UNIVERSIDAD NACIONAL DE GENERAL SARMIENTO

LICENCIATURA EN SISTEMAS

PROGRAMACIÓN III

TRABAJO PRÁCTICO II: DISENANDO REGIONES

FECHA DE ENTREGA: 14/05/2024

PROFESORES:

PATRICIA BAGNES

IGNACIO SOTELO

ALUMNOS:

FRANCISCO DOMINGUEZ

MELINA GOMEZ

MICAELA BARBARA PALOMEQUE

2024

INTRODUCCIÓN

La idea de dividir regiones geográficas en unidades de menor tamaño, como provincias o estados, es importante para tomar decisiones y asignar recursos en diferentes contextos (por ejemplo, administrar recursos naturales, planificación urbana, logística, etc.).

Por ese motivo, nos vimos en la necesidad de diseñar un sistema automatizado para la división de regiones geográficas, donde el usuario pueda situarse en cualquier parte del mundo y marcar cuantos lugares quisiera, brindando datos de los mismos.

Nuestra misión fue implementar algoritmos eficientes para la identificación y delimitación optima, utilizando técnicas de grafos y herramientas de procesamiento de datos geográficos. Todo esto, acompañado de una interfaz grafica amigable, donde se pudo representar gráficamente la ruta que se solicita, incluyendo estadísticas que aportan una gran información al cliente.

En este documento, se detallará el proceso de la implementación que llevamos a cabo y como la modelamos para llegar al sistema mencionado.

IMPLEMENTACIÓN

Para la implementación del sistema, optamos por organizar el código en 5 paquetes, cada uno con un propósito específico. De esta manera, mantenemos una separación clara entre las diferentes partes que lo componen, lo cual permite tener una estructura clara para su desarrollo, y en caso de ser necesario, poder hacer modificaciones del sistema accediendo de manera más fácil y prolija.

- Paquete *diseñando_regiones* que se encarga de la creación y análisis de regiones geográficas en un mapa. A su vez el mismo cuenta con las clases: "*CoordenadasContinentes*", "*dibujanteDeLineas*", "*diseñando_regiones*" y "*estadisticasRegiones*" para una implementación más ordenada de la misma.

- Paquete **grafo**, con una única clase "*Grafo*", la misma como su nombre lo indica va a representar unidades de grafos no dirigidos mediante una lista de adyacencia para almacenar los vecinos de cada nodo y una matriz de pesos para almacenar los pesos de las aristas. Para elegir como implementar las mismas, nos basamos en tener O(1) para cuando se requiera obtener los vecinos del nodo y el peso de una arista. La misma permitirá tomar acciones sobre los mismos tales como agregar, validar o eliminar sus aristas y obtener los nodos, su peso, sus vecinos, entre otros.

- Paquete ***metodos_auxiliares***, cuenta con una clase "*Auxiliar*", la cual tendrá métodos auxiliares para operar con grafos, la misma cuenta con el método AGM(Grafo grafo) la cual generará el árbol generador mínimo mediante el algoritmo de Prim (Lo explicaremos más adelante en el informe con mayor detalle).

  En la clase mencionada, los métodos que destacan son:

  esGrafoConexo(Grafo grafo) que valida si el grafo brindado por parámetro es conexo, utilizando el algoritmo de búsqueda BFS, es decir, recorre el grafo y marca los nodos alcanzables desde un nodo arbitrario. Si cuando éste recorrido finaliza, los nodos marcados son la totalidad de nodos del grafo, entonces el grafo es conexo.

```java
public static boolean esGrafoConexo(Grafo grafo)
{
    if (grafo.getCantNodos() == 0 || grafo.getCantNodos() == 1)
    {
        return true;
    }
    else
    {
        Queue<Integer> listaPendientes = new LinkedList<Integer>();
        HashSet<Integer> marcados = new HashSet<Integer>();
        listaPendientes.add(0);

        while (!listaPendientes.isEmpty())
        {
            Integer actual = listaPendientes.poll();
            marcados.add(actual);
            for (Integer vecino : grafo.getVecinos(actual))
            {
                if (!marcados.contains(vecino))
                {
                    listaPendientes.add(vecino);
                }
            }
        }
       return marcados.size() == grafo.getCantNodos();
    }
}
```

