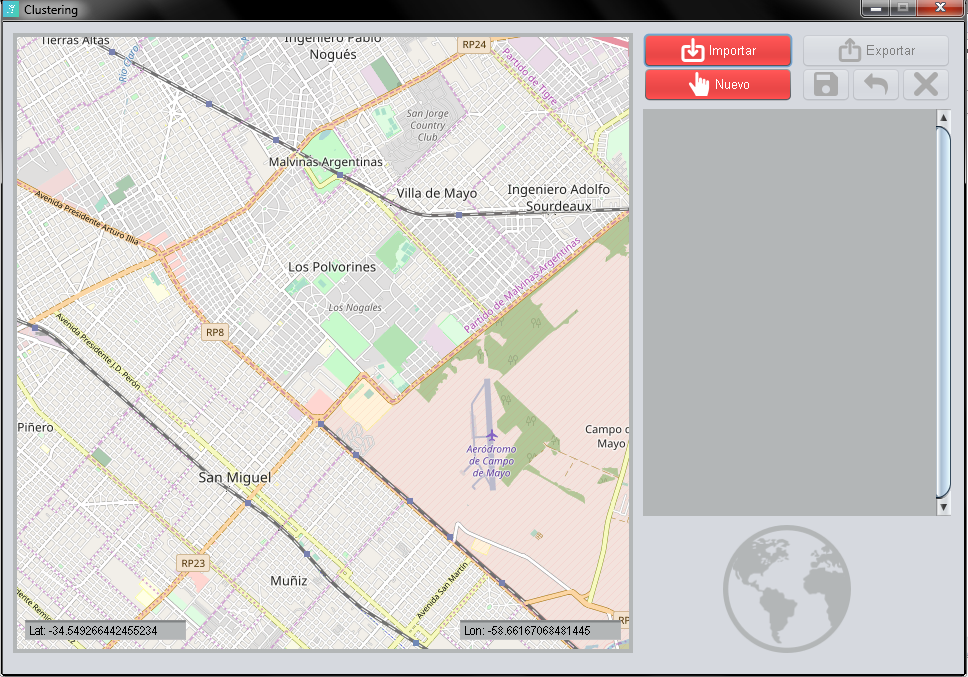
* *armarGrafoOriginal():* resetea el grafo para obtener el grafo creado originalmente en la función **armarGrafo()** para poder hacer clustering más de una vez.
* *eliminarAristasMayores(int n):* elimina la cantidad **‘n’** de aristas con mayor peso del grafo.
* *actualizarDatos(ArrayList<Arista> aristas, int n):* método que modifica los datos estadísticos del grafo y que se lo invoca al momento de armar el grafo por primera vez y después de hacer clustering.

PAQUETE

**VISTA**

**Clase VentanaPrincipal**

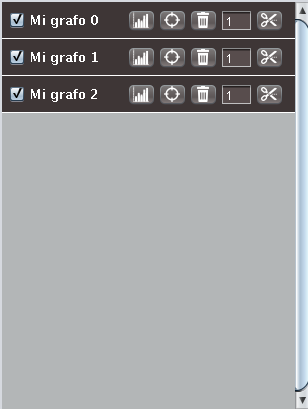
Esta clase es la que implementa la interfaz principal.



Cuenta con una serie de seis botones, un panel a la derecha con un mapa y dos textos que marcan la longitud y latitud de la posición del puntero en el mapa, un panel a la izquierda (panel implementado por nosotros en una clase que se explicara más adelante) en el que se irán colocando otros paneles más pequeños (también implementado por nosotros en otra clase) y una imagen en la parte inferior izquierda.

**Clase PanelDeControles**

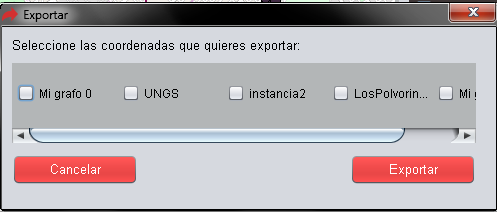
Clase que hereda de JPanel y que representa al panel izquierdo de la ventana principal. Fue creada ya que no pudimos encontrar un layaout que coloque los paneles que controlan los grafos uno debajo de otro como en la siguiente imagen:



Lo logramos usando el Absolute Layaout y creando el método agregar, el cual manipula una variable que al momento de agregar un panel lo posicione en el valor que indica esa variable y aumente su valor para que el próximo lo coloque debajo. También tiene el método eliminar para que los paneles se acomoden automáticamente cuando se eliminan.

**Clase VentanaExportar**

Clase que implementa la ventana que se abre al presionar el botón Exportar de la ventana principal:



Tiene 2 botones y un panel central que contiene varios JCheckBox con los nombres de los grafos.

**Clase PanelGrafo**

Esta es una clase que hereda de JPanel y es un panel que se ira agregando en el panel izquierdo de la ventana principal a medida que se vayan creando distintos grafos en el mapa. Tiene el siguiente aspecto:



Es un panel pequeño y contiene un JCheckBox, 4 botones y un texto, con los cuales podremos interactuar con el grafo que le fue asignado.

