Universidad Nacional de General Sarmiento
Instituto de Industria

Trabajo practico n°2

Programación III

Comisión: 2.

Docentes: Alejandro Nelis, Fernando Torres.

Alumnos: Chamorro Franco, Montenegro Tomas, Romano Facundo, Sandoval Maximiliano

El siguiente informe explica la composición y organización de cómo se realizó el segundo trabajo practico de la materia Programación III, en el cual implementamos diferentes y nuevos elementos que tuvimos que aprender a utilizar y contemplar para poder cumplir con los requerimientos pedidos por parte del enunciado de dicho trabajo practico. Algunos de ellos son

JMapViewer, que es una librería y/o herramienta que nos permite usar mapas y la implementación de JSON que sirve principalmente para el intercambio de datos.

A diferencia del trabajo practico anterior, comenzamos trabajando mucho más organizados en cuento al modelado del mismo, lo cual nos permitió obtener una perspectiva mucho más clara de los objetivos a realizar.

Comenzamos fraccionando el proyecto en 3 diferentes packages, que son “Interfaz”, “Negocio” y “Test”. Esto nos permitió desde un principio ser más organizados a la hora de realizar el diagrama de clases, la cual suministramos de forma opulenta.

En el package “Interfaz” creamos las clases “Principal_TP”, “MenuPrincipal” y “VistaMapa” que contiene todo el código implementado para brindar la mecánica que dispone Windowsbuilder.

La clase “MenuPrincipal” nos dará los elementos necesarios para ejecutar la pantalla que se inicia cuando se ejecuta el programa, que pedirá el precio por metro de lo que costaría la red. Sobre el textField de ingresar precios que capturara el input del usuario (cuanto es el costo por metro de la red). Tenemos el método “inicializarBotonEnviar” que si es presionado captura lo que ingreso el usuario (costo), limitandolo a que ingrese un numero para poder continuar:

```
private void inicializarBotonEnviar() {
          btnEnviar = new JButton("ENVIAR");
          btnEnviar.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent e) {
               if
(textFieldIngresarPrecio.getText().length()>0) {


precioPormetro=Double.parseDouble(textFieldIngresarPrecio.getText
() ); cambiarDeVentana();
                    }
                    else
                         System.out.println("INGRESE UN PRECIO
PARA CONTINUAR");
               }
          });
          btnEnviar.setBounds(355, 470, 105, 30);
          menuPP.getContentPane().add(btnEnviar);
     }
```

Tenemos también “limitarInputUsuario” que verifica que el usuario ingrese números en el precio, limitando la longitud de números a 7, para que no se pueda poner un valor que exceda el límite y eso pueda generar un error:

```
private void inicializarBotonEnviar() {
          btnEnviar = new JButton("ENVIAR");
          btnEnviar.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent e) {
               if
(textFieldIngresarPrecio.getText().length()>0) {


precioPormetro=Double.parseDouble(textFieldIngresarPrecio.getText
() ); cambiarDeVentana();
                    }
                    else
                         System.out.println("INGRESE UN PRECIO
PARA CONTINUAR");
               }
          });
          btnEnviar.setBounds(355, 470, 105, 30);
          menuPP.getContentPane().add(btnEnviar);
     }
```

Una vez que presionemos el botón enviar y cumple las condiciones establecidas, cambiamos de ventana del mapa.

En la clase “VistaMapa” agregamos los métodos necesarios para poder implementar el JMapViewer. La creación de diversos paneles propios del mapa (JMapViewer), los botones y labels, con el inicio de los mismos (ubicaciones en la interfaz, tamaño, etc.) y acciones necesarias para poder trabajar con los grafos en la interfaz.

Por ejemplo, tenemos del método “detectarCoordenadasClick”:

```
private void detectarCoordenadasClick() {
mapa.addMouseListener(new MouseAdapter() {
               @Override
               public void mouseClicked(MouseEvent e) { if
                    (e.getButton() == MouseEvent.BUTTON1) {
                         Coordinate marcadorClick = (Coordinate)
mapa.getPosition(e.getPoint());
```

```
                        String nombre =
JOptionPane.showInputDialog("NOMBRE: ");
                        mapa.addMapMarker(new
MapMarkerDot(nombre, marcadorClick)); logica.altaLugar(nombre,
                        marcadorClick);
                    }
                }
            });
    }
```

Que nos permitirá detectar una coordenada sobre el mapa, clickeando con el mouse en alguna parte de él y dicha coordenada se guardará con un nombre y su respectiva posición.

