ESCUELA POLITÉCNICA NACIONAL

FACULTAD DE INGENIERÍA DE SISTEMAS

ESTRUCTURAS Y ALGORITMOS II

GRUPO: No. 3

DEBER: Proyecto Bimestral

Alumnos:

Huilca Villagómez Fernando Eliceo

Quisilema Flores Juan Mateo

Simbaña Guarnizo Mateo Nicolás

PROFESORA: Dra. María Pérez.

FECHA DE ENTREGA: 23-11-2024

Deber de:

Proyecto primer bimestre: Defensa pública en el horario de clase

Tema: Proyecto bimestral

Resolución de pequeños problemas de la vida real, aplicando los tópicos vistos hasta la fecha en el periodo 2024B.

Objetivos profesor:

- Reforzar los temas estudiados en clase, permitiéndoles trabajar con todos los algoritmos vistos en la solución de problemas del entorno.
- Potenciar el trabajo en equipo y la modalidad colaborativa así como incentivar la investigación formativa del estudiante.
- Aprender a expresar, tanto de forma oral como escrita conocimientos, procedimientos, resultados e ideas en su campo.
- Que el alumno desarrolle habilidades de abstracción y capacidad para planificar y organizar, así como también potenciar la capacidad de análisis y de síntesis sobre algoritmos.
- Afianzar en qué consiste el diseño y análisis de algoritmos, así como las etapas de diseño, análisis y verificación de los algoritmos y la aplicación de las distintas técnicas existentes para abordar de manera adecuada la solución para garantizar el buen desempeño del mismo, en la solución de un problema específico.
- Aplicar técnicas algorítmicas, para dar solución a problemas de la vida real.
- Definir las etapas de Diseño, Análisis y Verificación de algoritmos y conocer su importancia.
- Encontrar la manera de describir la solución a un problema para ser traducido posteriormente, a un lenguaje comercial sencillamente.
- Desarrollar habilidades y destrezas en programación

Objetivos:

- Reconocer la importancia de los grafos y el algoritmo DFS mediante su aplicación en la solución de un problema práctico de la vida real.
- Implementar una aplicación que simule un laberinto e identifique la ruta desde la entrada hasta la salida, a través de la IDE IntelliJ, utilizando el lenguaje de programación Java, con la finalidad de aplicar los conocimientos adquiridos.
- Analizar el coste computacional del programa desarrollado con el propósito de medir su eficiencia en la resolución de problemas similares.

Marco teórico:

Lista simple

- **Definición**

Una lista es un tipo de estructura lineal y dinámica de datos. Lineal porque a cada elemento le puede seguir sólo otro elemento; dinámica porque se puede manejar la memoria de manera flexible, sin necesidad de reservar espacio con antelación. [1]

Una lista simplemente ligada constituye una colección de elementos llamados nodos. El orden entre estos se establece por medio de punteros; es decir, direcciones o referencias a otros nodos. [1]

- **Estructura**

Un nodo consta de dos partes:

1. Un campo **INFORMACIÓN** que será el tipo de los datos que se requiera almacenar en la lista. [1]
2. Un campo **LIGA**, de tipo puntero, que se utiliza para establecer la liga o el enlace con otro nodo de la lista. Si el nodo fuera el último de la lista, este campo tendrá el valor de null, es decir, vacío. [1]

- **Operaciones**

Las operaciones que pueden efectuarse en una lista simple ligada son:

1. Recorrido de la lista
2. Inserción de un elemento
3. Borrado de un elemento
4. Búsqueda de un elemento

Lista simplemente enlazada.

Pila

- **Definición**

Una pila es una colección de objetos que se insertan y eliminan de acuerdo con el principio de último en entrar, primero en salir (LIFO). Un usuario puede insertar objetos en una pila en cualquier momento, pero solo puede acceder o eliminar el objeto insertado más recientemente que quede (en la llamada "parte superior" de la pila).

El nombre "pila" se deriva de la metáfora de una pila de platos en un dispensador de platos de cafetería con resorte. En este caso, las operaciones fundamentales implican el "empuje" y el "estallido" de placas en la pila. Cuando necesitamos un nuevo plato del dispensador, "sacamos" la placa superior de la pila, y cuando agregamos un plato, lo "empujamos" hacia abajo en la pila para convertirse en la nueva placa superior. [2]

- **Estructura**

Una pila contiene los siguientes componentes:

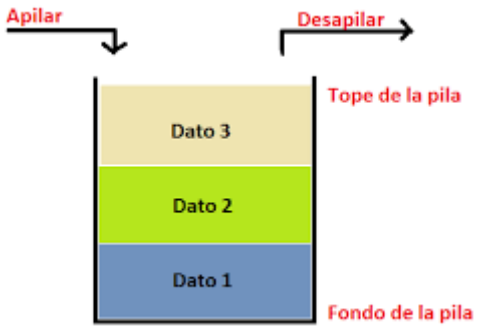
1. **Tope:** El elemento superior de la pila. [2]
2. **Tamaño:** El número de elementos en la pila. [2]

- **Operaciones**

Las pilas son la más simple de todas las estructuras de datos, pero también se encuentran entre las más importantes, ya que se utilizan en una gran cantidad de aplicaciones diferentes y como herramienta para muchas estructuras de datos y algoritmos más sofisticados. [2]

Formalmente, una pila es un tipo de datos abstractos (ADT) que admite las dos operaciones siguientes:

1. **push(e):** Agrega el elemento e a la parte superior de la pila. [2]
2. **pop( ):** Elimina y devuelve el elemento superior de la pila (o null si la pila está vacía). [2]



Grafos

- **Definición**

Un grafo es una estructura de datos matemática que consta de un conjunto de vértices (también llamados nodos) y un conjunto de aristas (también llamadas bordes) que conectan estos vértices. [3]

- **Componentes**

1. **Nodos (vértices):** Son elementos individuales en el grafo. Cada vértice puede representar entidades como personas, lugares, o cualquier objeto que se esté modelando. [3]
2. **Relaciones (aristas):** Son las conexiones entre los vértices. Estas aristas pueden ser direccionales o no direccionales, dependiendo del tipo de relación que se quiera representar. [3]

- **Tipos de grafos**

1. **Grafos dirigidos:** Cada arista tiene una dirección asociada. Esto significa que la relación entre dos vértices es unidireccional, y se representa mediante una flecha que indica la dirección de la conexión. La arista va desde un vértice inicial (fuente) hacia un vértice final (destino). [4]
2. **Grafos no dirigidos:** Las aristas no tienen dirección. Esto significa que la relación entre dos vértices es bidireccional y simétrica. La conexión entre dos nodos no tiene una orientación específica, y se representa mediante una línea que conecta los vértices. [4]
3. **Grafos ponderados:** Se dice que estos grafos están etiquetados cuando sus aristas contienen datos. En particular se dice que un grafo tiene peso si cada arista tiene un valor numérico no negativo que le proporciona condiciones de peso o longitud. [5]

- **Recorrido en un grafo**

El recorrido en un grafo es el proceso de visitar sus nodos (o vértices) siguiendo las conexiones definidas por sus aristas. Este proceso puede realizarse para buscar información, analizar estructuras o resolver problemas relacionados con el grafo.

