Programación III - COMISIÓN 02 - Universidad de General Sarmiento

- Trabajo Práctico 2 -

1er. Cuatrimestre 2023

Profesores

Daniel Bertaccini - Gabriel Carrillo

Alumnos

Cristian Pereyra López - Lautaro Lombardi - Jeremías Ale Carbajales

INTRODUCCIÓN

En el presente trabajo práctico planteamos una posible solución para el análisis e implementación del problema de planificación de redes de cableado dado como consigna del trabajo práctico nro 2. En el mismo intentamos utilizar todos los conceptos y herramientas aprendidos en clase con fines utilitarios y pragmáticos para la mejor resolución de todos los problemas que fuimos encontrando en su desarrollo. e

En cuanto a código procuramos utilizar y sacar provecho en todo momento de los pilares de la POO que vimos en materias anteriores como encapsulamiento y abstracción y conceptos a respetar en cuánto a producir software de buena calidad presentes en las lecturas de la materia.

Nuestra implementación se basa en el patrón de diseño arquitectónico *"separated presentation"*, es decir, el código que se utiliza para la interfaz debe estar bien encapsulado y separado del código de negocio que ejecuta las funciones de los cálculos para conectar las localidades. Nos ajustaremos al patrón de diseño de 3 capas (vista-negocios-datos) separando en GUI (Graphical User Interface) lo referido a la parte visual, Logic lo pertinente a lógica de negocio y dal (Data Access Layer) a lo correspondiente a la solicitud tanto en APIs o persistencia de datos.

Además, para este trabajo se añaden los conceptos de árbol y árbol generador mínimo, como herramientas fundamentales para la resolución del problema planteado.

PRESENTACIÓN – CONECTANDO CIUDADES

El objetivo del trabajo práctico es implementar una aplicación para planificar conexiones telefónicas entre localidades ubicadas en regiones despobladas. Dado un conjunto de localidades, se debe proporcionar un árbol generador mínimo entre ellas e informar el costo de la solución.

La aplicación debe permitir al usuario registrar una serie de localidades, incluyendo el nombre, provincia, latitud y longitud de cada una. Además, se deben proporcionar los siguientes costos:

- Costo en pesos por kilómetro de una conexión entre dos localidades.
- Porcentaje de aumento del costo si la conexión tiene más de 300 km.
- Costo fijo que se agrega si la conexión involucra localidades de dos provincias distintas.

Cuando el usuario solicita la planificación, se debe resolver el problema de árbol generador mínimo sobre el grafo completo dado por todas las localidades, y cuyas aristas tienen costos calculados de acuerdo con los ítems anteriores. Se debe presentar al usuario la solución, especificando las conexiones telefónicas a construir y el costo total de las instalaciones.

IMPLEMENTACIÓN

A partir de nuestra elección como arquitectura de diseño el modelado de tres capas y su idea de *"separated presentation"* se optó por realizar una división del código en tres paquetes: por un lado tenemos un paquete de GUI con la interfaz de usuario que contiene todo lo relacionado a la parte visual (el selector de costos, ciudades, y la gestión del mapa donde se muestra el agm), otro paquete de la lógica con las clases relativas a la lógica de negocio en donde se encuentra las clases que gestionan los datos de las ciudades rescatados, la conexión entre ciudades y la creación y gestión del agm, y por último un paquete de acceso a datos en donde se ubica las clases que interviene en la solicitud de datos a sus respectivas APIs o recupero en archivos de datos guardados de las distintas ciudades que el programa tiene guardadas.

Visual /IGU

Decidimos crear tres ventanas:

HomeCreen: la ventana de "Bienvenida", esta contiene 3 sliders que permiten regular los 3 costos primordiales de los cableados (costo por km del cableado, porcentaje de aumento si la conexión tiene más de 300 KM y el costo extra si las localidades están en distintas provincias) y el botón "continuar a selección de ciudades" que nos permite dejar fijados los costos y pasar a la siguiente ventana.

MainScreen: la ventana de gestión principal, aquí el usuario puede elegir las ciudades que quiera agregar para realizar el agm (además de eliminar alguna que seleccionó y no quiere agregar), contiene un ComboBox que lista las ciudades precargadas por el programa disponibles para agregar, una sección con cuatro TextField para agregar una ciudad que no esté precargada (ciudad, provincia, latitud y longitud), un JTable donde van agregándose las ciudades elegidas por el usuario y al lado de este está el botón *"Generar Conexión*", para indicarle al programa que genere el AGM y continúe a la siguiente ventana.