El package “Negocio” creamos la clase “Lógica”, “Grafo”, “Arista”, “Lugar”, “LugaresJSON” y “ArbolPrim”, que contienen el código que le otorga la funcionalidad al proyecto.

En la clase “ArbolPrim” utilizamos el algoritmo de Prim, ya que, si bien el algoritmo de Kruskal es más eficiente, el algoritmo de Prim nos pareció más óptimo en cuanto a nuestra implementación y mucho más claro/transparente, al momento de llevarlo a la práctica, cumpliendo finalmente el objetivo buscado. Antes de pasar a explicar el mero funcionamiento de estos métodos, nos pareció importante recalcar la lógica y el racionamiento que llevamos a cabo para poder aplicar Prim. En primer lugar, el proyecto se adapta perfectamente a las propiedades de Prim, entre sus condiciones el hecho de que estamos trabajando sobre un grafo conexo (existe al menos una arista que conecta a los vértices), en dicho grafo, cada arista posee un peso (distancia), y tiene la cualidad de no ser dirigido. En cuanto a la implementación teoría, nosotros indexamos o accedemos a la primera arista, para luego avanzar e ir seleccionando las aristas siguientes, con menor peso, comprobando siempre que no se generen ciclos, para evitar romper el AGM (ya que perdería su propiedad principal).

Los métodos dentro de la clase, son complementarios porque los mismos se llaman entre sí:

```
private void resolverArbolConPrim(List<Arista> ariClonadas) {
        if (!ariClonadas.isEmpty()) {
            Lugar l = ariClonadas.get(0).lugarA;
            lugaresRecorridos.add(l);
        }

        while (!ariClonadas.isEmpty()) {
            Arista ari =
pesoMinimoSinCiclos(lugaresRecorridos);
```

```
                if (ari != null) {
                    retAristas.add(ari);

                    if
                    (lugaresRecorridos.contains(ari.lugarA))
                    lugaresRecorridos.add(ari.lugarB); else
                    lugaresRecorridos.add(ari.lugarA);
                } else ariClonadas.clear(); // LIMPIO
                    ARISTAS
INUTILIZABLES
            }
        }
```

El método anterior llamara al método “pesoMininoSinCiclos” de la misma clase e ira almacenando, mediante un ciclo, una nueva arista a la cual le pasa la arista que se encuentra más cerca a la arista actual (la de menor peso) que no genera ciclos (ya que si lo generara se rompería la propiedad de un AGM), luego va a devolver “null” cuando ya solo queden aristas que generen ciclos e ira registrando los lugares (aristas) por los que ya paso.

Sobre la clase “Arista” en un principio habíamos sobrescrito el “equals” que podemos generar por defecto en java, por una cuestión de que funcionara correctamente el clone y otros métodos que estábamos empleamos, ya que, al ser un objeto, quedarían algunas propiedades en “null”. Luego realizamos una nueva implementación (mas optima) en la cual no es necesario hacer uso de un Override del método “equals”.

Para la nueva implementación, tenemos el Override del objeto clone (objeto heredado de la clase Object de java) que lo sobrescribimos por una cuestión de herencia que luego utilizaremos en la clase “ArbolPrim”, justamente cuando lo inicializamos en el constructor, cuyo clone lo estará heredando de la clase “Arista”. Esto es necesario hacerlo para que no ocurra aliasing, sobrescribiendo el método original de java contemplando los datos que necesitamos para que se pueda clonar bien el objeto y no queden secciones o variables en “null”.

```
@Override
    protected Object clone() throws CloneNotSupportedException {
final Arista newObj = new Arista(lugarA,lugarB);
        newObj.setDistancia(this.distancia);
        return newObj;
    }
```

