

	VICERRECTORADO DOCENTE	Código: GUIA-PRL-001
	CONSEJO ACADÉMICO	Aprobación: 2016/04/06
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		

		FORMATO DE GUÍA DE PRÁCTICA DE LABORATORIO / TALLERES / CENTROS DE SIMULACIÓN – PARA DOCENTES	
CARRERA: CIENCIAS DE LA COMPUTACIÓN		ASIGNATURA: Estructura de Datos	
NRO. PRÁCTICA:	1	TÍTULO PRÁCTICA: Implementación de un Algoritmo para Encontrar la Ruta Óptima en un Laberinto Aplicando Programación Dinámica y Estructuras de Datos	
OBJETIVO: Desarrollar una aplicación que implemente algoritmos de búsqueda y optimización para encontrar la ruta óptima entre dos puntos en un laberinto, utilizando técnicas de programación dinámica y estructuras de datos lineales y no lineales, fomentando la eficiencia y el pensamiento algorítmico.			
Descripción del problema  El presente proyecto aborda el desafío de encontrar una <b>ruta óptima</b> desde un punto de inicio hasta un punto de destino dentro de un <b>laberinto bidimensional</b> , representado como una matriz de celdas transitables y no transitables.  El problema consiste en desarrollar una solución informática que permita al usuario configurar libremente el laberinto, seleccionar diferentes algoritmos de búsqueda y visualizar paso a paso cómo se resuelve el camino. Esto implica no solo implementar algoritmos eficientes, sino también diseñar una interfaz amigable, manejar estructuras de datos adecuadas y registrar métricas de rendimiento para comparar los resultados.			
Url Proyecto: <a href="https://github.com/Mely18Elizabeth/icc-est-Proyecto-final">https://github.com/Mely18Elizabeth/icc-est-Proyecto-final</a>			
ACTIVIDADES POR DESARROLLAR			
1. <b>BFS (Breadth-First Search):</b>  Es un algoritmo de búsqueda que recorre los nodos de un grafo (o laberinto) de manera <b>horizontal</b> , es decir, explora primero todos los vecinos de un nodo antes de avanzar a los siguientes niveles. Se implementa usando una <b>cola (FIFO)</b> y es ideal cuando se busca encontrar el <b>camino más corto</b> , siempre y cuando el grafo no esté ponderado. En el contexto de laberintos, BFS garantiza la solución óptima si todos los movimientos tienen el mismo costo. Su desventaja principal es el consumo de memoria, ya que debe almacenar muchos nodos en cada nivel de expansión.			
2. <b>DFS (Depth-First Search):</b>  DFS recorre el laberinto de forma <b>vertical o profunda</b> , explorando un camino completamente antes de retroceder y probar otros. Se puede implementar con una <b>pila (LIFO)</b> o con recursión. Es eficiente en términos de memoria, especialmente para laberintos con caminos largos y estrechos. Sin embargo, <b>no garantiza</b> encontrar la ruta más corta, ya que puede desviarse por rutas extensas sin salida antes de llegar al destino. A pesar de eso, su estructura simple lo hace útil en escenarios donde la eficiencia de tiempo no es una prioridad.			
3. <b>Backtracking:</b>  El backtracking es una técnica de prueba y error que <b>explora todas las posibles soluciones</b> , descartando aquellas que no cumplen las condiciones requeridas. Se implementa comúnmente con recursión y es ideal para resolver laberintos donde se quiere verificar <b>todas las rutas posibles</b> , no solo una. Cuando se aplica correctamente, puede encontrar una solución incluso en los casos más complejos. Sin embargo, su <b>complejidad computacional puede</b>			

	VICERRECTORADO DOCENTE	Código: GUIA-PRL-001
	CONSEJO ACADÉMICO	Aprobación: 2016/04/06
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		

**ser alta**, ya que el número de posibles caminos aumenta rápidamente en laberintos grandes, lo que hace necesario el uso de técnicas como la poda o la memoización.

