

Práctica 2 – PREDA

El robot en busca del tornillo.

Vuelta atrás.

Nombre: Pablo Suárez Peña

DNI: 53647180W

Centro Asociado: Asturias

Email: psuarez142@alumno.uned.es

Código de Asignatura: 71902019

0) INDICE:

Portada	Pagina 1
Índice	Pagina 2
Introducción a la Práctica	Pagina 3
Introducción teórica	Pagina 3
Cuestiones teóricas	Pagina 6
Ejemplos de Ejecución	Pagina 7
Código fuente	Pagina 12
a-Diagrama de Clases	Pagina 12
b-Código fuentes (Clases)	Pagina 12
1- Clase Robot	Pagina 13
2- Clase Edificio	Pagina 19
3- Clase Casilla	Pagina 21

1) INTRODUCCIÓN A LA PRÁCTICA:

- a. Sistema Operativo : macOS Monterrey 12.6.8
- b. IDE : BlueJ 5.2.0
- c. Versión de Java 17.0.4.1

2) INTRODUCCIÓN TEÓRICA:

Este problema es un ejemplo de aplicación del esquema a la búsqueda de carminas en laberintos. Un robot se mueve en un edificio en busca de un tornillo. Se trata de diseñar un algoritmo que le ayude a encontrar el tornillo y a salir después del edificio. El edificio debe representarse como una matriz de entrada a la función, cuyas casillas contienen uno de los siguientes tres valores: L para "paso libre", E para "paso estrecho" (no cabe el robot) y T para "tornillo". El robot sale de la casilla (1,1) y debe encontrar la casilla ocupada por el tornillo. En cada punto, el robot puede tomar la dirección Norte, Sur, Este u Oeste siempre que no sea un paso demasiado estrecho. El algoritmo debe devolver la secuencia de casillas que componen el camino de regreso desde la casilla ocupada por el tornillo hasta la casilla (1,1). Supondremos que la distancia entre casillas adyacentes es siempre 1.

Como no es necesario encontrar el camino más corto, sino encontrar un camino lo antes posible, una búsqueda en profundidad resulta más adecuada que una búsqueda en anchura. Si el edificio fuera infinito entonces una búsqueda en profundidad no sería adecuada porque no garantiza que se pueda encontrar una solución. Nosotros suponemos que el edificio es finito. Al ir recorriendo casillas es posible que el robot llegue a una que no tenga salida, por lo que es necesario que el algoritmo disponga de un mecanismo para deshacer movimientos, lo que hace que el esquema de vuelta atrás sea el adecuado. Vamos a utilizar el esquema de vuelta atrás de forma que la búsqueda se detenga en la primera solución y devuelva la secuencia de casillas desde el tornillo hasta la salida También iremos registrando las casillas que ya se han explorado para evitar repeticiones.

Para representar las casillas podemos utilizar un registro de dos enteros x e y , que nos indiquen una posición en el tablero. Para registrar los nodos explorados podemos utilizar una matriz del tamaño del edificio pero de valores booleanos. Vamos a utilizar una lista de casillas para almacenar la solución y otra para las extensiones del camino k -prometedoras. Contaremos con las siguientes funciones del tipo *lista*: *CrearLista*, *ListaVacía?*, *Añadir* y *Primero*.

El algoritmo puede tomar entonces la siguiente forma :

```
tipo TEedificio = matriz[0..LARGO, 1..ANCHO] de caracter
tipo TEedificioB = matriz[0..LARGO, 1..ANCHO] de booleano
tipo TCasilla =
    registro
        x,y: entero
    fregistro
tipo TListaCasillas = Lista de TCasilla
fun BuscaTornillo (edificio: TEedificio, casilla: TCasilla,
    exploradas: TEedificioB, solucion: TListaCasillas, exito: booleano)
    exploradas[casilla.x, casilla.y] ← cierto
    si edificio[casilla.x, casilla.y] = T entonces
        solución ← CrearLista()
        solución ← Añadir(solución, casilla)
        exito ← cierto
    sino
        hijos ← Caminos(edificio, casilla)
        exito ← falso
        mientras ¬ exito ∧ ¬ ListaVacía?(hijos) hacer
            hijo ← Primero (hijos)
            si ¬ exploradas [hijo.x, hijo.y] entonces
                BuscaTornillo (edificio,hijo,exploradas,solución,exito)
            fsi
        fmientras
        si exito entonces
            solución ← Añadir(solución, casilla)
        fsi
    fsi
ffun
```