Sobre la clase “Grafo” para representar adecuadamente los grafos, implementamos la forma “listas de vecinos” (la idea es almacenar la relación y conexión que tiene un vértice con el otro), en la que implementamos dos listas que almacenaran, por un lado, aristas y por otro lado los lugares.

```
	private List<Arista> listaAristas;//caminos
	public List<Lugar> listaLugares;

	public Grafo() { listaLugares = new
		LinkedList<Lugar>(); listaAristas = new
		LinkedList<Arista>();
}
```

Dentro de la implementación de la clase, brindamos los métodos para poder trabajar sobre estas listas, por ejemplo, le método agregarLugar:

```
public void agregarLugar(Lugar l) { if
		(!(listaLugares.contains(l))) {
		listaLugares.add(l);
		cargarAristas(l);
		}
	}
```

En el cual recibe un lugar como parámetro y pregunta si ese lugar no está contenido en la lista “listaLugares”, sino lo está la agrega a dicha lista y también llamará al método “cargarAristas” que recibirá como parámetro el mismo que “agregarLugar”.

```
	private void cargarAristas(Lugar l) { for (int i = 0;
		i < listaLugares.size()-1; i++) {
			Arista arista = new Arista(listaLugares.get(i),
l); listaAristas.add(arista);
		}
	}
```

El método anterior recorre con un ciclo la lista “listaLugares” y creara una arista con el lugar pasado por parámetro y la con la respectiva posición obtenida por el índice del ciclo que recorre la lista, agregándola a la lista “listaAristas”.

Dentro de la clase “Lógica” realizamos los métodos que van a llevar a cabo el funcionamiento del proyecto, en el cual genera un grafo (contextualizado al trabajo serían los puntos que se

nosotros creamos) con su respectiva lista de coordenadas. La clase posee un método llamado "arbolGeneradorMinimo":

```
public LinkedList<MapPolygonImpl> arbolGeneradorMinimo() {
          caminosMinimos.clear(); aristasPrim
          = grafo.aristasConAGM();

          for (Arista arista : aristasPrim) {
               Coordinate c1 = arista.lugarA.getCoordenada();
               Coordinate c2 = arista.lugarB.getCoordenada();

               MapPolygonImpl camino = new MapPolygonImpl(c1, c2,
c1); caminosMinimos.add(camino);
          }
          return caminosMinimos;
     }
```

El método anterior junta las coordenadas y lo guarda en una lista (camino más óptimo). Eso lo realiza por métodos que ya están implementados. En líneas generales acopla todas las coordenadas y las deja listas para después dibujarla en la interfaz del mapa.

Por otra parte, dentro de la clase tenemos el método "altaLugar":

```
public void altaLugar(String nombre, Coordinate c) {
          Lugar l = new Lugar(nombre, c);
          coordLugares.add(c);
          grafo.agregarLugar(l);
          actualizarCaminos();
     }
```

Este método lo que hace básicamente es cargar un lugar (sección de interfaz) y actúa como un intermediario ya que después queda almacenado en la sección de negocio.

También tenemos el método "pesoTotalArbolPrim":

```
public double pesoTotalArbolPrim(){
          double sumaTotal = 0; for
          (Arista a : aristasPrim) {
          sumaTotal += a.Distancia();
          }
          return sumaTotal;
```

```
    }
```

El método anterior tiene como función recorrer todas las aristas y hacer la sumatoria de las distancias para ver cuál es el peso total del árbol.

Luego por su parte, se utiliza el resultado de tal sumatoria, para poder brindarle al usuario, cual sería el costo final para su red telefónica, lo obtenemos multiplicando esta sumatoria (peso total del árbol) y el precio por metro (que se capturo, al momento de iniciar la ejecución del programa, en el menú principal), dando como consecuencia el valor exacto de los costos que tendría. Además, mostramos por medio de interfaz gráfica, a través de labels, información complementaria, como el costo total del árbol, el costo monetario de la red, y el costo de producción por metro, todo esto para que sea más ameno, amigable e informativo al consumidor final de nuestro software.

