Universidad Nacional de General Sarmiento

Trabajo Practico N2

Gabriela Fernández - Javier Santillán - Nicolás Luna

Docentes:

Alejandro Nelis
Fernando Torres

Fecha de entrega:

10 de Noviembre del 2020

Introducción:

El objetivo del trabajo práctico es implementar una aplicación para identificar automáticamente grupos de personas sobre la base de sus preferencias. De cada persona tenemos el nombre y su interés en cuatro campos distintos: los deportes, las noticias, la música y la ciencia. Lo que se intenta es agruparlos teniendo en cuenta estos intereses. La cantidad de interés en estos campos se expresa con un número del 1 al 5, para saber que dos personas tienen gustos parecidos se ve la similaridad entre ambos.
El índice de similaridad es una cuenta sencilla dada entre los interés de dos personas i y j :

$$similaridad(i, j) = |di - dj| + |ni - nj| + |mi - mj| + |ci - cj|.$$

La similaridad es 0 cuando ambas personas tienen gustos iguales, el valor aumenta cuanto menos parecidos son sus niveles de interés.

Separación de capas y denominación de clases

Nuestra aplicación es capaz de:

- Recibir y almacenar datos de personas
- Crear dos grupos de personas según sus intereses
- Mostrar ambos grupos de personas
- El promedio de interés referido a cada campo
- Actualizar una lista de personas en un .json

Para la implementación de la aplicación se decidió separar en varias capas:

- *Lógica*
- *Gráfica*
- *JSON*

Estas partes son responsables de un aspecto de la aplicación, la parte lógica cuenta con todo lo que le da funcionalidad al proyecto, mientras que la parte gráfica se encarga de brindar interacción entre la parte lógica y el usuario.

Capa Lógica:

- *Persona*
- *Grafo*
- *AGM*
- *Cluster*

Capa Gráfica:

- *Presentacion*
- *Form_ingreso*
- *Grupos*

JSON

- PersonasJSON
- ManejoDato

VerGrafo

- aletotidad
- arista
- panel
- vértice

Desarrollo y decisiones de implementación

En la capa Lógica:

En primer lugar se decidió implementar la clase Grafo la cual asemeja la estructura de un grafo con pesos y sus operaciones básicas, contiene la siguiente interfaz:

public void agregarVertice(Persona persona) → alta vertice

public boolean existePersona(Persona p) → consulta vértice

public void agregarArista(Persona persona1,Persona persona2) → alta arista

public void eliminarVertice(Persona p) → baja vertice

public void eliminarArista(Persona persona1, Persona persona2) → baja arista

public int cantAristas() → devuelve cantidad de aristas

public int cantVertices() → devuelve cantidad de vértices

public ArrayList<Integer> AristaMasPesada() → devuelve lista con los vértices de la arista más pesada

public ArrayList<Persona> vertices() → devuelve todos los vertices

public HashMap<Integer,Integer> getVecinos (Persona p) → devuelve vecinos a partir de una persona

public HashMap<Integer,Integer> getVecinos (int i) → devuelve vecinos a partir de un número de vértice

public ArrayList<HashMap<Integer,Integer>> getVecinos() → devuelve la lista de vecinos

public ArrayList<String> getNombredeVecinos (Persona p) → devuelve los nombres de los vecinos de una persona

public Persona getVertice(String nombre) → devuelve una persona por su nombre

public void setVecinos(ArrayList<HashMap<Integer, Integer>> vecinos) → setea lista de vecinos

public void setVertices(ArrayList<Persona> vertices) → setea lista de vertices

En segundo lugar implementamos la clase Cluster que tiene los métodos relacionamos al clustering de personas:

public void dividirGrupos() → genera un agm y le quita su arista más pesada

public void generadorDeSimilitudes() → relaciona todos los vértices entre con aristas de peso igual a la similitud entre las personas

En tercer lugar tenemos la clase AGM que hereda de Grafo y se encarga de transformar un Grafo Completo en un Árbol Generador Mínimo y que cuenta con los siguientes métodos:

public AGM(Grafo g) → El constructor de AGM
private void generar(Grafo ingresado) → Recibe un Grafo y genera otro con las características de un AGM
private Persona buscarVecinoMenorPeso(Grafo grafo, ArrayList<Persona> listaPersonas) → Busca y retorna el vecino con menor peso de entre una lista de vértices
public static Persona vecinoMenorPeso(Grafo grafo, Persona persona, ArrayList<Persona> excepciones) → Un método estático, retorna el vecino de menor peso de un vértice, excluyendo a los que pertenecen a una lista
private int calcularDistanciaTentativa(Grafo g) → calcula la distancia tentativa, un valor mayor al que pueden alcanzar las aristas del grafo

Por otro lado implementamos la clase BFS que se encarga de recorrer un grafo en orden de distancia creciente de los vertices, tiene la siguiente interfaz:

public static boolean esConexo(Grafo g) → Devuelve true si el grafo es conexo sino false
public static Set<Integer> alcanzables(Grafo g, int origen) → devuelve los vértices alcanzables a partir del número de un vértice origen.