En el caso de encontrar el tornillo se detiene la exploración en profundidad y al deshacer las llamadas recursivas se van añadiendo a la solución las casillas que se han recorrido. Como se añaden al final de la lista, la primera será la del tornillo y la última la casilla (1,1), tal como queríamos. La función *caminos* comprueba que la casilla no es estrecha y que no está fuera del edificio. En cada caso en que esto se cumple añade a la lista de hijos una nueva alternativa a explorar.

```

fun Caminos (edificio: TEdificio, casilla: TCasilla): TListaCasillas
  var
    hijos: TListaCasillas
    casilla_aux: TCasilla
  fvar
    hijos  $\leftarrow$  CrearLista()
  si casilla.x+1  $\leq$  LARGO entonces
    si edificio[casilla.x+1,casilla.y]  $\neq$  E entonces
      casilla_aux.x  $\leftarrow$  casilla.x+1
      casilla_aux.y  $\leftarrow$  casilla.y
      hijos  $\leftarrow$  Añadir (solución, casilla_aux)
    fsi
  fsi
  si casilla.x-1  $\geq$  1 entonces
    si edificio[casilla.x-1,casilla.y]  $\neq$  E entonces
      casilla_aux.x  $\leftarrow$  casilla.x-1
      casilla_aux.y  $\leftarrow$  casilla.y
      hijos  $\leftarrow$  Añadir (solución, casilla_aux)
    fsi
  fsi
  si casilla.y+1  $\leq$  ANCHO entonces
    si edificio[casilla.x,casilla.y+1]  $\neq$  E entonces
      casilla_aux.x  $\leftarrow$  casilla.x
      casilla_aux.y  $\leftarrow$  casilla.y+1
      hijos  $\leftarrow$  Añadir (solución, casilla_aux)
    fsi
  fsi
  si casilla.y-1  $\geq$  1 entonces
    si edificio[casilla.x,casilla.y-1]  $\neq$  E entonces
      casilla_aux.x  $\leftarrow$  casilla.x
      casilla_aux.y  $\leftarrow$  casilla.y-1
      hijos  $\leftarrow$  Añadir (solución, casilla_aux)
    fsi
  fsi
  dev hijos
ffun

```

Suponemos "edificio" inicializado con la configuración del edificio y "exploradas" inicializado con todas las posiciones a falso.

El espacio de búsqueda del problema es un árbol en el que cada nodo da lugar como máximo a cuatro ramas correspondientes a las cuatro direcciones de búsqueda. El número de niveles del árbol es el número de casillas del tablero, n^2 . Luego una cota superior es 4^{n^2}

3) CUESTIONES TEÓRICAS:

1. Indica y razona sobre el coste temporal y espacial del algoritmo.

El coste del caso peor de los algoritmos de búsqueda (Vuelta atrás) exhaustiva es del orden del tamaño del espacio de búsqueda. Las funciones de poda que utilizemos reducen el coste, aunque con frecuencia no es posible evaluar de qué forma, porque la poda depende de los datos concretos a los que se aplica el algoritmo. Por ello, en la mayoría de los casos daremos una cota superior al coste que calcularemos en función del tamaño del árbol de soluciones.

En nuestro caso en particular:

El espacio de búsqueda del problema es un árbol en el que cada nodo da lugar como máximo a cuatro ramas correspondientes a las cuatro direcciones de búsqueda. El número de niveles del árbol es el número de casillas del tablero, n^2 . Luego una cota superior es 4^{n^2}

2. Explica qué otros esquemas pueden resolver el problema y razona sobre su idoneidad.

Para resolver el problema de búsqueda del tornillo en el edificio, el enfoque de "vuelta atrás" o "backtracking" es el más adecuado. A continuación, explicaré por qué este enfoque es idóneo y por qué los otros enfoques mencionados no son apropiados:

Algoritmos Voraces:

No idóneos porque los algoritmos voraces suelen buscar soluciones óptimas basadas en elecciones locales en lugar de explorar todas las opciones posibles. Dado que el objetivo no es encontrar la solución óptima en este caso, los algoritmos voraces no son adecuados.

Divide y Vencerás:

No idóneos porque el enfoque de "divide y vencerás" se utiliza para descomponer un problema en subproblemas más pequeños. No es adecuado para este problema, ya que no se presta a una descomposición natural en subproblemas más pequeños.