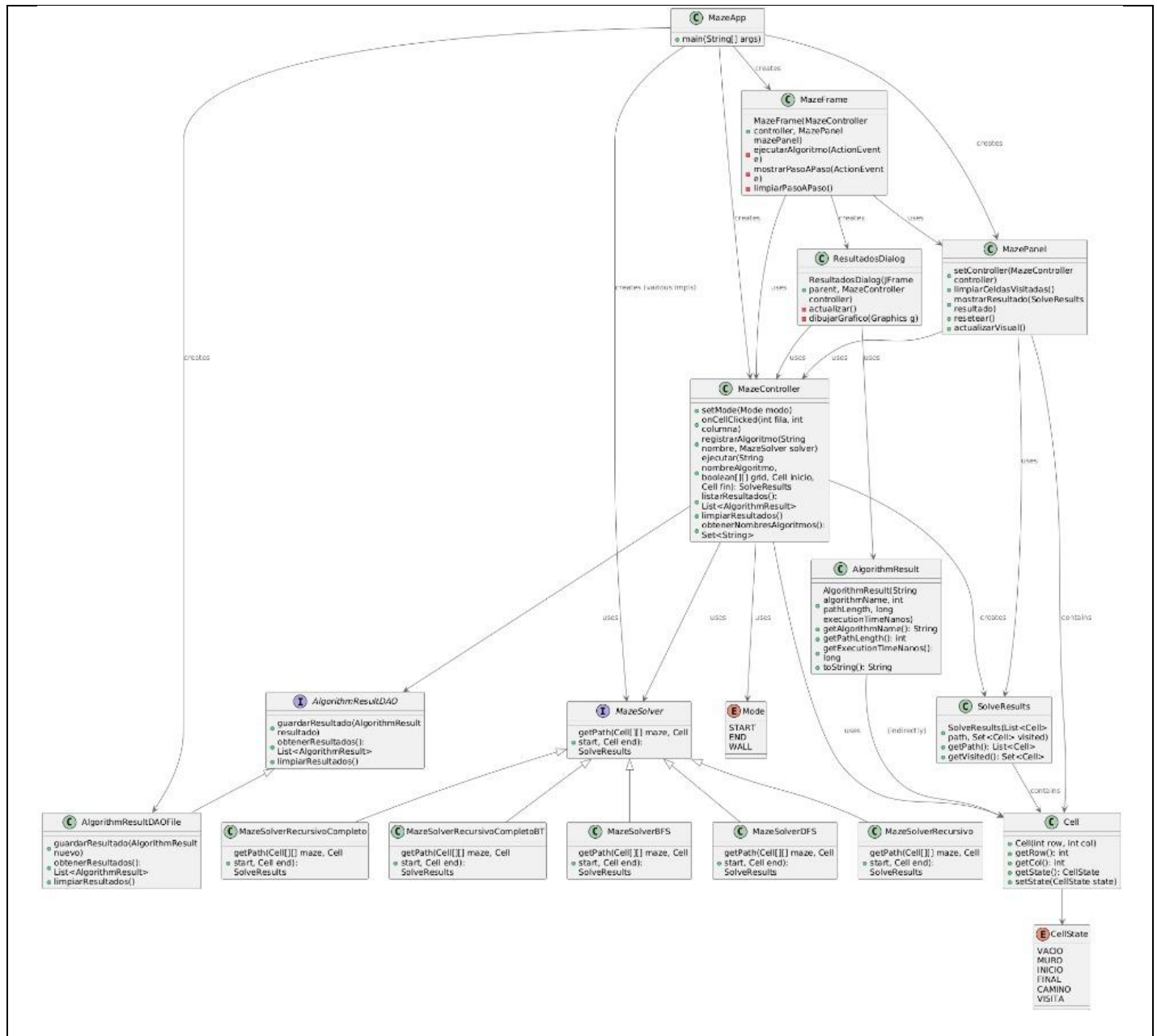
#### 4. Recursión:


La recursión es una herramienta fundamental en muchos algoritmos de búsqueda. Consiste en que una función se llama a sí misma con parámetros más simples, hasta llegar a un caso base que detiene las llamadas. Es muy útil para expresar algoritmos como DFS y backtracking de forma clara y compacta. No obstante, su mal uso puede provocar **desbordamiento de pila** (stack overflow) si la profundidad del laberinto es grande o si no se controla bien la condición de parada. Aun así, en muchos casos es la opción más natural para recorrer estructuras complejas como laberintos o árboles.

