Integrantes:

Janelly Dayanna Chacha Vélez

jchachav@est.ups.edu.ec

Kelly Valeria Guamán León

Kguamanl13@est.ups.edu.ec

Asignatura:

Estructura de datos

Docente:

Ing. Pablo Torres

Fecha:

09/02/2026

Carrera:

Computación

Tema:

Sistema de búsqueda de rutas en grafos

1. Introducción

Las estructuras no lineales permiten modelar problemas reales donde los elementos no siguen un orden secuencial. Una de las estructuras más importantes es el **grafo**, el cual permite representar redes de caminos, mapas, redes sociales o sistemas de transporte.

En el presente proyecto se desarrolló una aplicación gráfica que permite construir un grafo sobre un mapa e identificar rutas entre dos nodos seleccionados por el usuario. Para ello se implementaron los algoritmos de búsqueda **Breadth First Search (BFS)** y **Depth First Search (DFS)**, permitiendo visualizar el recorrido y comparar su comportamiento. El sistema además registra los tiempos de ejecución y mantiene persistencia de la información aun después de cerrar el programa.

2. Objetivos

2.1. Objetivo general

Desarrollar una aplicación que represente un grafo y permita analizar rutas utilizando algoritmos de búsqueda.

2.2.Objetivos Específicos

- Representar grafos mediante listas de adyacencia.
- Implementar BFS para búsqueda por niveles.
- Implementar DFS utilizando recursión.
- Visualizar gráficamente los recorridos.
- Medir el tiempo de ejecución de los algoritmos.
- Guardar y recuperar la información del grafo.
- Aplicar el patrón Modelo-Vista-Controlador.

3. Descripción del problema

En muchas aplicaciones reales, como los sistemas de navegación GPS, videojuegos, mapas interactivos y planificación de rutas, es necesario encontrar caminos entre diferentes ubicaciones. Este problema puede modelarse mediante grafos, donde cada ubicación representa un nodo y cada conexión entre ubicaciones representa una arista.

Sin embargo, determinar qué camino seguir no siempre es trivial. Dependiendo de la estrategia de búsqueda utilizada, el sistema puede encontrar rutas más rápidas, más cortas o simplemente cualquier ruta disponible.

Por ello, se requiere un sistema capaz de:

- Representar un mapa mediante grafos.
- Permitir la creación y almacenamiento de nodos y conexiones.
- Aplicar algoritmos de búsqueda de caminos.
- Visualizar la exploración del recorrido.

Para resolver este problema se implementó un sistema que utiliza los algoritmos de recorrido **BFS (Breadth-First Search)** y **DFS (Depth-First Search)** para encontrar rutas entre nodos dentro de un grafo.

4. Problema de solución

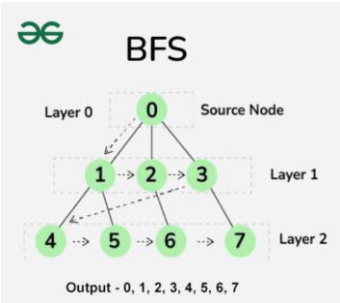
4.1. Marco Teórico

4.1.1. Grafos

Un grafo es una estructura de datos compuesta por un conjunto de vértices (nodos) y aristas (conexiones). Permite modelar relaciones entre elementos. En este proyecto se utilizó un grafo no ponderado representado mediante **listas de adyacencia** (Goodrich, Tamassia & Goldwasser, 2014)**.**

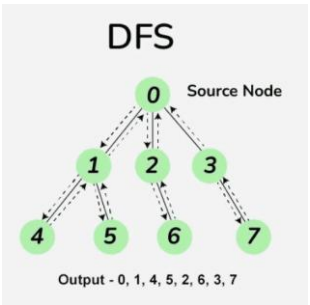
4.1.2. BFS (Breadth First Search)

BFS recorre el grafo por niveles utilizando una cola. Primero visita todos los vecinos inmediatos del nodo inicial y posteriormente continúa con los siguientes niveles. Su principal característica es que garantiza encontrar la **ruta más corta** en grafos no ponderados (Cormen et al., 2022).