- **Importancia de los recorridos en los grafos**

Es importante porque permite comprender las relaciones entre los vértices y las interacciones dentro del grafo. Los recorridos, como el de la ruta más corta, son útiles para realizar cálculos importantes.

En redes sociales, los grafos ayudan a analizar las interacciones entre usuarios. Cada usuario se representa como un nodo, y sus conexiones (amigos o seguidores) como las aristas. Al recorrer estos nodos, se puede simular cómo se propaga la información, cómo se forman comunidades o cómo se difunden tendencias.

Por ejemplo, cuando un nodo se activa (un usuario recibe una noticia), se puede observar cómo esa información se distribuye a través de la red.

Recorrido en profundidad (DFS)

El recorrido explora el grafo siguiendo un camino hasta llegar al final, y retrocede cuando no puede avanzar más. Utiliza una pila como estructura de datos para almacenar los nodos pendientes por explorar, lo que permite regresar y continuar la exploración desde puntos no visitados. [6]

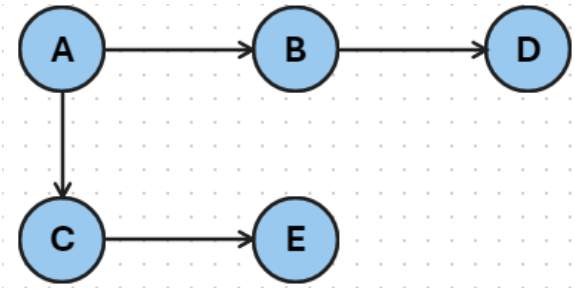
Se usa en una amplia variedad de aplicaciones, desde juegos hasta la resolución de problemas complejos en inteligencia artificial. [6]

- **Procedimiento**

El procedimiento es el siguiente:

1. Se inicializa una pila vacía y se empuja el nodo inicial.
2. Mientras la pila no esté vacía, se extrae el nodo de la cima y se verifica si ha sido visitado.
3. Si no ha sido visitado, se marca como visitado y se empujan sus vecinos no visitados a la pila.

- **Ejemplo**



Si se comienza el DFS en el nodo A, el algoritmo seguirá este recorrido:

1. Se inicia en el nodo A y se marca como visitado. A se apila en la estructura de la pila.
2. Se mueve al siguiente nodo adyacente no visitado (en este caso, B), se marca como visitado y se apila.
3. Desde B, se mueve al nodo adyacente no visitado (D), lo marca como visitado y lo apila.
4. Desde D, como no hay más nodos adyacentes no visitados, el algoritmo desapila D y retrocede al nodo B.
5. Desde B, no hay más nodos por explorar, así que el algoritmo desapila B y regresa a A.
6. Desde A, se mueve al siguiente nodo adyacente no visitado, que es C, lo marca como visitado y lo apila.
7. Desde C, se mueve al nodo adyacente no visitado E, lo marca como visitado y lo apila.
8. Como no hay más nodos adyacentes a E, el algoritmo desapila E, luego desapila C, y finalmente desapila A, completando el recorrido.

De modo que, el orden de visita de los nodos sería: A → B → D → C → E

- **Aplicaciones**

El algoritmo DFS tiene múltiples aplicaciones en problemas del mundo real, especialmente en escenarios donde es necesario explorar detalladamente caminos o estructuras. Algunas de sus aplicaciones más comunes incluyen:

1. **Resolución de laberintos:** Se utiliza para encontrar un camino desde la entrada hasta la salida de un laberinto o para verificar si existe una salida.
2. **Detección de ciclos en grafos:** En redes sociales, transacciones financieras o sistemas de dependencias, DFS ayuda a identificar ciclos que puedan representar problemas como bucles infinitos o inconsistencias.
3. **Análisis de conectividad:** Determina componentes conexas en redes (como redes de computadoras, carreteras o infraestructura de energía).
4. **Planificación de tareas:** En sistemas de dependencias, como proyectos con tareas interdependientes, se utiliza DFS para determinar un orden de ejecución válido.
5. **Generación y análisis de mapas:** Ayuda a recorrer y analizar mapas geográficos, sistemas de navegación o áreas en videojuegos.
6. **Solución de juegos de lógica:** Se aplica en rompecabezas o problemas como el Sudoku, donde es necesario explorar exhaustivamente las combinaciones posibles.
7. **Búsqueda en árboles de decisión:** Se utiliza para explorar todos los posibles resultados en problemas como el ajedrez, toma de decisiones empresariales o inteligencia artificial.