#### Tecnologías Utilizadas

- Lenguaje: **Java (JDK 11+)**
- Interfaz gráfica: **Java Swing**
- Control de versiones: **Git y GitHub**
- Persistencia de datos: **Archivos .csv** para almacenar los tiempos de ejecución
- Organización del proyecto: **Patrón MVC (Modelo – Vista – Controlador)**

#### Diagrama UML




	VICERRECTORADO DOCENTE	Código: GUIA-PRL-001
	CONSEJO ACADÉMICO	Aprobación: 2016/04/06
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		



La interfaz corresponde a una aplicación diseñada para resolver laberintos usando distintos algoritmos de búsqueda como DFS, BFS o A\*. Su objetivo principal es permitir al usuario seleccionar un algoritmo que encuentre una ruta desde un punto inicial (verde) hasta un punto final (rojo) dentro del laberinto, el cual está formado por celdas libres y obstáculos (muros negros). Durante la ejecución, el programa muestra visualmente el camino explorado (rosado) y, en caso de encontrarlo, el camino final hacia la meta. El usuario puede configurar la posición de inicio, fin y muros, así como controlar la ejecución con botones para resolver, avanzar paso a paso, limpiar el laberinto o ver resultados estadísticos. Por ejemplo, con DFS el algoritmo explora profundamente antes de retroceder, lo que puede llevar a caminos no óptimos pero es sencillo de entender. En resumen, esta herramienta ayuda a visualizar y comparar el comportamiento y eficiencia de diferentes algoritmos de búsqueda aplicados a laberintos.



La interfaz corresponde a una aplicación diseñada para resolver laberintos usando distintos algoritmos de búsqueda como BFS, DFS o A\*. Su objetivo principal es permitir al usuario seleccionar un algoritmo que encuentre una ruta desde un punto inicial (verde) hasta un punto final (rojo) dentro del laberinto, el cual está formado por celdas libres y obstáculos (muros negros). Durante la ejecución, el programa muestra visualmente el camino explorado (rosado) y,

	<b>VICERRECTORADO DOCENTE</b>	<b>Código:</b> GUIA-PRL-001
	CONSEJO ACADÉMICO	<b>Aprobación:</b> 2016/04/06
<b>Formato:</b> Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		

en caso de encontrarlo, el camino final hacia la meta. El usuario puede configurar la posición de inicio, fin y muros, así como controlar la ejecución con botones para resolver, avanzar paso a paso, limpiar el laberinto o ver resultados estadísticos. Por ejemplo, con BFS el algoritmo explora los nodos en orden de cercanía desde el inicio, garantizando encontrar el camino más corto en términos de pasos. En resumen, esta herramienta ayuda a visualizar y comparar el comportamiento y eficiencia de diferentes algoritmos de búsqueda aplicados a laberintos.

#### **Codigo del ejemplo de BFS:**

Este código define una clase MazeSolverBFS que implementa la interfaz MazeSolver para encontrar un camino en un laberinto representado por una matriz de celdas (Cell[][]). El método principal getPath recibe el laberinto, la celda de inicio y la celda de destino, y devuelve un objeto SolveResults con el camino encontrado y las celdas exploradas.

Primero, se inicializan variables para filas y columnas, una matriz booleana visitedMatrix que marca las celdas ya visitadas, un mapa parent para guardar la relación de cada celda con la anterior (esto permite reconstruir el camino), una cola queue para manejar las celdas a explorar siguiendo la lógica FIFO propia de BFS, y una lista explored para registrar las celdas que se visitan.

Se añade la celda inicial a la cola y se marca como visitada. Luego, mientras la cola no esté vacía, se procesa la celda al frente (current), añadiéndola a explored. Si current es la celda destino, se detiene la búsqueda.

```
public class MazeSolverBFS implements MazeSolver { 1 usage  🧑 Melany
    @Override 1 usage  🧑 Melany
    public SolveResults getPath(Cell[][] maze, Cell start, Cell end) {
        int rows = maze.length;
        int cols = maze[0].length;
        boolean[][] visitedMatrix = new boolean[rows][cols];
        Map<Cell, Cell> parent = new HashMap<>();
        Queue<Cell> queue = new LinkedList<>();
        List<Cell> explored = new ArrayList<>();

        Cell inicio = maze[start.getRow()][start.getCol()];
        Cell destino = maze[end.getRow()][end.getCol()];
        queue.add(inicio);
        visitedMatrix[inicio.getRow()][inicio.getCol()] = true;

        while (!queue.isEmpty()) {
            Cell current = queue.poll();
            explored.add(current);

            if (current.equals(destino)) break;

            for (int[] dir : new int[][]{{1, 0}, {-1, 0}, {0, 1}, {0, -1}}) {
                int r = current.getRow() + dir[0];
                int c = current.getCol() + dir[1];
                if (r >= 0 && r < rows && c >= 0 && c < cols) {
                    Cell neighbor = maze[r][c];
                    if (!visitedMatrix[r][c] && neighbor.getState() != CellState.MURO) {
                        visitedMatrix[r][c] = true;
                        parent.put(neighbor, current);
                        queue.add(neighbor);
                    }
                }
            }
        }
    }
}
```

```

        List<Cell> path = new ArrayList<>();
        Cell current = destino;
        while (parent.containsKey(current)) {
            path.add(current);
            current = parent.get(current);
        }

        if (current.equals(inicio)) {
            path.add(inicio);
            Collections.reverse(path);
        } else {
            path.clear();
        }

        return new SolveResults(path, new LinkedHashSet<>(explored));
    }
}