4.1.3. DFS (Depth First Search)

DFS recorre el grafo en profundidad. Avanza por un camino hasta que no puede continuar y luego retrocede. Se implementa mediante **recursión**, utilizando implícitamente la pila del sistema. No garantiza la ruta más corta, pero suele requerir menos memoria (Cormen et al., 2022).

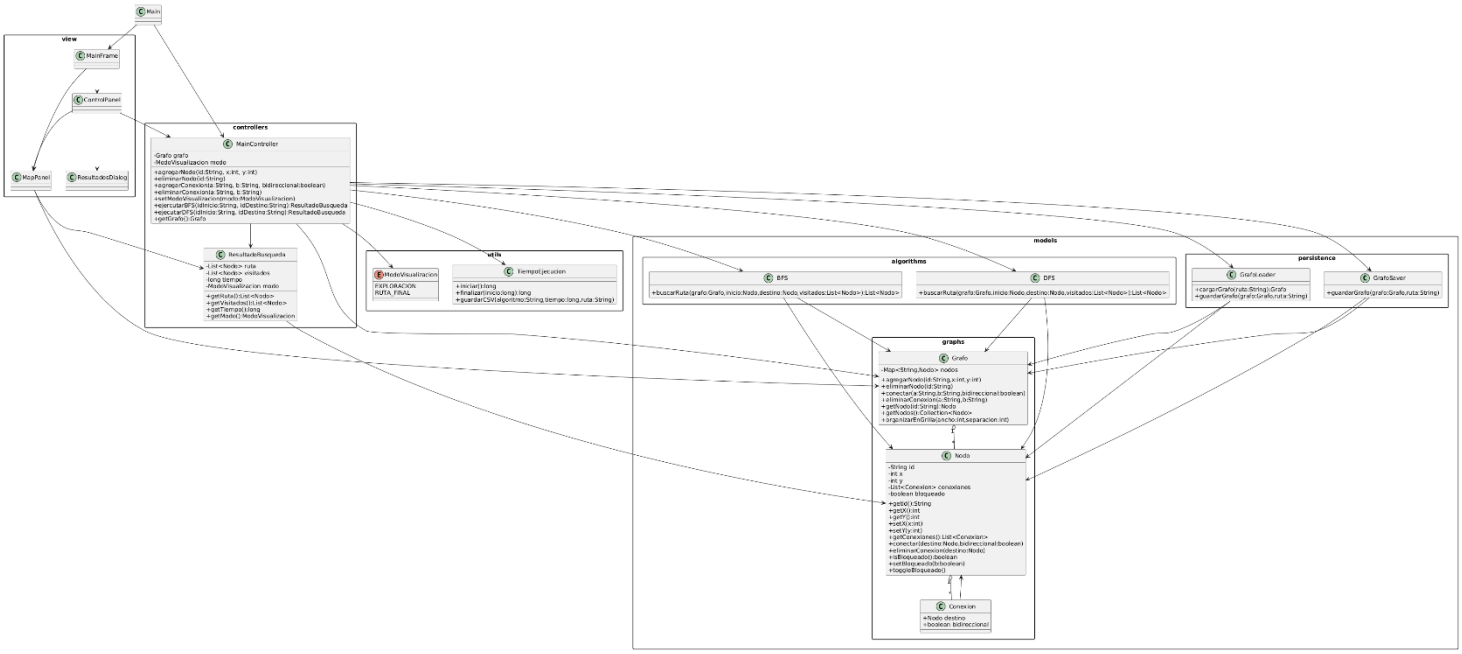


4.2. Tecnologías utilizadas

- Java 17 y Javax Swing
- Programación Orientada a Objetos
- Github

4.3.Diagrama UML

El sistema sigue una arquitectura modular.