Recursividad

Se llama recursividad a un proceso mediante el que una función se llama a sí misma de forma repetida, hasta que se satisface alguna determinada condición. El proceso se utiliza para computaciones repetidas en las que cada acción se determina mediante un resultado anterior. Se pueden escribir de esta forma muchos problemas iterativos. [7]

Se deben satisfacer dos condiciones para que se pueda resolver un problema recursivamente:

1. El problema se debe escribir en forma recursiva. [7]
2. La sentencia del problema debe incluir una condición de fin. [7]

Dato tipo genérico

Un *tipo genérico* es un elemento de programación único que se adapta para ejecutar la misma funcionalidad para distintos tipos de datos. Cuando se define una clase o un procedimiento genérico, no es necesario definir una versión independiente para cada tipo de datos para el que quiera ejecutar esa funcionalidad. [8]

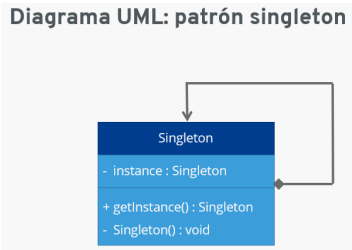
Al definir un tipo genérico, este se parametriza con uno o más tipos de datos. Esto permite que el código que lo utiliza se adapte a los requisitos específicos de los tipos de datos. Un mismo código puede declarar múltiples elementos de programación a partir del tipo genérico, cada uno operando con un conjunto diferente de datos. Sin embargo, todos los elementos generados mantienen la misma lógica, independientemente de los tipos de datos utilizados. [8]

Patron Singleton

El propósito de este patrón es evitar que sea creado más de un objeto por clase. Esto se logra creando el objeto deseado en una clase y recuperándolo como una instancia estática. [9]

Para asegurarse de que permanezca con una sola instancia única, se debe impedir que los usuarios creen nuevas instancias. Esto se logra mediante el constructor, declarando el patrón

como “privado”. Esto significa que sólo el código en el singleton puede instanciar el singleton en sí mismo. [9]



Objetivo del proyecto:

Desarrollar las técnicas algorítmicas necesarias, para dar solución a un problema planteado y que, una vez resuelto, hace la defensa oral.

Desarrollo de la práctica:

Clase Nodo

La clase Nodo representa un vértice genérico en un grafo, con sus propiedades y métodos diseñados para gestionar información, relaciones (adyacencias) y estados de visita.

Los atributos principales son infoNodo, que almacena información genérica asociada al nodo, nodosAdyacentes, que guarda una lista de nodos directamente conectados a este, y visitado, un booleano que indica si el nodo ha sido visitado. Esto último es especialmente útil en algoritmos de recorrido como DFS, donde es necesario marcar los nodos ya explorados para evitar ciclos.

La clase cuenta con dos constructores. Uno permite inicializar un nodo con información (infoNodo), mientras que el otro crea un nodo vacío. Ambos inicializan la lista de nodos adyacentes y configuran el estado de visita en false, asegurando que el nodo esté preparado para interactuar en el contexto del grafo desde el momento de su creación. Esto se muestra en la siguiente imagen.

```
public class Nodo<T> {  51 usages  FernandoHuilca *
    private T infoNodo;  4 usages
    private ArrayList<Nodo<T>> nodosAdyacentes;  7 usages
    private boolean visitado;  4 usages


    public Nodo(T infoNodo) {  11 usages  FernandoHuilca
        this.infoNodo = infoNodo;
        nodosAdyacentes = new ArrayList<>();
        visitado = false;
    }

    public Nodo() {  no usages  FernandoHuilca
        this.infoNodo = null;
        nodosAdyacentes = new ArrayList<>();
        visitado = false;
    }
```

Los métodos de esta clase se centran en tres aspectos principales: la manipulación de la información contenida en el nodo, la gestión de sus nodos adyacentes y el control del estado de visita. Por ejemplo, los métodos getInfoNodo y setInfoNodo permiten acceder y modificar la información del nodo. Para gestionar las conexiones entre nodos, se ofrecen métodos como agregarAdyacente, que añade un nuevo nodo a la lista de adyacencias, o getNodoAdyacente, que permite acceder a nodos específicos de la lista. Finalmente, los métodos setEstadoVisitado y estaVisitado gestionan el estado de visita del nodo, lo que resulta importante para evitar recorrer innecesariamente el mismo nodo en algoritmos como DFS. Esto se indica en la siguiente imagen.

```
public T getInfoNodo() {  8 usages  FernandoHuilca
    return infoNodo;
}

public void setInfoNodo(T infoNodo) {  1 usage  FernandoHuilca
    this.infoNodo = infoNodo;
}

public ArrayList<Nodo<T>> getNodosAdyacentes() {  13 usages  FernandoHuilca
    return nodosAdyacentes;
}

public void setNodosAdyacentes(ArrayList<Nodo<T>> nodosAdyacentes) {
    this.nodosAdyacentes = nodosAdyacentes;
}

public boolean agregarAdyacente(Nodo<T> nodoDestino) {  3 usages  Fernand
    nodosAdyacentes.add(nodoDestino);
    return true;
}

public void setNodoAdyacente(int numNodoDeLaLista, Nodo<T> nuevoNodo) {
nodosAdyacentes.set(numNodoDeLaLista,nuevoNodo);
}
public boolean agregarAdyacente(Nodo<T> nodoDestino) {  3 usages  Fernando
    nodosAdyacentes.add(nodoDestino);
    return true;
}

public void setNodoAdyacente(int numNodoDeLaLista, Nodo<T> nuevoNodo) {
nodosAdyacentes.set(numNodoDeLaLista,nuevoNodo);
}

public Nodo<T> getNodoAdyacente(int numLisNodosAdyacentes) {  14 usages
    return nodosAdyacentes.get(numLisNodosAdyacentes);
}

public void setEstadoVisitado(boolean valor) {  1 usage  FernandoHuilca
    visitado =valor;
}

public boolean estaVisitado() {  3 usages  FernandoHuilca
    return visitado;
}
```

Clase Grafo

La clase abstracta Grafo proporciona una estructura base para representar grafos.

Los atributos principales de la clase incluyen una lista de nodos (nodosDelGrafo), contadores de nodos y aristas, y una variable para almacenar un camino de salida (salida1).

Por otro lado, el constructor inicializa los atributos clave del grafo, preparando la lista de nodos y configurando los contadores en cero, tal como se visualiza en la siguiente imagen.

```
public abstract class Grafo {  1 usage  1 inheritor
    protected ArrayList<Nodo<String>> nodosDelGrafo;
    protected int contadorDeNodos;  3 usages
    protected int contadorDeAristas;  3 usages
    private String salida1 = null;  4 usages

    public Grafo() {  1 usage
        nodosDelGrafo = new ArrayList<>();
        contadorDeNodos = 0;
        contadorDeAristas = 0;
    }
```

Entre los métodos, se destaca agregarNodo, que permite añadir un nuevo nodo a la lista, incrementando automáticamente el contador de nodos. Este método devuelve siempre true, asumiendo que la adición será exitosa, aunque una posible mejora sería incluir validaciones para evitar nodos duplicados.