```

A continuación, se revisan los cuatro vecinos adyacentes (arriba, abajo, izquierda, derecha). Para cada vecino dentro de los límites del laberinto, que no sea muro y que no haya sido visitado, se marca como visitado, se registra su celda padre (la actual) y se añade a la cola para explorar más adelante.

Una vez finalizada la búsqueda (ya sea porque se encontró el destino o no quedan celdas por explorar), se reconstruye el camino desde la celda destino hacia atrás utilizando el mapa parent. Si se llega hasta la celda inicial, se invierte la lista para obtener el camino en orden correcto; si no, significa que no hay camino válido y se devuelve una lista vacía.


Finalmente, el método retorna un objeto SolveResults que contiene el camino encontrado y el conjunto de celdas exploradas durante la búsqueda, para que puedan ser visualizadas o analizadas posteriormente.

#### RESULTADO(S) OBTENIDO(S):

- Implementación funcional de múltiples algoritmos de búsqueda para laberintos.
- Comparación del rendimiento de los algoritmos mediante tiempos de ejecución.
- Interfaz gráfica dinámica y funcional para la configuración y visualización del laberinto.
- Documento técnico detallado en el archivo README.md que respalda el desarrollo del proyecto.

#### CONCLUSIONES:

- La práctica permitió aplicar de manera integral los conocimientos de estructuras de datos y algoritmos, fortaleciendo la capacidad de análisis y resolución de problemas complejos.
- El algoritmo más eficiente depende del tamaño y complejidad del laberinto, pero BFS mostró un buen balance entre precisión y velocidad en laberintos medianos.

	<b>VICERRECTORADO DOCENTE</b>	<b>Código:</b> GUIA-PRL-001
	CONSEJO ACADÉMICO	<b>Aprobación:</b> 2016/04/06
<b>Formato:</b> Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		

- El uso del patrón MVC facilitó la organización del código y su mantenimiento, permitiendo una mejor separación entre lógica, interfaz y datos.

#### **Conclusión Melany Pintado:**

Desde mi experiencia, el algoritmo **DFS** es más adecuado cuando se requiere una implementación sencilla y eficiente en memoria. Aunque no garantiza la ruta más corta, su estructura recursiva permite una exploración rápida en laberintos estrechos o con pocas bifurcaciones. Su bajo consumo de recursos lo hace ideal para entornos con limitaciones de procesamiento o cuando la prioridad no es la optimización del camino.

#### **Conclusión Dayanna Chacha:**

Considero que el algoritmo **BFS** es el óptimo para resolver laberintos, ya que garantiza siempre la ruta más corta entre dos puntos. Durante las pruebas, demostró ser eficiente al encontrar soluciones rápidas y precisas, especialmente en laberintos donde existen múltiples caminos posibles. Aunque su consumo de memoria es mayor, los beneficios en precisión justifican su uso en la mayoría de los casos prácticos.

#### **RECOMENDACIONES:**

- Utilizar herramientas de control de versiones como Git desde el inicio del proyecto para un mejor seguimiento del desarrollo.
- Investigar previamente los fundamentos teóricos de cada algoritmo antes de su implementación para evitar errores comunes
- Optimizar el código con estructuras de datos adecuadas y técnicas como memorización para mejorar el rendimiento general del programa.

**Docente / Técnico Docente:** \_\_\_\_\_

**Firma 1:** \_\_\_\_\_ **Firma 2:** \_\_\_\_\_