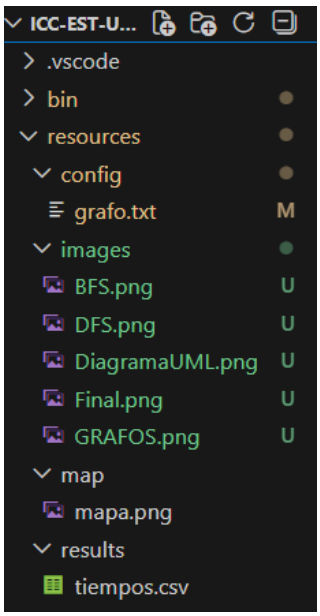


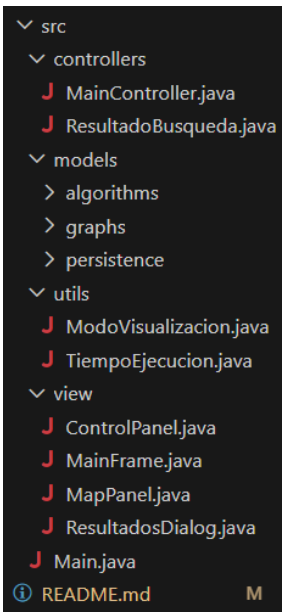
Las clases principales son:

- **Grafo**: administra nodos y conexiones.
- **Nodo**: representa cada ubicación del mapa.
- **BFS y DFS**: implementan los algoritmos de búsqueda.
- **GraphPanel**: permite la visualización gráfica.
- **Persistence** (**Loader y Saver**): guarda y carga el grafo desde archivo.

Esta organización separa la lógica del programa, la visualización y la persistencia de datos, facilitando el mantenimiento y escalabilidad del sistema.

4.4.Estructura de carpetas





4.5.Funcionamiento del sistema

El sistema permite crear nodos en el mapa, conectarlos entre sí y posteriormente seleccionar un nodo de inicio y uno destino. Una vez definidos, el usuario puede elegir entre BFS o DFS para buscar una ruta.

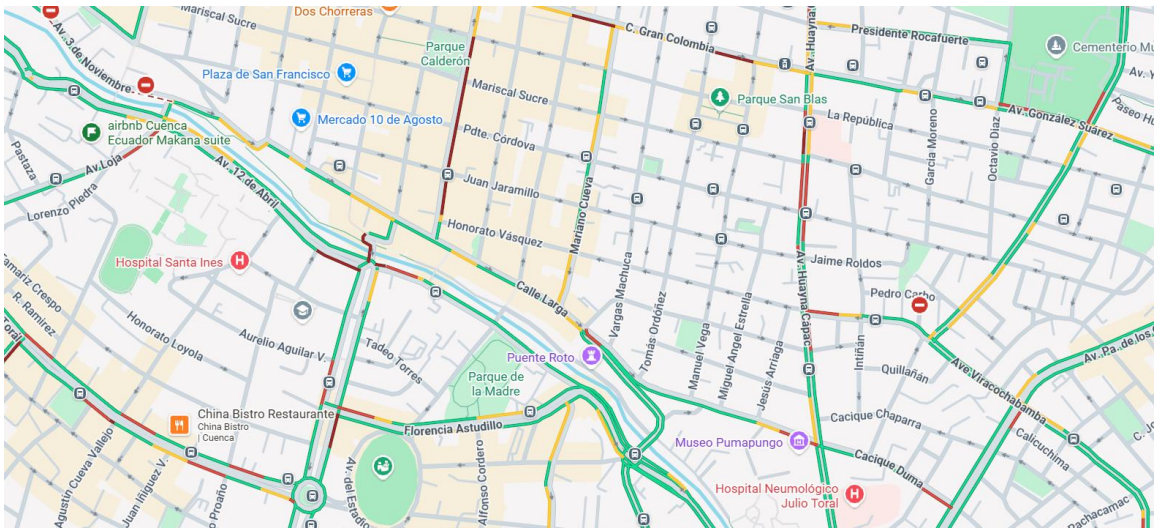
Durante la ejecución:

1. Se carga el grafo desde un archivo.
2. El usuario interactúa con el mapa.
3. El algoritmo recorre los nodos.
4. Se visualiza la exploración.
5. Se muestra la ruta final encontrada.