MapScreen: la ventana de final donde se muestra un JMap con la conexión realizada junto con una lista de todas las conexiones realizadas, sus costos por conexión y el costo total de la conexión total.

Lógica de Negocio/Logic

En cuanto a lógica se decidió implementar el sistema de generación por medio de una serie de clases que trabajan con grafos. Una clase de Grafo de lista de vecinos, la clase CompleteGraph quién es la que genera un grafo completo con las ciudades que pide el usuario (de donde saldrá el AGM), una clase de grafo con peso (que extiende de NeighborListGraph), una clase City (objeto ciudad con sus 4 datos fundamentales), una clase BFS (recorrido de grafos) y una clase Prim (generador de AGM). Todas estas clases son “gestionadas” por la clase ConnectingCities, que es una clase que se encarga del funcionamiento core de la aplicación, manejando objetos de distintas clases y capas en el camino de construcción del AGM. También incluímos acá la funcionalidad de construcción del POJO City para desacoplar las demás clases con la clase City, y las validaciones de parámetros antes de su definitiva creación.

DAL/Data Access Layer

En esta capa de la aplicación planteamos dos formas de acceder a los datos necesarios de las ciudades; una vía consumo de APIs y otra precargada. En cuanto a ciudades utilizamos una API de la Secretaría de la Nación la cual contiene un listado de todos los municipios del país en formato JSON. Una vez solicitado ese JSON, filtramos los datos que nos interesan y construimos una lista de ciudades. De forma similar recuperamos el recurso de valor del dólar de una API. Ambas solicitudes están cubiertas ante fallo de request con valores preseteados; el caso de las ciudades con un archivo JSON y el de valor del dólar un valor estipulado en el archivo Config de la aplicación. Ambos retrievers de las APIs tienen seteado un timer para lograr la conexión y consumo de datos, de lo contrario los consiguen de la utilizando la persistencia.

La interacción entre las tres capas es la adecuada al fin del funcionamiento de la aplicación, y acotado a lo respectivo de cada una.

**además creamos una clase Config en la cual se pueden resetear algunos valores de la aplicación tales como url de las APIs, costo máximo de kilómetro cableado, el valor por default del dólar, etc.*

PROBLEMAS ENCONTRADOS

Problemas a destacar más allá de los triviales al pensar y codear nuestro trabajo práctico:

- **Complejidad Algoritmo de Prim:**

1era implementación de selección de arista mínima:

```
private Edge selectMinimumEdge(WeightedGraph completeGraph, WeightedGraph mstGraph) {
                City nonMarkedminimumVertex = completeGraph.getVertex(0);
                Edge minimumEdge = null;
                     minimumEdge = new Edge(completeGraph.getVertex(0), completeGraph.getVertex(1),
                                                                                Float.POSITIVE_INFINITY);

                for (City vertex : markedVertexes) {
                       Set<City> vertexes = completeGraph.neighbors(vertex);
                       Iterator<City> iter = vertexes.iterator();
                       while (iter.hasNext()) {
                              City currentVertex = iter.next();
                              if (!BFS.reachables(mstGraph, vertex).contains(currentVertex)) {
                                           if (completeGraph.getEdgeWeight(vertex, currentVertex) <
                                                                                minimumEdge.getPeso()) {
                                                  if (!mstGraph.existsEdge(vertex, currentVertex)) {
                                                         nonMarkedminimumVertex = currentVertex;
                                                                        minimumEdge = new Edge(vertex,
                                                                                nonMarkedminimumVertex,

completeGraph.getEdgeWeight(vertex, nonMarkedminimumVertex));
                                                  }
                                           }
                                    }
                       }
                }
                markedVertexes.add(nonMarkedminimumVertex);
                return minimumEdge;
        }
}
```

2da implementación de selección de arista mínima:

```
private Edge findMinimumEdge(WeightedGraph completeGraph, WeightedGraph mstGraph) {
            Edge minimumEdge = null;
            double minimumWeight = Float.POSITIVE_INFINITY;
            for (City city : markedVertexes) {
                  Set<City> neighbors = completeGraph.neighbors(city);
                  for (City neighbor : neighbors) {
                        if (!isConnectedToMST(neighbor) && !mstGraph.existsEdge(city,
                                                                        neighbor)) {
                         double weight = completeGraph.getEdgeWeight(city, neighbor);
                  if (weight < minimumWeight) {
                                    minimumWeight = weight;
                                    minimumEdge = new Edge(city, neighbor, weight);
                              }
                        }
                  }
            }
            return minimumEdge;
      }
```

