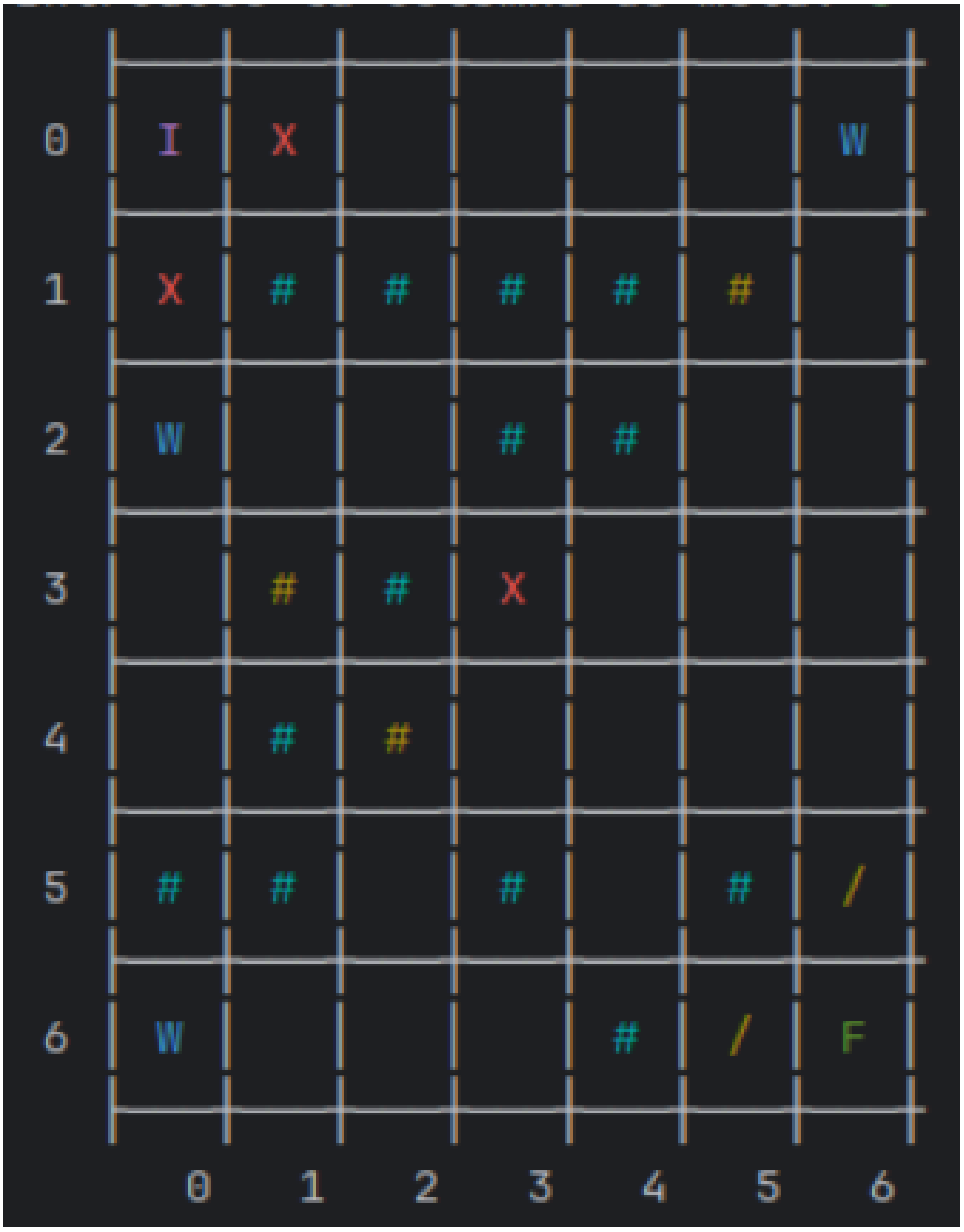
Algoritmo A* en Java



Jaime Alonso Fernández 2024/2025

Ingeniería del Conocimiento:Grado en Ingeniería Informática de la Universidad Complutense de Madrid. Plan académico 2019

1. Descripción del algoritmo base

La implementación del algoritmo base, se limita al mismo algoritmo que realizamos en papel en clase. Este va a contar con dos elementos dinámicos los cuales almacenarán las casillas visitadas y por visitar (lista abierta y lista cerrada). Para mí implementación he decidido crear una clase *Nodo* la cual contará con toda la información importante del mismo (su valor, su profundidad respecto a cada padre, su fila, su columna…)

Inicialmente contaremos con la casilla de salida únicamente en la lista abierta, al ser la única, esta será seleccionada para trasladarse a la lista cerrada. A partir de ahora cada elemento movido a la lista cerrada, añadirá a todas las casillas que le rodean (sus *Nodos* hijos) a la lista abierta siempre y cuando estos no estén ya en la lista cerrada.