Programación Dinámica:

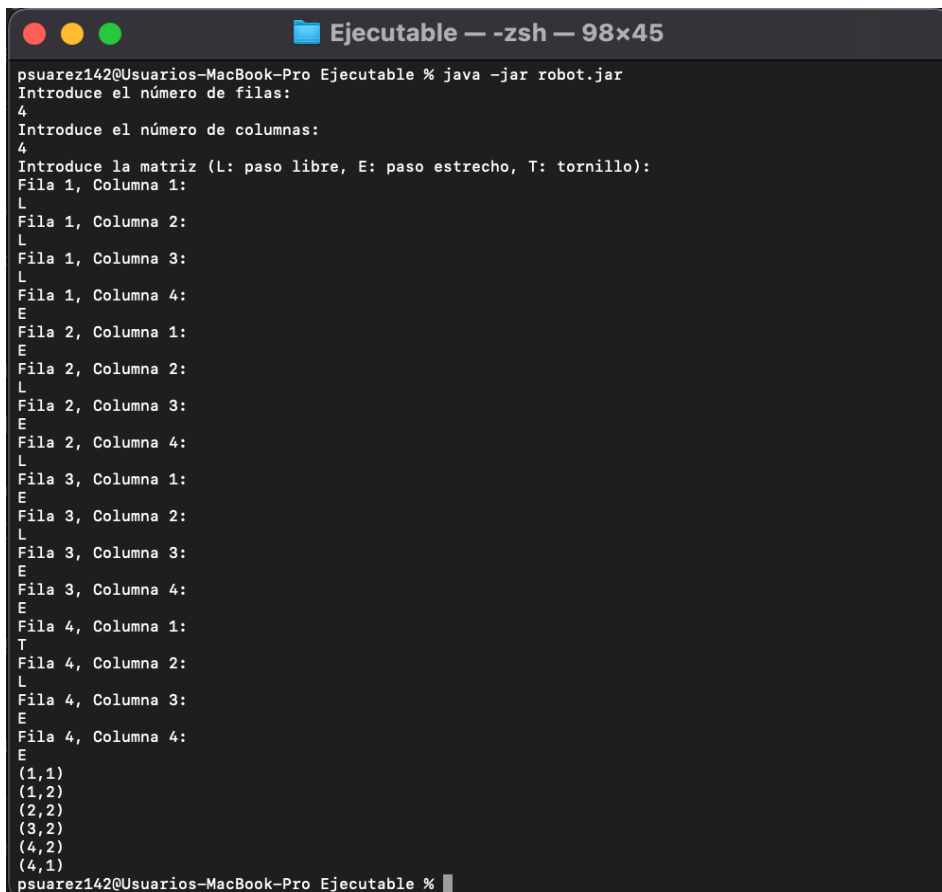
No idóneos porque la programación dinámica se utiliza para resolver problemas de optimización y no es adecuada para este problema, donde el objetivo es encontrar cualquier solución válida y no necesariamente la óptima.

Ramificación y Poda:

No idóneos porque la ramificación y poda se utiliza para buscar soluciones óptimas explorando ramas de búsqueda y podando aquellas que no conducen a soluciones mejores que la actual. Dado que el objetivo es encontrar cualquier solución válida y detenerse en la primera solución encontrada, este enfoque no es apropiado.

En resumen, el enfoque de "vuelta atrás" es el más adecuado para resolver el problema de búsqueda del tornillo en el edificio, ya que permite explorar todas las rutas posibles hasta encontrar una solución válida. Los otros enfoques mencionados no son apropiados para este problema debido a su naturaleza de búsqueda exhaustiva en lugar de búsqueda de soluciones óptimas.