generarComponentesConexas(Grafo grafo, int k): Este método genera k componentes conexas en un grafo eliminando las aristas de mayor peso de forma iterativa. Comienza eliminando la arista de mayor peso y continúa hasta que se hayan generado k componentes. La misma utiliza una instancia de EstadisticasRegiones para contar los nodos en cada región y almacena esta información en la lista estadísticas.

```java
public  Grafo generarComponentesConexas(Grafo grafo, int k)
{

    while (k-1 > 0) {
        int mayorPeso = 0;
        int origen = -1;
        int destino = -1;
        // Recorrer todas las aristas del grafo
        for (int i = 0; i < grafo.getCantNodos(); i++) {
            for (int j = 0; j < grafo.getCantNodos(); j++) {
                if (grafo.existeArista(i, j) && grafo.getPeso(i, j) > mayorPeso) {
                    mayorPeso = grafo.getPeso(i, j);
                    origen = i;
                    destino = j;
                }
            }
        }
        if (origen != -1 && destino != -1) {
            // Eliminar la arista de mayor peso encontrada
            grafo.eliminarArista(origen, destino);
            k--;
        }
    }
```

Luego cuenta con más métodos que permiten obtener las estadísticas de las componentes conexas y verificar si el grafo ya fue dividido en componentes conexas, por ejemplo.

- Paquete **interfaz**, en este paquete gestionamos la interfaz de usuario e interactividad del mismo con nuestro programa. La misma se divide en las siguientes clases:

  La clase "*VentanaMenu*" se encarga de gestionar la interfaz de usuario para seleccionar el continente y la cantidad de provincias o estados a visitar, y luego abrir la ventana del mapa correspondiente. (Nos mostrará un error en caso que no se informe la cantidad de provincias a visitar).

  la clase "*graficoEstadisticas*" se encarga principalmente de la visualización de los datos estadísticos en forma de gráfico de barras, para calcularlos tiene un método que calcula el ancho de las barras, determina la escala de los valores, y dibuja las barras y etiquetas correspondientes a cada región. El usuario puede acceder a ella desde los controles en la clase Mapa.

  La clase "*Mapa*" se encarga principalmente de manejar la interacción del usuario con el mapa, permitiendo agregar aristas, cambiar la similaridad, generar el AGM y dividirlo en regiones. También gestiona la visualización de estadísticas y el reinicio de la aplicación, todo mediante un panel con los botones necesarios para que el mismo pueda accionar.

  Para logarlo, además de implementar los métodos para que el usuario opere, destacamos el siguiente método en la clase:

```
private void botonArbol()
{
      btnArbol = new JButton("Crear mejor ruta");
      letra = new Font("Comic Sans MS", Font.BOLD, 12);
      btnArbol.setFont(letra);
      btnArbol.addActionListener(new ActionListener() {
          public void actionPerformed(ActionEvent arg0) {
              if(diseñoRegiones.cantAristas()>0 && mejorRuta==null && Auxiliar.esGrafoConexo(diseñoRegiones.getPais())) {
              diseñoRegiones.generarAGM();
              mejorRuta= diseñoRegiones.getAGM();
              dibujarAristasAGM(mejorRuta);
          }
              else {
                  JOptionPane.showMessageDialog(null, "Por favor, todos los caminos deben estar conectados");
              }
          }

      });

      int yPosition = btnCambiarSimilaridad.getY() + btnCambiarSimilaridad.getHeight() + 10;
      btnArbol.setBounds(10, yPosition, 195, 23);
      panelControles.add(btnArbol);
  }
```

  En método botonArbol() verificamos primero que exista al menos una arista en el grafo (diseñoRegiones.cantAristas()>0), que aun no se haya creado una mejor

ruta (mejorRuta==null) y que el grafo sea conexo (Auxiliar.esGrafoConexo(diseñoRegiones.getPais())).

Si las condiciones se cumplen, se genera el árbol generador mínimo y se guarda como la mejor ruta. Luego se dibujan las aristas del AGM en el mapa mediante el método dibujarAristasAGM(mejorRuta).

En caso que alguna de las condiciones no se cumpla el mismo arrojará un error para informarle al usuario lo que debe hacer.