El método abstracto agregarAristaANodo es clave, ya que delega la implementación de cómo se conectan los nodos a las clases derivadas. Esto es importante porque diferentes tipos de grafos (como dirigidos o no dirigidos) requieren reglas específicas para agregar aristas.

```
public boolean agregarNodo(Nodo<String> nuevoNodo) {  88 usages
    nodosDelGrafo.add(nuevoNodo);
    contadorDeNodos++;
    return true;
}

public abstract boolean agregarAristaANodo(int nodoOrigen, int nodoDestino);

public ArrayList<Nodo<String>> getNodosDelGrafo() {  no usages
    return nodosDelGrafo;
}
```

La clase incluye un método especializado, getTrazaRecorridoAProfundidad, para realizar un recorrido en profundidad (DFS) desde un nodo inicial. Este método emplea una pila para mantener el rastro de los nodos visitados y una lista para registrar los pasos del recorrido. La implementación detallada del DFS asegura que todos los nodos conectados al inicial sean explorados. Además, se verifica si quedan nodos no visitados en el grafo para cubrir componentes desconectados, como se muestra en la siguiente imagen.

```java
public ArrayList<String> getTrazaRecorridoAProfundidad(int numNodoInicial) {  1 usage
    Nodo<String> nodoInicial = this.getNodo(numNodoInicial);
    ArrayList<Nodo<String>> visitados = new ArrayList<>();
    Pila<Nodo> pila = new Pila<>();
    ArrayList<String> trazaRecorridoAProfundidad = new ArrayList<>();
    recorridoDFS(nodoInicial, visitados, pila, trazaRecorridoAProfundidad);
    return trazaRecorridoAProfundidad;
}
private void recorridoDFS(Nodo<String> nodoInicial, ArrayList<Nodo<String>> visitados, Pila<Nodo> pila,
    if (nodoInicial == null) return;  // Verifica que el nodo no sea nulo
    pila.add(nodoInicial);
    visitados.add(nodoInicial);


    String aux = " ";
    for (int i = 0; i < pila.getNumDeDatos(); i++) {
        aux = aux + " " + pila.getDato(i).getInfoNodo();
    }
    trazaRecorridoAProfundidad.add(aux);

    // Recorrer los nodos adyacentes (nodos apuntados)
    for (Nodo<String> nodoApuntado : nodoInicial.getNodosAdyacentes()) {
        // Si el nodo no ha sido visitado, se realiza la llamada recursiva
        if (!visitados.contains(nodoApuntado)) {
            recorridoDFS(nodoApuntado, visitados, pila, trazaRecorridoAProfundidad);
        }
    }
    pila.eliminarDato();

    String aux1 = " ";
    for (int i = 0; i < pila.getNumDeDatos(); i++) {
        aux1 = aux1 + " " + pila.getDato(i).getInfoNodo();
    }
    trazaRecorridoAProfundidad.add(aux1);

    // Verificar si quedan nodos no visitados en el grafo
    if (pila.estaVacia() && !yaRecorriTodosLosNodos()) {
        Nodo<String> siguienteNodoNoVisitado = this.getNoVisitado();
        if (siguienteNodoNoVisitado != null) {
            recorridoDFS(siguienteNodoNoVisitado, visitados, pila, trazaRecorridoAProfundidad);
        }
    }

}
```

Otro método destacado es getCaminoDeSalida, que busca un camino específico entre dos nodos, como una entrada y una salida del laberinto. Este algoritmo también utiliza DFS, pero adapta su lógica para detenerse al encontrar el nodo de destino. Si se encuentra un camino, se registra como una cadena de texto que muestra el recorrido completo desde la entrada hasta la salida, como se visualiza en la siguiente imagen.

```java
public String getCaminoDeSalida(int numNodoEntrada, int numNodoSalida) {  1 usage
    Nodo<String> nodoEntrada = this.getNodo(numNodoEntrada);
    Nodo<String> nodoSalida = this.getNodo(numNodoSalida);
    ArrayList<Nodo<String>> visitados = new ArrayList<>();
    Pila<Nodo> pila = new Pila<>();
    salida1 = null;
    caminoDeSalida(nodoEntrada, nodoSalida, visitados, pila);
    return salida1;
}
private void caminoDeSalida(Nodo<String> nodoEntrada, Nodo<String> nodoSalida, ArrayList<Nodo<String>> visitados,
    if (nodoEntrada == nodoSalida) {
        salida1 = nodoEntrada.getInfoNodo();
        return;
    }
    pila.add(nodoEntrada);
    visitados.add(nodoEntrada);


    // Recorrer los nodos adyacentes (nodos apuntados)
    for (Nodo<String> nodoApuntado : nodoEntrada.getNodosAdyacentes()) {

        if (nodoApuntado == nodoSalida) {
            String aux = " ";
            for (int i = 0; i < pila.getNumDeDatos(); i++) {
                aux = aux + " " + pila.getDato(i).getInfoNodo();
            }
            salida1 = aux + " " + nodoSalida.getInfoNodo();
            return;

        }
         if (!visitados.contains(nodoApuntado)) {
             caminoDeSalida(nodoApuntado, nodoSalida, visitados, pila);
         }
    }

    pila.eliminarDato();


    // Verificar si quedan nodos no visitados en el grafo
    if (pila.estaVacia() && !yaRecorriTodosLosNodos()) {
        Nodo<String> siguienteNodoNoVisitado = this.getNoVisitado();
        if (siguienteNodoNoVisitado != null) {
            caminoDeSalida(siguienteNodoNoVisitado, nodoSalida, visitados, pila);
        }
    }
}
```

Finalmente, la clase incluye métodos auxiliares para verificar si todos los nodos han sido visitados (yaRecorriTodosLosNodos) y para obtener el primer nodo no visitado (getNoVisitado), como se indica en la siguiente imagen.

```java
private boolean yaRecorriTodosLosNodos() {  2 usages
    // Verifica si todos los nodos del grafo han sido visitados
    for (Nodo<String> nodo : nodosDelGrafo) {
        if (!nodoHaSidoVisitado(nodo)) {
            return false;  // Si algún nodo no ha sido visitado, retornamos false
        }
    }
    return true;  // Si todos los nodos han sido visitados, retornamos true
}
public Nodo<String> getNoVisitado() {  2 usages  1 override
    for (Nodo<String> nodo : nodosDelGrafo) {
        if (!nodoHaSidoVisitado(nodo)) {
            return nodo;  // Retorna el primer nodo no visitado
        }
    }
    return null;  // Si todos los nodos han sido visitados, retorna null
}
```

Clase GrafoNoDirigido

La clase GrafoNoDirigido extiende la clase abstracta Grafo, implementando su comportamiento específicamente para grafos no dirigidos.