4) EJMPLOS DE EJECUCIÓN:



```
psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar
Introduce el número de filas:
4
Introduce el número de columnas:
4
Introduce la matriz (L: paso libre, E: paso estrecho, T: tornillo):
Fila 1, Columna 1:
L
Fila 1, Columna 2:
L
Fila 1, Columna 3:
L
Fila 1, Columna 4:
E
Fila 2, Columna 1:
E
Fila 2, Columna 2:
L
Fila 2, Columna 3:
E
Fila 2, Columna 4:
L
Fila 3, Columna 1:
E
Fila 3, Columna 2:
L
Fila 3, Columna 3:
E
Fila 3, Columna 4:
E
Fila 4, Columna 1:
T
Fila 4, Columna 2:
L
Fila 4, Columna 3:
E
Fila 4, Columna 4:
E
(1,1)
(1,2)
(2,2)
(3,2)
(4,2)
(4,1)
psuarez142@Usuarios-MacBook-Pro Ejecutable %
```

```
Ejecutable — -zsh — 98x52
[psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar -t
Introduce el número de filas:
4
Introduce el número de columnas:
4
Introduce la matriz (L: paso libre, E: paso estrecho, T: tornillo):
Fila 1, Columna 1:
L
Fila 1, Columna 2:
L
Fila 1, Columna 3:
L
Fila 1, Columna 4:
E
Fila 2, Columna 1:
E
Fila 2, Columna 2:
L
Fila 2, Columna 3:
E
Fila 2, Columna 4:
L
Fila 3, Columna 1:
E
Fila 3, Columna 2:
L
Fila 3, Columna 3:
E
Fila 3, Columna 4:
E
Fila 4, Columna 1:
T
Fila 4, Columna 2:
L
Fila 4, Columna 3:
E
Fila 4, Columna 4:
E
Explorando la casilla (0,0)...
Explorando la casilla (0,1)...
Explorando la casilla (1,1)...
Explorando la casilla (2,1)...
Explorando la casilla (3,1)...
Explorando la casilla (3,0)...
¡Tornillo encontrado en la casilla (3,0)!
(1,1)
(1,2)
(2,2)
(3,2)
(4,2)
(4,1)
psuarez142@Usuarios-MacBook-Pro Ejecutable %
```

```
Ejecutable — -zsh — 98x7
[psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar -h
SINTAXIS: java -jar robot.jar [-t] [-h] [fichero_entrada] [fichero_salida]
-t: traza el algoritmo utilizado.
-h: muestra una ayuda.
[fichero_entrada] Nombre del fichero de entrada.
[fichero_salida] Nombre del fichero de salida.
psuarez142@Usuarios-MacBook-Pro Ejecutable %
```



```
Ejecutable — -zsh — 98x19
[psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar entrada.txt ]
(1,1)
(2,1)
(3,1)
(4,1)
(5,1)
(5,2)
(4,2)
(3,2)
(3,3)
(2,3)
(2,4)
(1,4)
(1,5)
(2,5)
(3,5)
(4,5)
(5,5)
psuarez142@Usuarios-MacBook-Pro Ejecutable %
```

```
entrada.txt
5
6
L L E L L E
L E L L L E
L L L E L L
L L E L L L
L L L L T L
```

```
Ejecutable — -zsh — 98x37
[psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar -t entrada.txt ]
Explorando la casilla (0,0)...
Explorando la casilla (1,0)...
Explorando la casilla (2,0)...
Explorando la casilla (3,0)...
Explorando la casilla (4,0)...
Explorando la casilla (4,1)...
Explorando la casilla (3,1)...
Explorando la casilla (2,1)...
Explorando la casilla (2,2)...
Explorando la casilla (1,2)...
Explorando la casilla (1,3)...
Explorando la casilla (0,3)...
Explorando la casilla (0,4)...
Explorando la casilla (1,4)...
Explorando la casilla (2,4)...
Explorando la casilla (3,4)...
Explorando la casilla (4,4)...
¡Tornillo encontrado en la casilla (4,4)!
(1,1)
(2,1)
(3,1)
(4,1)
(5,1)
(5,2)
(4,2)
(3,2)
(3,3)
(2,3)
(2,4)
(1,4)
(1,5)
(2,5)
(3,5)
(4,5)
(5,5)
psuarez142@Usuarios-MacBook-Pro Ejecutable %
```

```
Ejecutable — -zsh — 98x5
(2,5)
(3,5)
(4,5)
(5,5)
[psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar entrada.txt salida.txt]

entrada.txt
5
6
L L E L L E
L E L L L E
L L L E L L
L L E L L L
L L L L T L

salida.txt
Solución:
(5,5)
(4,5)
(3,5)
(2,5)
(1,5)
(1,4)
(2,4)
(2,3)
(3,3)
(3,2)
(4,2)
(5,2)
(5,1)
(4,1)
(3,1)
(2,1)
(1,1)

Camino Seguido:
(4,5)
(3,5)
(2,5)
(1,5)
(1,4)
(2,4)
(2,3)
(3,3)
(3,2)
(4,2)
(5,2)
(5,1)
(4,1)
(3,1)
(2,1)
(1,1)
```

```
Ejecutable — -zsh — 98x37
[psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar -t entrada.txt salida.txt
Explorando la casilla (0,0)...
Explorando la casilla (1,0)...
Explorando la casilla (2,0)...
Explorando la casilla (3,0)...
Explorando la casilla (4,0)...
Explorando la casilla (4,1)...
Explorando la casilla (3,1)...
Explorando la casilla (2,1)...
Explorando la casilla (2,2)...
Explorando la casilla (1,2)...
Explorando la casilla (1,3)...
Explorando la casilla (0,3)...
Explorando la casilla (0,4)...
Explorando la casilla (1,4)...
Explorando la casilla (2,4)...
Explorando la casilla (3,4)...
Explorando la casilla (4,4)...
¡Tornillo encontrado en la casilla (4,4)!
(1,1)
(2,1)
(3,1)
(4,1)
(5,1)
(5,2)
(4,2)
(3,2)
(2,3)
(2,4)
(1,4)
(1,5)
(2,5)
(3,5)
(4,5)
(5,5)
psuarez142@Usuarios-MacBook-Pro Ejecutable % ]
```

```
salida.txt
Solución:
(5,5)
(4,5)
(3,5)
(2,5)
(1,5)
(1,4)
(2,4)
(2,3)
(3,3)
(3,2)
(4,2)
(5,2)
(5,1)
(4,1)
(3,1)
(2,1)
(1,1)

Camino Seguido:
(4,5)
(3,5)
(2,5)
(1,5)
(1,4)
(2,4)
(2,3)
(3,3)
(3,2)
(4,2)
(5,2)
(5,1)
(4,1)
(3,1)
(2,1)
(1,1)
```