El primer código que planteamos tenía una complejidad de **O(n^3)**, mientras que el segundo código tiene una complejidad de **O(n^2)**. La diferencia en complejidad se debe a la forma en que se selecciona la arista mínima en cada iteración del algoritmo. En el primer código, la función *selectMinimumEdge* iteraba sobre todos los vértices marcados y, para cada vértice, iteraba sobre sus vecinos para encontrar la arista de peso mínimo que no estaba en el grafo MST actual. Esto implicaba dos bucles anidados, lo que resultaba en una complejidad de **O(n^2)** en el peor caso. Sin embargo, como este proceso se repitia n-1 veces en el bucle principal, la complejidad total era de **O(n^3)**. En el segundo código, la función *findMinimumEdge* también itera sobre todos los vértices marcados, pero en lugar de iterar sobre todos los vecinos de cada vértice, simplemente verifica si un vecino no está conectado al MST actual y si no existe una arista entre el vértice y su vecino en el MST actual. Esto se logra utilizando el conjunto *markedVertexes* y el método *existsEdge* del grafo MST. Como resultado, se reduce el número de iteraciones en el bucle interno, lo que resulta en una complejidad de **O(n)** en el peor caso para encontrar la arista mínima. Dado que este proceso se repite n-1 veces en el bucle principal, la complejidad total es de **O(n^2)**. En conclusión, nuestra segunda implementación de Prim tiene una complejidad mejorada en comparación con el primer código debido a la forma en que se selecciona la arista mínima, lo que resulta en una complejidad de **O(n^2)** en lugar de **O(n^3)**.

- **Implementación del algoritmo de Prim:**

A la hora de la implementación para crear el agm, estábamos teniendo problemas para el correcto funcionamiento de Prim. Por la forma de implementación, se necesitaba al comienzo tener alguna arista auxiliar de referencia para que pudiera trabajar correctamente, sin embargo, si elegíamos un valor del propio grafo, el algoritmo elegía mal los caminos. Esto se solucionó temporalmente con la implementación de una arista auxiliar inicial de peso “infinito” (el mayor valor que permite Java).

```
Float.POSITIVE_INFINITY;
```

- **AGM correcto, pero el dibujo no se realiza correctamente:**

En la versión de JMapViewer utilizada, la única forma de poder dibujar líneas es por medio de la función “Polygon”, el problema es que, como su nombre indica, la función dibuja un polígono en vez de una línea, por lo que suele pedir mínimo tres parámetros (puntos) para su diseño, al utilizar solo dos, se dibujaban mal las líneas en el mapa. La solución fue simple ya que simplemente había que repetir dos veces el segundo vértice como segundo y tercer parámetro..

- **No se muestra la lista de ciudades seleccionadas:**

Cuando se seleccionaba una ciudad, si bien esta quedaba seleccionada “por detrás”, esta no aparecía visualizada como seleccionada en el MainScreen. Para solucionarlo, se decidió crear un JTable en el cual se muestre las ciudades agregadas y además sean seleccionables para poder deseleccionarlas.

- **Redondeo de montos:**

Al usar latitudes y longitudes específicas, los resultados de los montos quedaban con demasiados decimales. Esto se solucionó usando un valor de dólar, un costo de dólares por kilómetro y multiplicándolo por kilómetros de distancia entre localidades, para después redondear hacia arriba o hacia abajo según corresponda. En este momento pensamos la implementación de Big Decimal, pero la desestimamos por alcanzar para la resolución del problema con la primera idea.

- **Generación de mapa exageradamente lenta ante muchas ciudades (complejidad de Prim) / java.lang.OutOfMemoryError: Java heap space:**

  En las primeras versiones del programa, cuando se agregaban muchas ciudades, el programa tardaba muchísimo tiempo en crear el agm. Esto nos parecía extraño y nos llevó a varias revisiones de código para ver que estaba teniendo tanta complejidad.

  Resulta que el problema era nada más ni nada menos que el chequeo del valor de dólar, este chequeo se realizaba muchas veces dado que la función creaba constantes clases ConnectingCities, por lo que, debía conectarse exageradas veces a internet para chequear dicho costo, lo que generaba lentitud (ya que esta conexión tarda un poco, y mas con internet lenta). Esto se arregló cambiando este chequeo, haciendo que se ejecute una sola vez y se guarde el valor en una variable para todos los cálculos de costos siguientes.Lo mismo pasaba en un proceso en el que consumía repetidamente la API de ciudades y al mismo tiempo la volvía a consumir para generar una lista de provincias válida..

LINK A REPOSITORIO DE GITHUB

https://github.com/cristianpl99/tp_2_conectando_ciudades