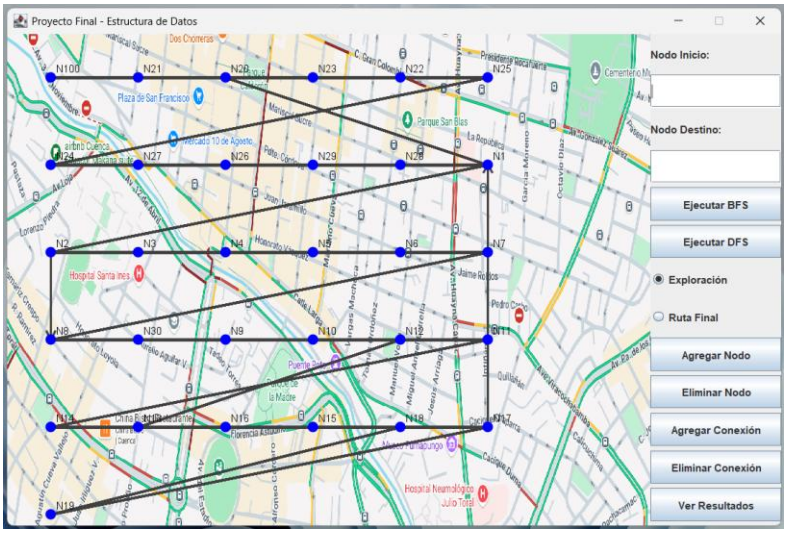
Además, cada vez que se agrega un nodo o conexión, estos quedan guardados permanentemente en un archivo, por lo que el mapa no se pierde al cerrar el programa.

4.6. Capturas de interfaz de mapas

1. **Mapa Base**



2. **Mapa con nodos**



4.7. Ejemplo de algoritmo comentado

```java
public class DFS {

    // Metodo principal que el sistema llama para buscar la ruta
    public static List<Nodo> buscarRuta(Grafo grafo, Nodo inicio, Nodo destino, List<Nodo> visitadosOrden){

        // Si el nodo inicio o destino estan bloqueados, no existe camino
        if (inicio.isBloqueado() || destino.isBloqueado()) {
            return new ArrayList<>();
        }

        // Conjunto para almacenar nodos ya visitados
        Set<Nodo> visitados = new HashSet<>();

        // Mapa que guarda quien fue el nodo anterior
        Map<Nodo, Nodo> precursor = new HashMap<>();

        // Llamada al metodo DFS recursivo
        boolean encontrado = dfsRecursivo(inicio, destino, visitados, precursor, visitadosOrden);

        // Si se encontró el destino se reconstruye la ruta
        if (encontrado) {
            return reconstruirRuta(precursor, inicio, destino);
        }

        // Si no se encontró camino se retorna lista vacia
        return new ArrayList<>();
    }

    // METODO RECURSIVO
    private static boolean dfsRecursivo(
        Nodo actual,
        Nodo destino,
        Set<Nodo> visitados,
        Map<Nodo, Nodo> precursor,
        List<Nodo> visitadosOrden
    )  {
        // Si el nodo esta bloqueado no se puede pasar
        if (actual.isBloqueado()) return false;

        // Marcar nodo como visitado
        visitados.add(actual);

        // Guardar el orden de exploración
        visitadosOrden.add(actual);

        // Si llegamos al destino terminamos la busqueda
        if (actual.equals(destino)) {
            return true;
        }

        // Recorrer todos los vecinos del nodo actual
        for (Nodo.Conexion c : actual.getConexiones()) {
            Nodo vecino = c.destino;

            // Solo visitar nodos no visitados y no bloqueados
            if (!visitados.contains(vecino) && !vecino.isBloqueado()) {

                //Guardamos desde que nodo llegamos
                precursor.put(vecino, actual);

                // Llamada recursiva
                boolean encontrado = dfsRecursivo(
                        vecino,
                        destino,
                        visitados,
                        precursor,
                        visitadosOrden
                );

                // Si alguna rama encontró el destino, terminamos
                if (encontrado) {
                    return true;
                }
            }
        }
        //Si ninugn vecino llevó al destino
        return false;
    }


    // RECONSTRUCCIÓN DE LA RUTA
    private static List<Nodo> reconstruirRuta(Map<Nodo,Nodo> predecesor, Nodo inicio, Nodo destino) {

        List<Nodo> ruta = new ArrayList<>();
        Nodo actual = destino;

        // Retrocede desde el destino hasta el inicio
        while (actual != null) {
            ruta.add(actual);
            actual = predecesor.get(actual);
        }

        // Invertir la lista para obtener inicio → destino
        Collections.reverse(ruta);

        // Validar que realmente empieza en el nodo inicial
        if (!ruta.isEmpty() && ruta.get(index: 0).equals(inicio)) {
            return ruta;
        }
        return new ArrayList<>();
    }
}
```