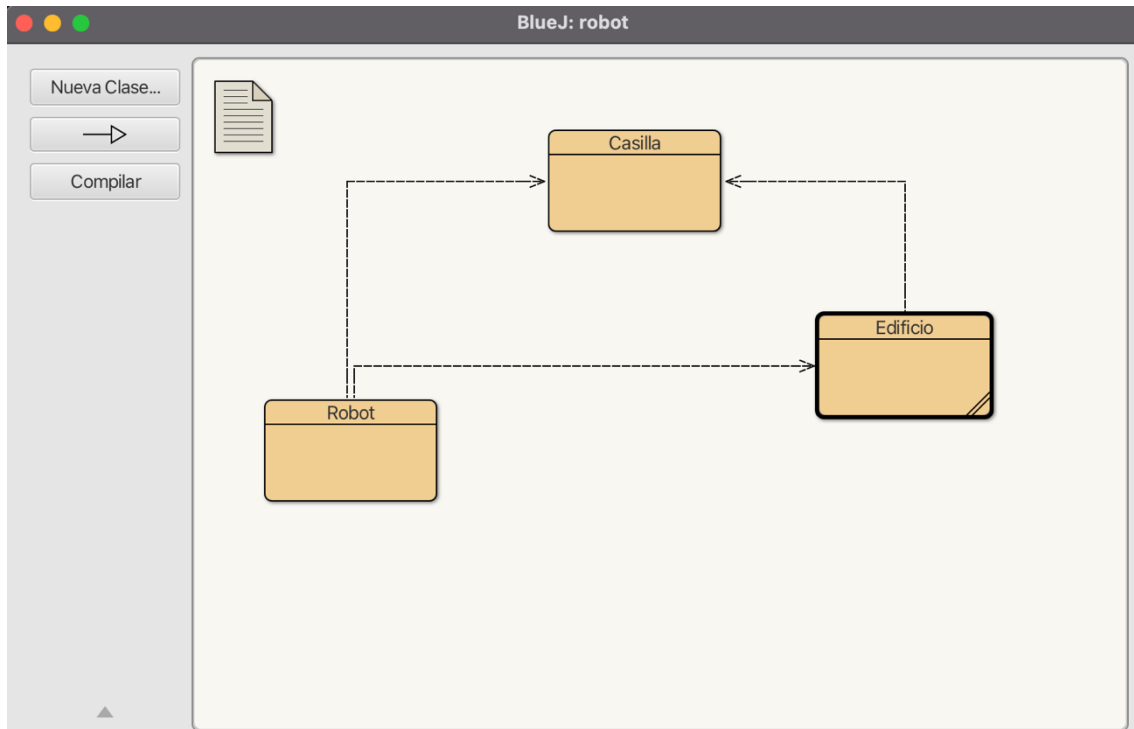
```
entrada.txt
5
6
L E L L E
L E L L E
L L E L L
L E L L L
L L L T L
```

```
Ejecutable — -zsh — 98x7
[psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar -h entrada.txt salida.txt
SINTAXIS: java -jar robot.jar [-t] [-h] [fichero_entrada] [fichero_salida]
-t: traza el algoritmo utilizado.
-h: muestra una ayuda.
[fichero_entrada] Nombre del fichero de entrada.
[fichero_salida] Nombre del fichero de salida.
psuarez142@Usuarios-MacBook-Pro Ejecutable % ]
```

```
Ejecutable — -zsh — 103x8
psuarez142@Usuarios-MacBook-Pro ~ % cd /Users/psuarez142/Desktop/PREDA_P2_pablo_suarezpeña/Ejecutable
[psuarez142@Usuarios-MacBook-Pro Ejecutable % java -jar robot.jar -t -h entrada.txt salida.txt
SINTAXIS: java -jar robot.jar [-t] [-h] [fichero_entrada] [fichero_salida]
-t: traza el algoritmo utilizado.
-h: muestra una ayuda.
[fichero_entrada] Nombre del fichero de entrada.
[fichero_salida] Nombre del fichero de salida.
psuarez142@Usuarios-MacBook-Pro Ejecutable % ]
```