Dentro del método mencionado, se llama al método generarAGM() el cual se encuentra en la clase "**Diseñando_regiones**" que se encarga de generar el árbol mínimo del grafo.

```
public void generarAGM()
{
    this.paisAGM = aux.AGM(pais);
}
```

Se utiliza el método AGM(pais) de la clase "**Auxiliar**" para calcular el AGM del grafo pais. Este método utiliza específicamente el algoritmo de Prim, es decir, a medida que se avanza en el grafo se va eligiendo el peso más chico y se van agregando al árbol para ir armándolo. A continuación, vemos como se implementó el método:

```java
public Grafo AGM(Grafo grafo)
{
    if (grafo.getCantNodos() > 0)
    {
        Grafo nuevoGrafo = new Grafo(grafo.getCantNodos());
        HashSet<Integer> marcados = new HashSet<Integer>();
        marcados.add(0);
        int cantMarcados = 1;

        while (cantMarcados < grafo.getCantNodos())
        {
            int menorPeso = Integer.MAX_VALUE;
            int origen = -1;
            int destino = -1;

            for (Integer nodoMarcado : marcados)
            {
                for (Integer vecino : grafo.getVecinos(nodoMarcado))
                {
                    if (!marcados.contains(vecino) && grafo.getPeso(nodoMarcado, vecino) < menorPeso)
                    {
                            origen = nodoMarcado;
                            destino = vecino;
                            menorPeso = grafo.getPeso(nodoMarcado, vecino);
                    }
                }
            }


            nuevoGrafo.agregarArista(origen, destino, menorPeso);
            marcados.add(destino);
            cantMarcados++;
        }
        return nuevoGrafo;
    }
    else
    {
        throw new RuntimeException("El grafo ingresado está vacío");
    }
}
```

El método se implementa principalmente con ciclo while el cual se repite hasta que todos los nodos estén marcados.

Dentro del mismo se inicializa la variable menorPeso con un valor muy grande y las variables origen y destino con el valor -1 (Son valores estimativos para iniciar). Luego, se recorren todos los nodos marcados (nodoMarcado) y se busca un vecino no marcado (vecino) con la arista de menor peso que conecta un nodo marcado con uno no marcado.

Si se encuentra una arista con un peso menor que menorPeso, se actualizan las variables origen, destino y menorPeso con los valores de la nueva arista encontrada. Una vez encontrada la arista de menor peso, se agrega esta arista al nuevoGrafo, conectando el nodo origen con el nodo destino y con el peso correspondiente.

Luego se marca el nodo destino como parte del AGM, se incrementa cantMarcados y se repite el ciclo hasta finalizarse. En caso que el grafo pasado por parámetro no tenga nodos (vacío) se lanza una excepción indicándoselo al usuario.

Luego se devuelve el nuevoGrafo, el cual será un árbol que abarca todos los nodos del grafo pasado por parámetro, con el menor peso posible.

- Paquete **test**, en este paquete (y último de los representados), generamos casos de prueba para probar el funcionamiento de los métodos implementados en AGM, Aristas y Vecinos, cada uno respectivamente en su clase. Realizamos diversas pruebas para observar cómo se comportaban los métodos según los diferentes casos de información utilizados en las pruebas. Cabe mencionar que, gracias a esta implementación, fuimos capaces de reconocer errores y corregirlos.

  Algunos de los implementados fueron los siguientes, cuyos nombres informan lo que se desea verificar en el test.

```
@Test
public void aristaExistenteTest()
{
    Grafo grafo = new Grafo(5);
    grafo.agregarArista(2, 3,10);
    assertTrue(grafo.existeArista(2, 3));
}

@Test
public void aristaOpuestaTest()
{
    Grafo grafo = new Grafo(5);
    grafo.agregarArista(2, 3,7);
    assertTrue(grafo.existeArista(3, 2));
}

@Test
public void aristaInexistenteTest()
{
    Grafo grafo = new Grafo(5);
    grafo.agregarArista(2, 3,2);
    assertFalse(grafo.existeArista(1, 4));
}

@Test
public void agregarAristaDosVecesTest()
{
    Grafo grafo = new Grafo(5);
    grafo.agregarArista(2, 3,2);
    grafo.agregarArista(2, 3,2);

    assertTrue(grafo.existeArista(2, 3));
}
```

CONCLUSIÓN

En conclusión, esta experiencia no solo nos permitió mejorar en el campo de programación de Java, sino que también nos llevó a profundizar en el uso de herramientas como WindowBuilder, con la cual ya teníamos cierta experiencia previa, así como a explorar y aplicar nuevas herramientas como JMapViewer, que resultó fundamental para la generación de divisiones de regiones geográficas en nuestro proyecto. Nos encontramos con nuevos desafíos que nos enseñaron distintas formas de implementar lo que se requería, fuimos capaces de captar y solucionar errores y volcamos lo aprendido en clase en el código, tal como se mencionó al momento de detallar los métodos (Tales como el algoritmo de Prim y búsquedas BFS).

En resumen, este proyecto fue una oportunidad para aplicar los conocimientos adquiridos en clase y para aprender de forma práctica, fortaleciendo nuestras habilidades en el desarrollo de software y la resolución de problemas.