El algoritmo DFS funciona explorando el grafo en profundidad:

1. Inicia desde el nodo de origen.
2. Marca el nodo como visitado para evitar ciclos infinitos.
3. Visita uno de sus vecinos.
4. Continúa avanzando por ese camino hasta no poder seguir.
5. Si el camino no llega al destino, el algoritmo retrocede (backtracking) y prueba otro vecino.
6. Cuando el destino es encontrado, se reconstruye la ruta utilizando el mapa de predecesores.

Además, el sistema guarda el orden de exploración, lo que permite visualizar en la interfaz cómo el algoritmo recorrió el mapa paso a paso. Este comportamiento simula la exploración de un laberinto: el algoritmo sigue un camino hasta el final y, si no encuentra la salida, regresa y prueba otra dirección.

5. Conclusiones

1. **Janelly Dayanna Chacha Vélez:**
   Como conclusión algoritmo BFS resulta mas adecuado cuando se necesita encontrar una ruta mas corta entre dos nodos, ya que lo hace por niveles, sin embargo, el DFS es más rápido en exploración y consume menos memoria pero de igual manera no garantiza el camino optimo.
2. **Kelly Valeria Guamán León:**
   Se compararon los algoritmos BFS y DFS en grafos, considerando su recorrido, eficiencia y uso de recursos. BFS explora por niveles y garantiza el camino más corto en grafos no ponderados, pero usa más memoria. DFS profundiza en el grafo, consume menos memoria y no asegura la ruta más corta.
   En conclusión, BFS es mejor para encontrar el camino óptimo, mientras que DFS resulta útil para exploraciones profundas. La elección depende del objetivo y los recursos disponibles.

6. Recomendaciones y aplicaciones futuras

Se recomienda mejorar el sistema incorporando grafos ponderados y algoritmos como Dijkstra. Esto permitiría calcular rutas más eficientes considerando distancias reales.

1. Aplicaciones posibles:
2. Videojuegos
3. Sistemas GPS
4. Redes de comunicación
5. Planificación del transporte publico

7. Links

1. **Enlace GitHub:**
   https://github.com/Dayanna-01/icc-est-u2-ProyectoFinal
2. **Enlace Video:**
   https://www.youtube.com/watch?v=pNWupA1IAqw

8. Bibliografía:

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.
- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data Structures and Algorithms in Java* (6th ed.). Wiley.
- Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.
- Weiss, M. A. (2013). *Data Structures and Algorithm Analysis in Java* (3rd ed.). Pearson.
- Horowitz, E., Sahni, S., & Anderson-Freed, S. (2008). *Fundamentals of Data Structures in C++* (2nd ed.). Silicon Press.