5) CÓDIGO FUENTE:

5.a) Diagrama de clases:



5.b) Código Fuente de Completo*:

En nuestro código fuente hemos implementado 3 clases:

- Clase Robot
- Clase Casilla
- Clase Edificio

Robot:

Esta clase es la entrada principal del programa. En su método main, se realiza la lectura del edificio desde un archivo, se instancia la clase Edificio, se verifica si se debe realizar una traza, se muestra la ayuda si es necesario, se realiza la búsqueda del tornillo y se imprime o guarda la solución según los argumentos proporcionados en la línea de comandos.

Función leerEdificio:

Esta función se encarga de leer la configuración del edificio desde un archivo. Utiliza un `BufferedReader` para contar las filas y columnas, inicializa la matriz del edificio y la llena con los datos del archivo.

Función escribirSolucion:

Esta función se encarga de escribir la solución en un archivo. Utiliza un `BufferedWriter` para escribir las coordenadas de cada casilla en el archivo.

Función mostrarSolucion:

Esta función imprime la solución por consola. Recorre la lista de casillas en orden inverso y muestra las coordenadas de cada casilla en la consola.

Función mostrarAyuda:

Esta función imprime información de ayuda y la sintaxis del comando en la consola. Se llama cuando se proporciona el argumento `-h` en la línea de comandos.

```
import java.io.*;
import java.util.List;
import java.util.Scanner;
public class Robot {
    public static void main(String[] args) {
        try {
            String archivoEntrada = null;
            String archivoSalida = "salida.txt";
            boolean traza = false;

            // Procesar los argumentos
            boolean mostrarAyuda = false;
            boolean ordenCorrecto = true; // Flag para verificar
            el orden correcto
            for (int i = 0; i < args.length; i++) {
                String arg = args[i];
                switch (arg) {
                    case "-t":
                        traza = true;
                }
            }
        }
    }
}
```

```

        break;
    case "-h":
        mostrarAyuda();
        return; // Salir después de mostrar la
ayuda
        default:
            if (archivoEntrada == null) {
                archivoEntrada = arg;
            } else if
            (archivoSalida.equals("salida.txt")) {
                archivoSalida = arg;
            } else {
                // Argumentos fuera de orden
                ordenCorrecto = false;
            }
            break;
        }
    }

    // Verificar el orden de los argumentos
    if (!ordenCorrecto) {
        System.out.println("Error: Argumentos en orden
incorrecto.");
        mostrarAyuda();
        return;
    }

    // Mostrar la ayuda si se solicita
    if (mostrarAyuda) {
        mostrarAyuda();
    } else {
        // Resto del código sin cambios...

        char[][] edificio;

        if (archivoEntrada != null) {
            // Leer el edificio desde el archivo de
entrada
            try {
                edificio = leerEdificio(archivoEntrada);
            } catch (FileNotFoundException e) {
                // El archivo de entrada no existe,
mostrar mensaje y salir
                System.out.println("Error: El archivo de
entrada no existe.");
                return;
            } catch (IOException e) {
                // Otro error al leer el archivo de
entrada, mostrar mensaje y salir

