Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Ciencias y Sistemas
Departamento de Matemática
Matemática para Computación 2
Primer Semestre 2023

Proyecto

Algoritmo de búsqueda en profundidad

Carnet	Nombres	Apellidos
201807185	Abraham Moises David	Fuentes Ramirez
202200226	Kevin Saul	Godínez Pérez

Manual Técnico

Descripción del sistema

El objetivo de la aplicación es mostrar el uso del Algoritmo de Búsqueda en profundidad por medio de la generación automática de los grafos.

la aplicación solo resuelve el algoritmo DFS por lo que cualquier otro algoritmo que da fuera del alcance de la aplicación.

Tecnologías Utilizadas:

lenguaje de programación: **Java**
Generador de Grafos: **Graphviz**

Cómo está construido

todo el trabajo de generación de los grafos lo realiza la clase Grafo la cual detallamos a continuación:

La clase Grafo tiene dos atributos principales: vértices, que es una lista de objetos Vértice, y aristas, que es una lista de objetos Arista.

los métodos que trabajan para realizar los grafos se describen a continuación:

1. `public Grafo()`: Este es el constructor de la clase Grafo. Inicializa las listas de vértices y aristas como listas vacías.

```
public Grafo() {
        this.vertices = new HashMap<>();
        this.aristas = new ArrayList<>();
    }
```

2. `public void agregarVertice(Vertice vertice)`: Este método agrega un objeto Vertice al grafo. Añade el vértice a la lista de vértices si no está presente.
3. `public void agregarArista(Arista arista)`: Este método agrega un objeto Arista al grafo. Añade la arista a la lista de aristas si no está presente.
4. `public Vertice obtenerVertice(String id)`: Este método busca y devuelve un objeto Vertice en el grafo, basándose en su identificador (id). Si no se encuentra ningún vértice con el id dado, devuelve null.
5. `public List<Vertice> dfs(String idInicio)`: Este método realiza una búsqueda en profundidad (Depth-First Search, DFS) en el grafo, comenzando desde el vértice con el id especificado. Devuelve una lista de vértices visitados en el recorrido.

```
    public List<Vertice> dfs(String idInicio) {
        Vertice inicio = obtenerVertice(idInicio);
        if (inicio == null) {
            throw new IllegalArgumentException("El vértice de inicio no existe en el grafo.");
        }

        List<Vertice> visitados = new ArrayList<>();
        dfsRecursivo(inicio, visitados);
        return visitados;
    }
```

6. `private void dfsRecursivo(Vertice actual, List<Vertice> visitados)`: Este método es una función auxiliar recursiva para la búsqueda en profundidad (DFS). Se llama a sí mismo con cada vértice

adyacente no visitado al vértice actual.

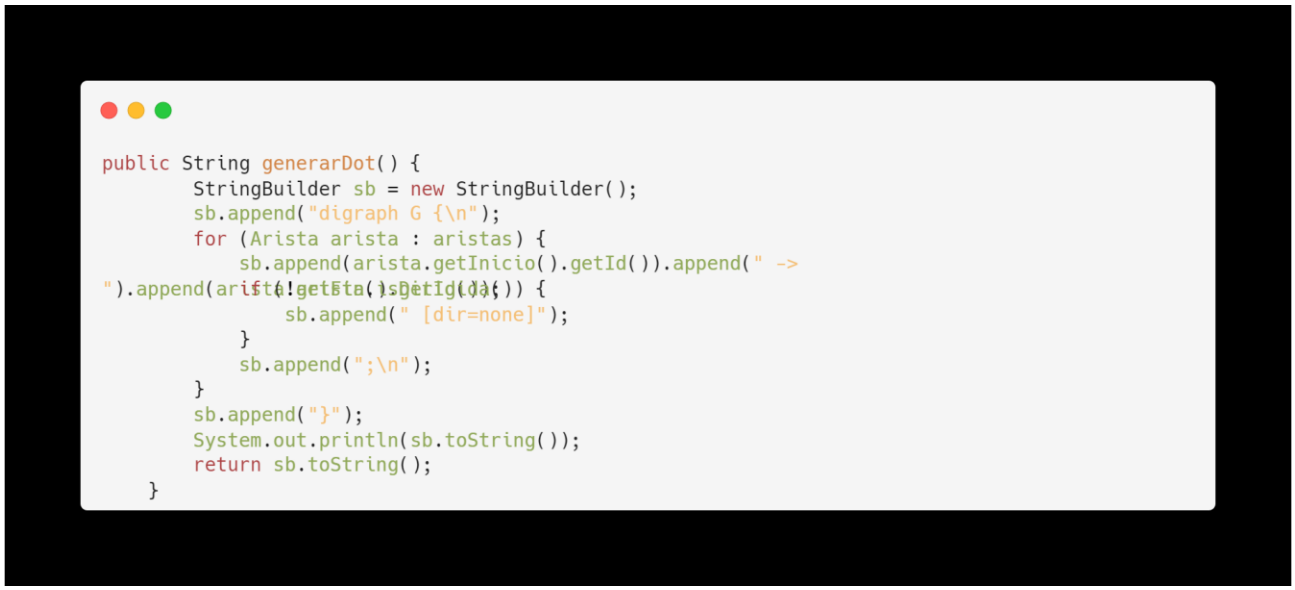
```
private void dfsRecursivo(Vertice actual, List<Vertice> visitados) {
        visitados.add(actual);
        for (Vertice adyacente : obtenerVerticesAdyacentes(actual)) {
            if (!visitados.contains(adyacente)) {
                dfsRecursivo(adyacente, visitados);
            }
        }
    }
```