El constructor de la clase simplemente invoca el constructor de la clase base para inicializar los atributos heredados, como la lista de nodos y los contadores. Esto asegura que la estructura básica del grafo esté lista para su uso, como se indica en la siguiente imagen.

```java
public class GrafoNoDirigido extends Grafo {  8 usages

    public GrafoNoDirigido(){  2 usages  FernandoHuilca
        super();
    }
```

Por otra parte, el método sobrescrito agregarAristaANodo se encarga de añadir conexiones bidireccionales entre dos nodos. Para lograrlo, utiliza un método auxiliar llamado apuntarNodo, que añade un nodo destino como adyacente al nodo origen.

El método privado apuntarNodo simplifica el proceso de agregar un nodo como adyacente a otro. Utiliza la lista de nodos del grafo para acceder al nodo origen y llama al método agregarAdyacente del nodo, lo que permite establecer la relación entre los nodos.

Además, la clase redefine el método getNoVisitado para devolver el primer nodo que no haya sido marcado como visitado. Este método es particularmente útil en algoritmos de búsqueda, como el DFS, para explorar componentes del grafo que aún no han sido procesados. Si todos los nodos han sido visitados, el método retorna null, como se aprecia en la siguiente imagen.

```java
@Override  83 usages
public boolean agregarAristaANodo(int nodoOrigen, int nodoDestino) {
    if (apuntarNodo(nodoOrigen, nodoDestino) && apuntarNodo(nodoDestino,nodoOrigen)){
        contadorDeAristas++;
        return true;
    }
    return false;
}


private boolean apuntarNodo(int nodoOrigen, int nodoDestino) {  2 usages
    return nodosDelGrafo.get(nodoOrigen).agregarAdyacente(nodosDelGrafo.get(nodoDestino));
}

public Nodo<String> getNoVisitado() {  2 usages
    // Devuelve el primer nodo que no ha sido visitado
    for (Nodo<String> nodo : nodosDelGrafo) {
        if (nodo != null && !nodo.estaVisitado()) {
            return nodo;
        }
    }
    return null; // Si todos los nodos han sido visitados, retorna null
}
```

Clase GraphWorks

La clase GraphWorks actúa como un gestor centralizado para la manipulación y administración de grafos no dirigidos en el sistema. Implementa el patrón de diseño Singleton para asegurar que solo exista una instancia de la clase en toda la aplicación.

La clase tiene un atributo estático instance que almacena la única instancia de la clase y un atributo listGrafosNoDirigidos, que es una lista para almacenar instancias de GrafoNoDirigido, como se muestra en la siguiente imagen.

```java
public class GraphWorks {  14 usages
    // atributo único para el patron singleton
    private static GraphWorks instance;  3 usages
    //Otros atributos
    private ArrayList<GrafoNoDirigido> listGrafosNoDirigidos;


    //constructor privado
    private GraphWorks() {  1 usage
        this.listGrafosNoDirigidos = new ArrayList<>();
    }

    //Devolver la única instancia de la app
    public static GraphWorks getInstance() {  4 usages
        if (instance == null) {
            instance = new GraphWorks();
        }
        return instance;
    }
```

Entre las funcionalidades más importantes, GraphWorks permite agregar grafos y nodos a través de los métodos agregarGrafoNoDirigido y agregarNodoAlGrafo, como se indica a continuación.

```java
public boolean agregarGrafoNoDirigido(GrafoNoDirigido nuevoGrafoNoDirigido) {  4 usages
    if (listGrafosNoDirigidos.add(nuevoGrafoNoDirigido)) {
        return true;
    } else {
        return false;
    }
}

public boolean agregarNodoAlGrafo(int numeroDeLaListDeGrafos, Nodo<String> nuevoNodo) {
    if (listGrafosNoDirigidos.get(numeroDeLaListDeGrafos).agregarNodo(nuevoNodo)) {
        return true;
    }
    return false;
}
```

La clase también incluye métodos como getTrazaRecorridoAProfundidad, que permite visualizar un recorrido en profundidad (DFS) desde un nodo inicial en un grafo específico.