```

```

        System.out.println("Error: No se pudo
leer el archivo de entrada.");
        return;
    }
} else {
    // Introducir los datos por la entrada
estándar.
    System.out.println("Introduce el número de
filas: ");
    int filas = leerEntero();

    System.out.println("Introduce el número de
columnas: ");
    int columnas = leerEntero();

    edificio = new char[filas][columnas];

    System.out.println("Introduce la matriz (L:
paso libre, E: paso estrecho, T: tornillo):");
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            System.out.println("Fila " + (i + 1)
+ ", Columna " + (j + 1) + ": ");
            edificio[i][j] = leerCaracter();
        }
    }

    // Resto del código sin cambios...
    Edificio laberinto = new Edificio(edificio);

    boolean exito = laberinto.buscarTornillo(traza);

    if (exito) {
        List<Casilla> solucion =
laberinto.getSolucion();
        List<Casilla> caminoSeguido =
laberinto.getCaminoSeguido();

        // Mostrar por la salida estándar el
resultado de la ejecución del programa.
        mostrarSolucion(solucion);

        // Manejar la creación y posible
sobrescritura del archivo de salida
        File fileSalida = new File(archivoSalida);
        if (fileSalida.exists() &&
fileSalida.isFile()) {

```

```

        // Si salida.txt ya existe: se
sobrescribirá el fichero de salida existente.
        escribirSolucion(solucion,
caminoSeguido, archivoSalida);
    } else {
        // Si salida.txt no existe se genera el
fichero salida.txt
        escribirSolucion(solucion,
caminoSeguido, archivoSalida);
    }
    } else {
        System.out.println("No se encontró el
tornillo. Fin de la exploración.");
    }
}

} catch (IOException e) {
    e.printStackTrace();
}
}

private static int leerEntero() {
    Scanner scanner = new Scanner(System.in);
    return scanner.nextInt();
}

private static char leerCaracter() {
    Scanner scanner = new Scanner(System.in);
    return scanner.next().charAt(0);
}

private static char[][] procesarEntradaEstándar() {
    Scanner scanner = new Scanner(System.in);

    System.out.print("Introduce el número de filas: ");
    int filas = scanner.nextInt();

    System.out.print("Introduce el número de columnas: ");
    int columnas = scanner.nextInt();

    char[][] edificio = new char[filas][columnas];

    System.out.println("Introduce la matriz (L: paso libre,
E: paso estrecho, T: tornillo):");
    for (int i = 0; i < filas; i++) {
        for (int j = 0; j < columnas; j++) {
            edificio[i][j] = scanner.next().charAt(0);
        }
    }
}

```



```

        return edificio;
    }

    private static char[][] leerEdificio(String fileName) throws
IOException {
        BufferedReader reader = null;

        try {
            reader = new BufferedReader(new
FileReader(fileName));

            // Lee el número de filas y columnas
            int filas = Integer.parseInt(reader.readLine());
            int columnas = Integer.parseInt(reader.readLine());

            char[][] edificio = new char[filas][columnas];

            // Lee la matriz desde el archivo
            for (int i = 0; i < filas; i++) {
                String linea = reader.readLine();
                for (int j = 0; j < columnas; j++) {
                    edificio[i][j] = linea.charAt(j * 2); //
Salta el espacio en blanco
                }
            }

            return edificio;
        } finally {
            if (reader != null) {
                reader.close();
            }
        }
    }

    private static void escribirSolucion(List<Casilla> solucion,
List<Casilla> caminoSeguido, String fileName) throws IOException
{
        try (BufferedWriter writer = new BufferedWriter(new
FileWriter(fileName))) {
            writer.write("Solución:\n");
            for (Casilla casilla : solucion) {
                writer.write("(" + (casilla.x + 1) + "," +
(casilla.y + 1) + ")\n");
            }

            writer.write("\nCamino Seguido:\n");
            for (Casilla casilla : caminoSeguido) {