PAQUETE

**CONTROLADOR**

Dentro de este paquete se encuentran aquellas clases que se encargan de distribuir los datos obtenidos desde la interfaz hacia las clases que implementan la lógica interna, así como también controlar los distintos componentes que referencian al armado y configuración del armado general del grafo. Se destacarán aquellas variables y métodos que no resulten triviales de explicar

**Clase CtrlPanelGrafo**

**Variables de clase**

* *ArrayList<MapMarkerDot> puntos* contiene todos los puntos que corresponden a cada coordenada ubicada en el mapa.
* *ArrayList<MapPolygonImpl> aristas* contiene a todas las uniones entre las diferentes coordenadas marcadas en el mapa.
* *Color colorGrafo* representa el color que se le asignara a cada esquema de puntos sobre el mapa, que se formara de manera aleatoria.

**Métodos de clase**

Los siguientes métodos son los que representan el funcionamiento elemental de la clase, ya que los demás desarrollan una funcionalidad auxiliar:

* *void iniciar():* arma el esquema y dibuja en el mapa los puntos y las aristas entre ellos.
* *void eliminar():* elimina los puntos y las aristas del mapa.
* *void clustering():* toma la cantidad de clusters a representar y, si la cantidad resulta válida, vuelve a dibujar las aristas con la nueva configuración.

Luego, es importante mencionar a algunos métodos tales como:

* *void dibujarAristas():* dibuja en sobre el mapa una arista entre los puntos correspondientes. Para esto se toma a cada punto y se lo une con su vecino y este quede unido a si mismo ya que cada MapPolygon requiere tres referencias.
* *void verEstadisticas():* muestra un mensaje que arroja información sobre el estado actual del esquema mostrado en pantalla.

**Clase CtrlVentanaPrincipal**

**Variables de clase**

* *ArrayList<MapMarkerDot> marcasTemporales* contiene los puntos que están representados en el mapa en el estado actual.
* *ArrayList<CtrlPanelGrafo> ctrlPanelesGrafos* contiene los respectivos controladores para cada grafo representado en el mapa

**Métodos de clase**

* *void iniInteraccionConMapa():* controla la interacción del usuario con el mapa, es decir agrega los puntos que elija manualmente.
* *void buscarArchivos():* lanza un buscador de archivos del que se obtendrá el o los archivos con las instancias.
* *void leerArchivos(File[] archivos):* para cada archivo recibido se toma cada coordenada correspondiente a la latitud y la longitud y se agrega al grafo que es representado por la clase Modelo.
* *void exportar():* lanza una nueva VentanaExportar que se encargara de almacenar la instancia elegida en un archivo.

Luego, es importante mencionar a algunos métodos tales como:

* *void colocarPanelGrafo(String nombre)*: que recibe el nombre de la nueva instancia y crea su respectivo CtrlPanelGrafo y su instancia de modelo.
* *Void borrarMarcasTemporales();void borrarUltimaMarca()*

que eliminan del mapa la ultima marca agregada o bien todas aquellas.

**Clase CtrlVentanaExportar**

**Variables de clase**

* *VentanaExportar vExportar* representa la clase que implementa la interfaz visual.
* *ArrayList<CtrlPanelGrafo> controladoresGrafos* donde se almacena cada controlador del grafo correspondiente.

**Métodos de clase**

* *void elegirDirectorio():* lanza un nuevo JFileChooser del que se obtendrá el directorio donde se almacenara el archivo.
* *void crearArchivos(File directorio):* que recibe el directorio elegido y crea un nuevo archivo “.txt” en esa ruta con las coordenadas de la instancia elegida.
* *boolean existe(File directorio):* que retorna verdadero si el archivo existe en esa ruta.
* *ArrayList<CtrlPanelGrafo> panelesElegidos():* retorna el conjunto de los paneles de las instancias elegidas.

**Problemas encontrados y soluciones**

Durante el desarrollo de la aplicación surgieron algunos inconvenientes que tuvimos que afrontar:

* Uno de los principales problemas que afrontamos fue el de decidirnos por un algoritmo que se encargue de procesar el grafo y devuelva un AGM, para esto tuvimos en cuenta dos algoritmos famosos, Kruskal y Prim. Luego identificamos que Prim era el que se adaptaba de mejor manera a nuestro problema, de esa manera lo implementamos junto con su respectiva suite de test.
* Por otro lado, resultaba engorroso o poco eficiente importar las instancias una a la vez, por lo que modificamos el código de JFileChooser para que permita recibir más de un archivo a la vez y que también se muestren prioritariamente los “.txt” que son el tipo de archivos con los que trabaja nuestro programa.
* En último lugar, la cantidad de clases y código abarco un espacio más grande que los proyectos que realizamos hasta el momento, de ahí viene la idea de separar cada componente visual de su controlador, es decir tener distintos paquetes en los que solo se encuentren clases visuales y otros que contengan clases que se ocupen de su funcionalidad interna, lo que brinda claridad y estructura propia a cada clase.