El método “noEsUnNumero” verifica si no es un numero para que, en la pantalla de carga, el usuario ponga un número y no una letra.

```
public static boolean noEsUnNumuero(char c) { return     ((c
        < '0') || (c > '9')) && (c != '\b');
    }
```

La clase “LugaresJSON” la creamos fundamentalmente para poder almacenar los datos de las localidades ingresadas por el usuario y poder preservarlos entre una ejecución de la aplicación y la siguiente, para ello utilizamos la librería JSON. Cada vez que iniciamos la aplicación el método “abrirJSONyCopiar”, guarda los datos que están en el archivo JSON en la variable “jsonConLugares” y así mantener la información en un “JsonArray”.

```
    protected static void abrirJSONyCopiar() {
        LugaresJSON abrirArchivoJSON =
        LugaresJSON.leerJSON(rutaDeJSON); if
        (abrirArchivoJSON != null) { for
        (JsonElement l :
        abrirArchivoJSON.getListaLugares()) {

    LugaresJSON.jsonConLugares.agregarLugar((JsonObject) l);
            }
```

```
            } else {
                guardarLocalidadesEnJSON();
            }
      }
```

Luego cuando el usuario crea el árbol generador mínimo, guardamos los lugares que el usuario selección en la variable “jsonConLugares”, para esto transformamos el objeto lugar que el usuario selecciono, en un objeto “JsonObject” con el método “transformarEnJson”. Luego lo agregamos a la variable “jsonConLugares” para que finalmente el método “guardarLocalidadesEnJSON” se encargue de actualizar la información nueva en el archivo JSON.

```
protected static void guardarLocalidadesEnJSON() {
          String nuevoDatosJSON =
LugaresJSON.jsonConLugares.generarJSON();
          LugaresJSON.jsonConLugares.escribirJSON(nuevoDatosJSON,
rutaDeJSON);
      }
```

La clase “Lugar” la podríamos pensar como si fuera una localidad. Sería el objeto que contiene en sí el nombre de ese lugar y la coordenada, compuesta por latitud y longitud y además tiene la lista de vecinos, que almacenara los puntos que están cerca (luego se usara para verificar el camino más corto). Dentro de la clase tenemos el método “agregarVecino”: `public void agregarVecino(int indice,Lugar l) { listaVecinos.put(indice, l);`

```
      }
```

Este método agrega a la localidad actual le agrega el vecino que tiene.

También tenemos el método “calcularDistanciaA_b”:

```
public double calcularDistanciaA_B(Lugar lugarB) {

          double radioTierra = 6371; // km

          Double lat1 = Math.toRadians(coordenada.getLat());
          Double lon1 = Math.toRadians(coordenada.getLon());
          Double lat2 =
Math.toRadians(lugarB.coordenada.getLat());
          Double lon2 =
```

```
Math.toRadians(lugarB.coordenada.getLon());

            double dlon = (lon2 - lon1);
            double dlat = (lat2 - lat1);

            double sinlat = Math.sin(dlat / 2);
            double sinlon = Math.sin(dlon / 2);

            double a = (sinlat * sinlat) + Math.cos(lat1) *
            Math.cos(lat2) * (sinlon * sinlon);
            double c = 2 * Math.asin(Math.min(1.0, Math.sqrt(a)));

            double distanciaEnMts = radioTierra * c * 1000;

            return (int) distanciaEnMts;
      }
```

Este método lo que hace es calcular la distancia que hay entre un punto y otro (punto A y punto B) y retorna dicho calculo en metros.

El resultado de esta operatoria en metros, fue chequeado y lo logramos corroborar de que los datos son verídicos (expresados en metros reales), por medio de una calculadora en línea externa a nuestro programa, que verifica el mismo objetivo en común que estamos realizando, para poder lograr esto, investigamos sobre algunos temas externos que desconocíamos como, posiciones geográficas y cálculos geoespaciales. Más que nada tuvimos presente este detalle, para poder brindarle un escenario real al usuario, y no valores estimativos o que no representasen nada relevante.

En el package “Test” creamos las clases para poder realizar los respectivos tests unitarios de las clases que se encuentran dentro del package “Negocio”. No profundizaremos demasiado en el informe sobre los tests ya que los mismos nos proveen de la información necesaria para entender el código (funcionan como documentación del proyecto).