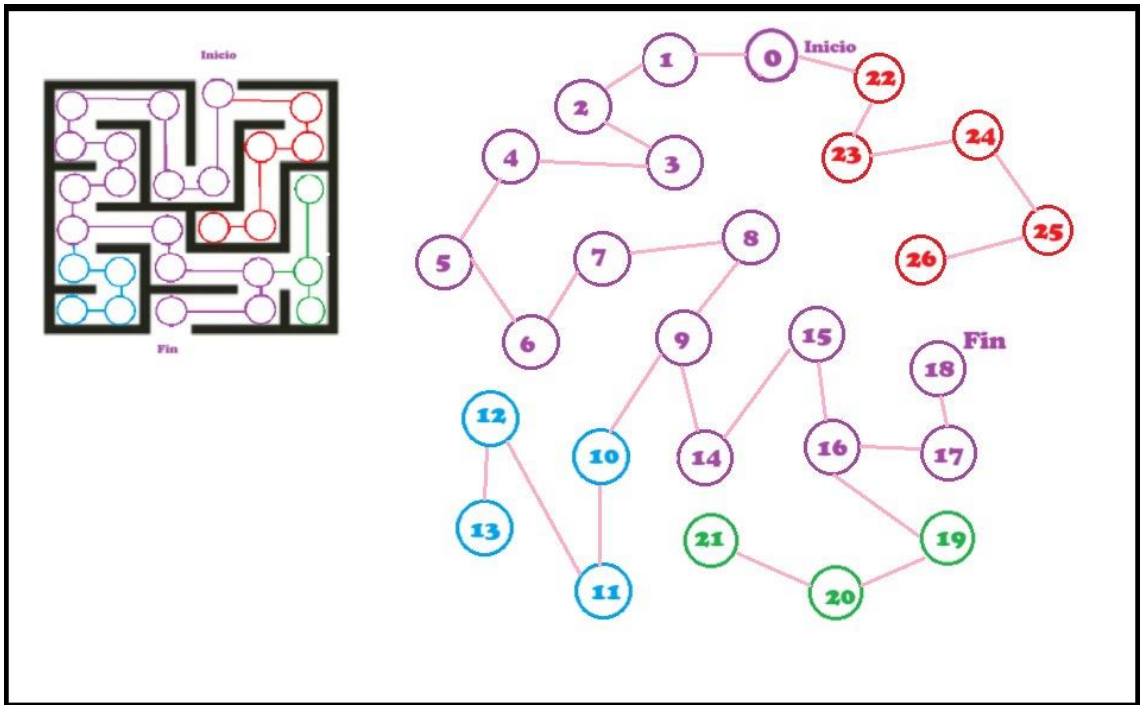
Además, GraphWorks proporciona estadísticas clave mediante los métodos getNumAristas y getNumNodos, que permiten obtener el número de aristas y nodos en un grafo. Para problemas más complejos, como encontrar un camino dentro de un grafo (por ejemplo, resolver un laberinto), el método getCaminoDeSalida calcula un recorrido desde un nodo de entrada hasta un nodo de salida, como se muestra en la siguiente imagen.

```java
public ArrayList<String> getTrazaRecorridoAProfundidad(int numGrafoGuardado, int nodoInicio) {  4 usages
        return listGrafosNoDirigidos.get(numGrafoGuardado).getTrazaRecorridoAProfundidad(nodoInicio);
}

public int getNumAristas(int numGrafoGuardado) {  4 usages
    return listGrafosNoDirigidos.get(numGrafoGuardado).getContadorDeAristas();
}

public int getNumNodos(int numGrafoGuardado) {  4 usages
    return listGrafosNoDirigidos.get(numGrafoGuardado).getContadorDeNodos();
}

public String getCaminoDeSalida(int numGrafoGuardado, int numNodoEntrada, int numNodoSalida) {  4 usages
    return listGrafosNoDirigidos.get(numGrafoGuardado).getCaminoDeSalida(numNodoEntrada, numNodoSalida);
}
```

Análisis de resultados:

Para verificar la funcionalidad del programa, se utilizaron dos grafos generados a partir de laberintos preexistentes, como se indica en la siguiente imagen.

El primer grafo se muestra a continuación.



Con el grafo generado a partir del laberinto, en el método LaberintoApplication se instanció la aplicación como graphWorks, se creó un grafo no dirigido y luego se almacenó dicho grafo en la aplicación. Esto se visualiza en la siguiente imagen.

```
GrafoNoDirigido grafoNoDirigido = new GrafoNoDirigido();
graphWorks.agregarGrafoNoDirigido(grafoNoDirigido);
```

Como siguiente paso, se agregaron los 27 nodos que contiene el grafo, como se muestra en la siguiente imagen.

```
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "0"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "1"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "2"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "3"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "4"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "5"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "6"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "7"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "8"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "9"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "10"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "11"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "12"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "13"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "14"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "15"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "16"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "17"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "18"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "19"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "20"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "21"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "22"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "23"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "24"));
```

A continuación, se agregaron las aristas y se establecieron las relaciones uno a uno con los vértices, siguiendo la estructura del grafo previamente mencionado. Es relevante destacar que los nodos se almacenaron en una lista indexada desde 0, lo que significa que el nodo 0 ocupará el índice 0, y los demás nodos se dispondrán de manera secuencial, tal como se muestra en la siguiente imagen.

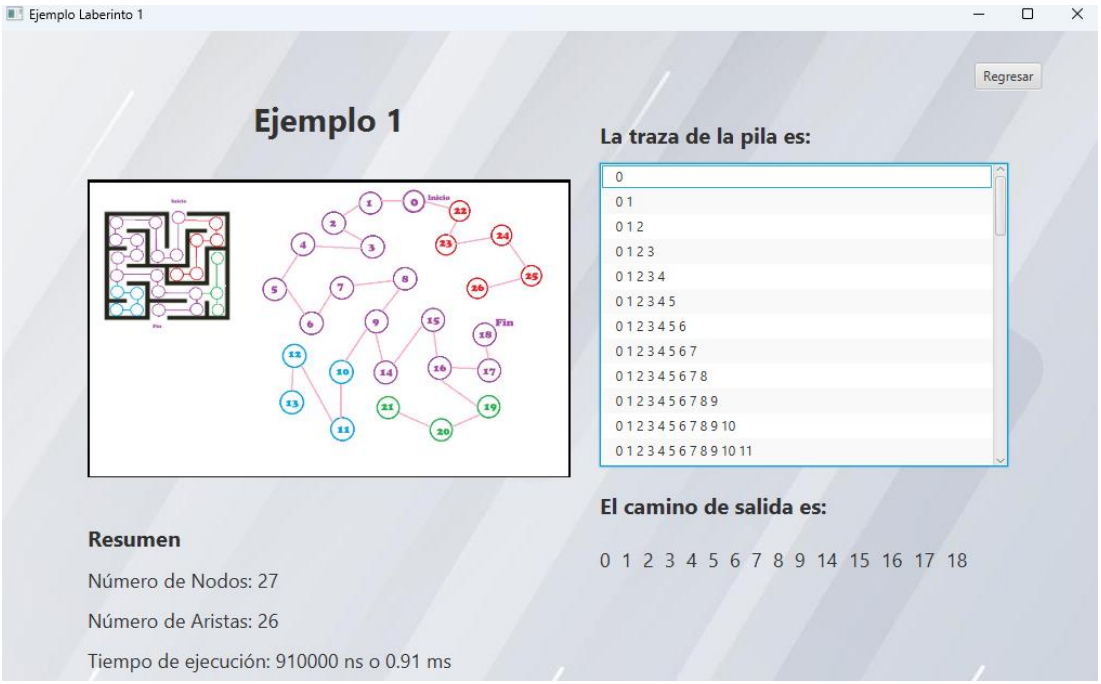
```
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "0"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "1"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "2"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "3"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "4"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "5"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "6"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "7"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "8"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "9"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "10"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "11"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "12"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "13"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "14"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "15"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "16"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "17"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "18"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "19"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "20"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "21"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "22"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "23"));
grafoNoDirigido.agregarNodo(new Nodo<>( infoNodo: "24"));
```