```

```

        writer.write("(" + (casilla.x + 1) + "," +
(casilla.y + 1) + ")\n");
    }
}

private static void mostrarSolucion(List<Casilla> solucion)
{
    for (int i = solucion.size() - 1; i >= 0; i--) {
        Casilla casilla = solucion.get(i);
        System.out.println("(" + (casilla.x + 1) + "," +
(casilla.y + 1) + ")\n");
    }
}

private static void mostrarAyuda() {
    System.out.println("SINTAXIS: java -jar robot.jar [-t]
[-h] [fichero_entrada] [fichero_salida]");
    System.out.println("-t: traza el algoritmo utilizado.");
    System.out.println("-h: muestra una ayuda.");
    System.out.println("[fichero_entrada] Nombre del fichero
de entrada.");
    System.out.println("[fichero_salida] Nombre del fichero
de salida.");
}
}

```

Edificio:

Esta clase representa el edificio en el que se mueve el robot. Almacena la matriz de caracteres que describe la estructura del edificio y una matriz booleana que registra las casillas exploradas. La función principal es buscarTornillo, que utiliza una búsqueda en profundidad recursiva para encontrar el tornillo. También incluye funciones auxiliares como caminos para determinar los movimientos posibles desde una casilla y buscarTornilloRecursivo para llevar a cabo la búsqueda de manera recursiva.

```
import java.util.ArrayList;
import java.util.List;

public class Edificio {
    private char[][] matriz;
    private boolean[][] exploradas;
    private List<Casilla> solucion;
    private List<Casilla> caminoSeguido;

    public Edificio(char[][] matriz) {
        if (matriz == null || matriz.length == 0 ||
matriz[0].length == 0) {
            throw new IllegalArgumentException("La matriz no
puede ser nula o vacía");
        }

        this.matriz = matriz;
        this.exploradas = new
boolean[matriz.length][matriz[0].length];
        this.solucion = new ArrayList<>();
        this.caminoSeguido = new ArrayList<>();
    }

    public List<Casilla> getSolucion() {
        return solucion;
    }

    public List<Casilla> getCaminoSeguido() {
        return caminoSeguido;
    }

    public boolean buscarTornillo(boolean traza) {
        Casilla inicio = new Casilla(0, 0);
        boolean exito = buscarTornilloRecursivo(inicio, traza);
        return exito;
    }

    private boolean buscarTornilloRecursivo(Casilla casilla,
boolean traza) {
```

```

        exploradas[casilla.x][casilla.y] = true;

        if (traza) {
            System.out.println("Explorando la casilla (" +
casilla.x + "," + casilla.y + ")...");
        }

        if (matriz[casilla.x][casilla.y] == 'T') {
            solucion.add(new Casilla(casilla.x, casilla.y));

            if (traza) {
                System.out.println(";Tornillo encontrado en la
casilla (" + casilla.x + "," + casilla.y + ")!");
            }
            return true;
        }

        List<Casilla> hijos = caminos(casilla);

        for (Casilla hijo : hijos) {
            if (!exploradas[hijo.x][hijo.y]) {
                if (buscarTornilloRecursivo(hijo, traza)) {
                    solucion.add(new Casilla(casilla.x,
casilla.y));
                    caminoSeguido.add(new Casilla(casilla.x,
casilla.y));
                    return true;
                }
            }
        }

        return false;
    }

    private List<Casilla> caminos(Casilla casilla) {
        List<Casilla> hijos = new ArrayList<>();
        if (casilla.x + 1 < matriz.length && matriz[casilla.x +
1][casilla.y] != 'E') {
            hijos.add(new Casilla(casilla.x + 1, casilla.y));
        }
        if (casilla.x - 1 >= 0 && matriz[casilla.x -
1][casilla.y] != 'E') {
            hijos.add(new Casilla(casilla.x - 1, casilla.y));
        }
        if (casilla.y + 1 < matriz[0].length &&
matriz[casilla.x][casilla.y + 1] != 'E') {
            hijos.add(new Casilla(casilla.x, casilla.y + 1));
        }
    }

```

```

        if (casilla.y - 1 >= 0 && matriz[casilla.x][casilla.y -
1] != 'E') {
            hijos.add(new Casilla(casilla.x, casilla.y - 1));
        }
        return hijos;
    }
}

```

Casilla:

Una simple clase que representa una casilla en el edificio. Cada instancia de Casilla tiene coordenadas x e y. Esta clase se utiliza para representar las posiciones en el edificio y para almacenar la solución encontrada.

```

public class Casilla {
    public int x;
    public int y;

    public Casilla(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

```