7. `public Grafo generarGrafoRecorridoDFS(List<Vertice> recorrido)`: Este método genera un nuevo grafo a partir de un recorrido DFS dado. El nuevo grafo contiene solo los vértices y las aristas presentes

en el recorrido.

```java
public Grafo generarGrafoRecorridoDFS(List<Vertice> recorrido) {
        Grafo grafoDFS = new Grafo();

        // Agregar los vértices al grafoDFS
        for (Vertice v : recorrido) {
            grafoDFS.agregarVertice(v);
        }

        // Agregar las aristas al grafoDFS
        for (int i = 0; i < recorrido.size() - 1; i++) {
            Vertice inicio = recorrido.get(i);
            Vertice fin = recorrido.get(i + 1);
            boolean aristaEncontrada = false;
            for (Arista arista : aristas) {
                if ((arista.getInicio().equals(inicio) && arista.getFin().equals(fin)) ||
                        (!arista.isDirigida() && arista.getInicio().equals(fin) &&
arista.getFin().equals(inicio))) {
                    grafoDFS.agregarArista(arista);
                    aristaEncontrada = true;
                    break;
                }
            }

            // Si no se encuentra la arista en el grafo original, crea una nueva arista y agrégala al
grafoDFS
            if (!aristaEncontrada) {
                Arista nuevaArista = new Arista(inicio, fin, false); // Asume que no es dirigida
                grafoDFS.agregarArista(nuevaArista);
            }
        }

        return grafoDFS;
    }
```

8. `public String generarDot()`: Este método genera una representación en formato DOT del grafo, que se puede utilizar para visualizarlo utilizando herramientas como Graphviz.



```java
public String generarDot() {
        StringBuilder sb = new StringBuilder();
        sb.append("digraph G {\n");
        for (Arista arista : aristas) {
            sb.append(arista.getInicio().getId()).append(" ->
").append(arista.getFin().getId());
            if (!arista.isDirigida()) {
                sb.append(" [dir=none]");
            }
            sb.append(";\n");
        }
        sb.append("}");
        System.out.println(sb.toString());
        return sb.toString();
    }
```

9. `public String generarDotRecorrido(List<Vertice> recorrido, List<Arista> aristasRecorrido)`: Este método genera una representación en formato DOT del grafo, resaltando el recorrido y las aristas proporcionadas. Este método es útil para visualizar el recorrido de un algoritmo, como la búsqueda en profundidad (DFS).

```java
public String generarDotRecorrido(List<Vertice> recorrido, List<Arista> aristasRecorrido) {
        StringBuilder sb = new StringBuilder();
        sb.append("digraph G {\n");

        for (Arista arista : aristas) {
            boolean aristaEnRecorrido = aristasRecorrido.stream().anyMatch(a ->
a.getInicio().equals(arista.getInicio()) && a.getFin().equals(arista.getFin()));
            sb.append("\t").append(arista.getInicio().getId()).append(" ->
").append(arista.getFin().getId());

            if (aristaEnRecorrido || !arista.isDirigida()) {
                sb.append(" [");
            }

            if (aristaEnRecorrido) {
                sb.append("color=red");
                if (!arista.isDirigida()) {
                    sb.append(", ");
                }
            }

            if (!arista.isDirigida()) {
                sb.append("dir=none");
            }

            if (aristaEnRecorrido || !arista.isDirigida()) {
                sb.append("]");
            }

            sb.append(";\n");
        }

        sb.append("}");
        System.out.println(sb.toString());
        return sb.toString();
    }
```

Conclusión:

La aplicación está construida de tal manera que pueda ser extensible para otros algoritmos, ya que muchos de los métodos que contienen pueden ser reutilizables.