Una vez implementado completamente el grafo en la aplicación, en la clase EjemploLaberinto1Controller, se utilizaron diversos métodos para obtener información relevante sobre el grafo.

```
// Inicializar datos del resumen
numeroNodos.setText("Número de Nodos: " + graphWorks.getNumNodos( numGrafoGuardado: 0));
numeroAristas.setText("Número de Aristas: " + graphWorks.getNumAristas( numGrafoGuardado: 0));

ArrayList<String> auxTrazaPilaDFS = graphWorks.getTrazaRecorridoAProfundidad( numGrafoGuardado: 0,  nodoInicio: 0);

trazaPila.getItems().addAll(auxTrazaPilaDFS);


long inicio = System.nanoTime(); // Marca el inicio del tiempo
String caminoDeSalida = graphWorks.getCaminoDeSalida( numGrafoGuardado: 0,  numNodoEntrada: 0,  numNodoSalida: 18);
long fin = System.nanoTime(); // Marca el fin del tiempo
this.tiempoEjecucion.setText("Tiempo de ejecución: " + tiempoEjecucion + " ns o "
                             + (tiempoEjecucion / 1_000_000.0) + " ms");
```

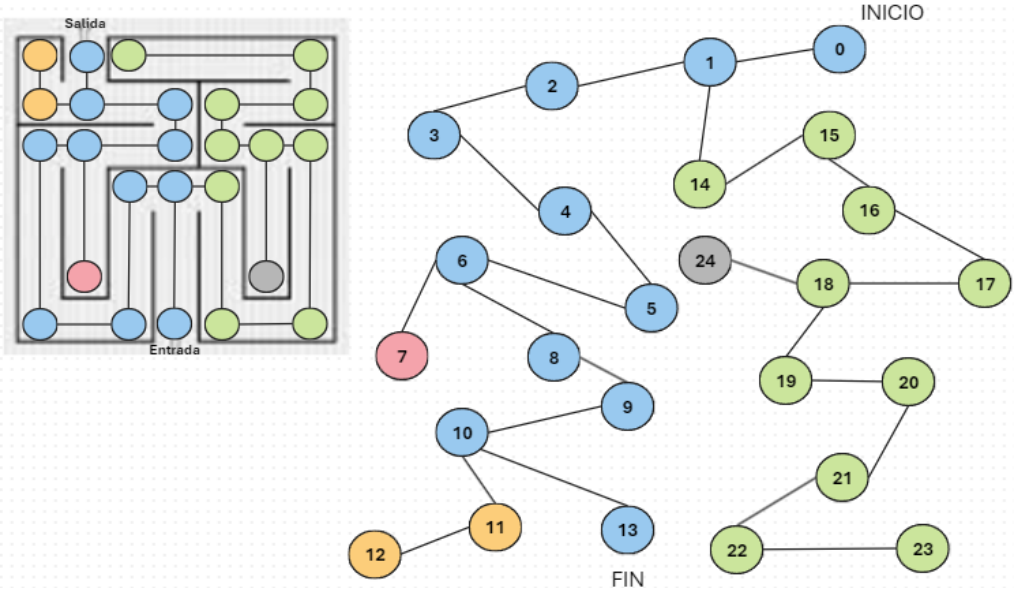
Resultado del primer grafo

En una interfaz gráfica, se mostraron el número de nodos y aristas del grafo. También se generó y visualizó la traza de la pila utilizada para recorrer todo el grafo, lo que permitió identificar el camino desde la entrada hasta la salida del laberinto. Finalmente, se imprimió el recorrido hacia la salida, cumpliendo con el objetivo principal del programa.



Por otro lado, el tiempo que tardó la aplicación en recorrer todo el grafo mediante DFS fue de 0,91 milisegundos.

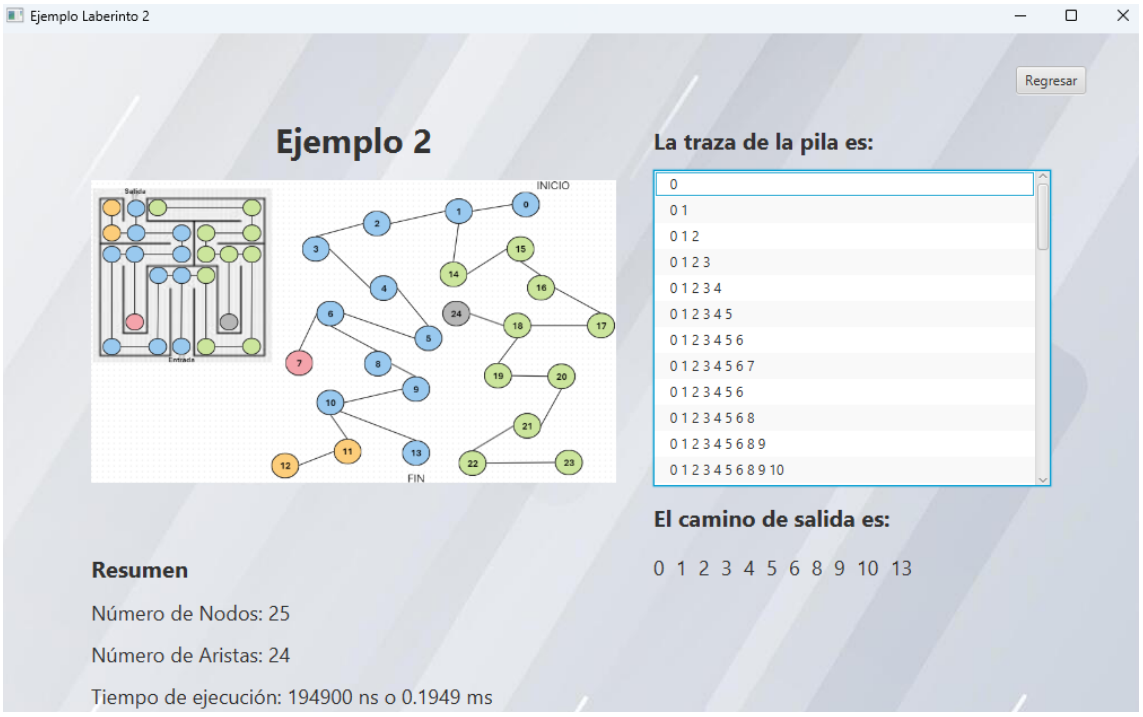
El segundo grafo usado dentro del programa es el siguiente.