También se implementó ClusteringHumano, la más importante, que une las funciones de Cluster y de ManejoDato:

public void agregarPersona(Persona persona) → Agrega una persona a ManejoDato.
public void dividirgrupos() → Inicializa un cluster, actualiza el .json, carga el cluster con todos los datos que se encuentran en el .json y divide el cluster en dos grupos.
public void inicializarCluster() → Inicializa el cluster de tamaño de la cantidad de personas que se encuentran cargadas en el ManejoDato.
public void actualizarJSON() → Guarda los nuevos datos cargados al archivo .json
public void cargarCluster() → Llena cluster con los datos que hay en el archivo .json
public double promedioInteres(int indice) → retorna el promedio de interés de un campo buscandolo por índice.

Por último tenemos la clase Persona que modela a una persona y tiene la siguiente interfaz:
public void clonar(Persona p) → clona los datos de una persona p en otra.
private int esIndice(int valor) → verifica que los valores de interés estén dentro del rango correcto, sino tira una excepción.
private void verificarNombre(String nombre) → verifica que el nombre no contenga caracteres inválidos, si es asi tira una excepción.

En la capa Gráfica:

La capa gráfica se encuentra dividida en dos, la parte auxiliar que crea elementos y setea la fuente e icono de la aplicación, y la parte de interfaz que usa estos auxiliares para crear la aplicación.

En la parte auxiliar se encuentran dos clases:

- *Clase Elementos* → que contiene métodos para la creación de botones, textField, separadores y labels. También se puede setear un Font general.
- *Clase estilo* → que contiene el seteo del título y el icono de la ventana.

En la parte interfaz se encuentra:

- *Presentacion* → Muestra una imagen de presentación de la aplicación y da las opciones de iniciar o salir.
- *Form_ingreso* → Muestra un formulario en el que se puede ingresar varias personas y especificar el nivel de interés en cuatro temas diferentes, se le dan al usuario las opciones de ingresar una persona, de generar la división del grupo o de salir de la aplicación.
- *Grupos* → Muestra los dos grupos de personas, el promedio de interés sobre cada tema, y dibuja un grafo. Se le dan las opciones al usuario de volver a ingresar personas o de salir de la aplicación.

JSON

Se usó la librería GSON para crear un archivo .json que contiene las personas ingresadas al grafo, esto con el objetivo de poder cargar las personas antes cargadas cuando se vuelve a iniciar la aplicación. Cuando se agregan personas esta lista se actualiza.

- PersonasJSON → guardan las personas y genera el .json
- ManejoDato → Usa PersonasJSON. Crea un .json específico donde se guardan los datos de personas.

VerGrafo

Se crea el grafo en donde se va a ver por la interfaz. Vamos a tener una primera parte donde se inicializa los valores de la arista, vértice y los crea aleatoriamente. En la segunda parte crea y obtiene esos valores para pintar todo el grafo completo.

En una primera parte se encuentra la clases:

- Arista: recibe el valor que obtiene del peso y consigue las coordenadas x e y de las personas a la cual tiene que unir. En el método paint pinta la arista o línea con un color random.
- Vértice: recibe el nombre y las coordenadas donde se va a ubicar (con un x e y). Con esos datos en el método paint lo pinta.
- Aleatoriedad: para que no se pisen los nodos. Guardamos los puntos de cada vértice, creamos los círculos. Pregunta si el rango excede o limita al vértice anterior. Si es así lo crea.

En la segunda parte se encuentra:

- Panel: dibuja el grafo en el panel. En el constructor creamos una lista que contiene los vértices y sus vecinos. En el método crearMapa creamos los vértices y lo guardamos en una lista llamada conjunto en el cual lo transformamos al vértice en un String y un Point, para luego cuando queremos crear la arista obtenemos su ubicación y podemos pintar la línea o conexión.

Conclusión

Pudimos comprobar durante el desarrollo del proyecto que al separar la lógica de lo visual y elegir nombres descriptivos para sus clases, funciones o métodos, así como también elegir qué tarea específica iban a realizar en cada clase, se nos hacia mas fácil ver que parte del cluster fallaba e ir directo a las funciones que encontrábamos defectuosas para luego modificarlas.

La implementación de los tests unitarios nos ayudaron rápidamente a identificar si los métodos de la capa de negocio cumplían bien su funcionamiento y a actualizar el código cuando teníamos que cambiarlo.