El grafo generado a partir del laberinto fue implementado en la aplicación utilizando los mismos pasos seguidos para el primer grafo. Sin embargo, para este caso también se desarrolló una clase adicional llamada EjemploLaberinto2Controller, diseñada específicamente para mostrar la información relevante de este grafo.

Resultado del segundo grafo

En la siguiente imagen, el programa detalla la cantidad de nodos y aristas que conforman el grafo. Asimismo, se incluye la traza de la pila, la cual refleja el recorrido realizado para visitar todos los vértices del grafo. Finalmente, se destaca el camino identificado como una ruta para llegar a la salida del laberinto.



Además, es importante mencionar que el tiempo de ejecución del algoritmo fue de 0,19 milisegundos.

Cálculo de Big-O para el algoritmo de DFS implementado

El cálculo de la complejidad temporal del algoritmo DFS implementado para obtener la ruta desde la entrada hacia la salida del laberinto considera las operaciones realizadas y su relación con la cantidad de nodos (V) y aristas (E) del grafo.

- **Operaciones principales**

Recursión y visitas: Cada nodo es visitado una sola vez, lo que implica O(V) visitas.

Recorrido de adyacentes: Para cada nodo, se recorren sus nodos adyacentes, lo que en total equivale a recorrer todas las aristas del grafo una vez. Esto toma O(E) en grafos representados con listas de adyacencia.

- **Otras operaciones**

Verificar si un nodo ha sido visitado (visitados.contains()) tiene complejidad O(V) en el peor caso. Sin embargo, esta operación se realiza O(E) veces (por cada arista), lo que podría dar O(V·E) en implementaciones poco óptimas.

- **Casos adicionales**

Operaciones auxiliares como agregar y eliminar elementos de la pila son O(1) y no afectan significativamente la complejidad global.

Los métodos yaRecorriTodosLosNodos() y getNoVisitado() introducen una verificación adicional de O(V), pero como son llamados una sola vez al finalizar, no afectan la complejidad global.

Debido a que se recorre cada nodo y arista una única vez, en el peor caso, con un grafo completamente conectado, la complejidad es:

$$\boldsymbol{O(V + E)}$$

Conclusiones y recomendaciones

Conclusiones:

El proyecto mostró cómo los grafos y el algoritmo DFS se pueden usar para resolver problemas prácticos, como encontrar rutas en un laberinto. Esto resalta la utilidad de estas herramientas en áreas como videojuegos y sistemas de navegación, destacando la relevancia de la teoría de grafos en situaciones cotidianas.

Se logró implementar una aplicación funcional en IntelliJ utilizando Java, que permite simular un laberinto y determinar una ruta eficiente desde la entrada hasta la salida, aplicando de manera efectiva los conocimientos adquiridos en la construcción de grafos y algoritmos de recorrido.

El análisis de la complejidad del algoritmo DFS implementado para obtener la ruta desde la entrada hacia la salida del laberinto revela que su complejidad temporal es O(V+E), donde V es el número de nodos (vértices) y E es el número de aristas en el grafo. Esto significa que el tiempo de ejecución crece linealmente con el tamaño del grafo. En términos prácticos, este algoritmo es eficiente para grafos pequeños o medianos, pero puede enfrentar limitaciones en escenarios con grafos extremadamente grandes, especialmente si la memoria disponible es limitada.

Recomendaciones:

Se recomienda optimizar el algoritmo utilizando heurísticas, como A*, para reducir el número de nodos explorados y mejorar el rendimiento en laberintos grandes. Esto ayudará a encontrar el camino más rápido hacia la salida, ahorrando tiempo y memoria.

Aunque el proyecto resuelve el problema del laberinto, agregar una interfaz visual interactiva que muestre el recorrido en tiempo real podría ayudar a mejorar la comprensión y usabilidad del sistema, haciendo que los usuarios puedan seguir el algoritmo paso a paso.

Finalmente, para poder identificar el camino más corto dentro de un laberinto, se puede implementar otro algoritmo en la aplicación, como el algoritmo de Dijkstra o el algoritmo de Bellman-Ford, que son muy eficaces para este tipo de problemas en grafos ponderados.

Bibliografía:

[1] Data Structures and Algorithms in Java Michael T. Goodrich,Roberto Tamassia and Michael H. Goldwasser John Wiley & Sons, edit Sexta, 2014, ISBN: 978-1-118-77133-4

[2] Cairo Osvaldo, Guardati Silvia, Estructura de datos, ISBN: 970105908-5, Tercera Edición 2006. [En línea]. Disponible en: https://drive.google.com/file/d/0B_XimPSyUDLcM2ZtU3VCVHhLUUk/edit?pref=2&pli=1

[3] A. Castro, «saberpunto.com,» 24 de septiembre de 2024. [En línea]. Disponible en: https://saberpunto.com/programacion/que-es-el-recorrido-de-grafos-recorrido-en-profundidad-y-anchura/#Recorrido_en_Anchura_(BFS)

[4] L. Llamas, «Luis Llamas,» 22 Abril 2024. [En línea]. Disponible en: https://www.luisllamas.es/que-es-un-grafo/

[5] ApInEm, «ApInEm Web,» 22 de febrero de 2024. [En línea]. Disponible en: https://www.apinem.com/grafos-en-programacion/#3-1-grafos-dirigidos

[6] Khan Academy, «Khan Academy,» 18 de noviembre de 2014. [En línea]. Disponible en: https://es.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/representing-graphs

[7] J. F. Vázquez, "Capítulo 6: Tipos de Datos", *Facultad de Ciencias de la Computación, Universidad de Granada*. [En línea]. Disponible en: https://ccia.ugr.es/~jfv/ed1/c/cdrom/cap6/cap66.htm

[8] Microsoft, "Tipos genéricos en Visual Basic", *Microsoft Learn*. [En línea]. Disponible en: https://learn.microsoft.com/es-es/dotnet/visual-basic/programming-guide/language-features/data-types/generic-types

[9] IONOS, "Patrón Singleton", *Guía Digital de IONOS*. [[En línea]. Disponible en: https://www.ionos.com/es-us/digitalguide/paginas-web/desarrollo-web/patron-singleton/