Con estos nuevos elementos en la lista abierta, miraremos la heurística de cada uno (su menor profundidad respecto a los padres del *Nodo* en la lista cerrada) y moveremos a la lista cerrada a aquel que tenga menor heurística. En caso de haber un empate se tomará arbitrariamente.

El algoritmo finalizará cuando el *Nodo* que se mueva a la lista cerrada, sea aquel que buscamos, el *Nodo* objetivo.

1.1. Implementación del algoritmo base

Para entender el funcionamiento de la implementación, primero debemos entender la estructura de clases y paquetes que existen en el proyecto. Entendida esta parte fundamental, podemos pasar a ver los diferentes pasos que fueron tomados.

Estructura de paquetes:

Paquete Integración:

Dentro de este paquete se encuentran todas las clases ligadas a la interacción con el usuario. Para ello contamos con una Interfaz, la cual permite polimorfismo en el funcionamiento final, y una implementación de la misma que permite el uso de la terminal para trabajar con ella. Estas son declaradas de la siguiente manera:

- Interfaz *UserInterface*
- Clase *ConsoleUserInterface implements UserInterface*.

Paquete Estructura:

Dentro de este paquete se encuentran todas las clases ligadas a la estructura del proyecto. Contamos con las siguientes clases:

- Clase *Node*:
- Clase *NodeManager*:
- Clase *Table*:

- Clase *OrderedNodeList*:

Clase *Main*:

Como su nombre indica esta es la clase principal y desde la que se ejecuta todo. Es la encargada de llamar a los procesos iniciales haciendo uso de la interfaz y de la clase *Table*. Esta es la clase que se ha de ejecutar para probar el programa.

Inicialización de la tabla:

Comenzamos definiendo los valores numéricos que pueda tomar cada casilla de la tabla en función del tipo que sea:

Casilla Visitada	-1
Casilla Normal	0
Casilla de Salida	1
Casilla de Meta	2
Casilla Prohibida	4

Estos valores serán ingresados por el usuario y definirán las rutas que se puedan tomar.

Creando el primer *Nodo*:

En el momento de crear un *Nodo*, tendremos que hacer uso de una clase estática NodeManager. Esta cuenta con un método *createNewNode(padre, fila, columna, tabla, lista cerrada)* la cual se encargará de que el mismo se cree de la manera correcta, almacenando en el padre los hijos y en el mismo su padre.

Atendiendo a esto, durante el proceso de creación de la tabla registramos la fila y la columna correspondiente con la casilla inicial en una variable. Esto nos permite crear fácilmente el *Nodo* inicial.

Para que cuando lleguemos al *Nodo* final sepamos que ruta tomar de vuelta, vamos a estar usando un mapa con la estructura *Map<hijo, padre>* dónde se almacenará para cada hijo, su padre con la heurística más baja (siempre y cuando este se encuentre en la lista cerrada). Para el caso particular del *Nodo* inicial, usaremos como padre la variable *null*.

Ejecución del algoritmo:

Una vez creado el *Nodo* inicial, lo añadiremos a la lista abierta, el algoritmo lo sacará de la misma y lo moverá a la cerrada y calculará todos sus hijos (con su heurística) para añadirlos a la lista abierta.

Orden de prioridad del algoritmo:

Para mantener el orden correcto en todo momento durante la ejecución del algoritmo, he implementado una lista personalizada la cual utiliza una búsqueda binaria para ordenar los *Nodos* en función de su heurística. Esta clase *OrderedNodeList* es empleada únicamente en la lista abierta ya que la lista cerrada no necesita mantener ningún orden.

1.2. Expansión del algoritmo: Celdas Peligrosas

Acabada ya la base de la práctica, me propuse implementar todas las "expansiones" que pudiera. Para esta, la premisa es muy simple: Se añadiría un nuevo tipo de celda la cual se debería de evitar en la medida de lo posible, pero que en caso de no ser posible, se podría recorrer sin problema.

Para ello, comenzamos definiendo un nuevo tipo de celda que toma el siguiente valor de cara a la lógica del programa:

Casilla Peligrosa	3

Esta casilla tiene una repercusión en su heurística la cual hace que el cálculo base (acorde a la profundidad del *Nodo*) conlleve un aditamento del 10% de la diagonal de la tabla como penalización.

Por la naturaleza de los cálculos, esto no es contemplable con todas las disposiciones de tablas. En lo personal he percibido lo siguiente:

- Si existe un camino que pase de una celda 'a' a una celda 'b' a través de una celda peligrosa u a través de una celda normal en el mismo número de pasos, siempre se tomará el camino de la celda normal.

- Si existe un camino que pase de una celda 'a' a una celda 'b' a través de una celda peligrosa u a través de una celda normal pero esta vez conllevando un paso más, solo se tomará el camino de la celda normal. Esto variará según las dimensiones de la tabla.

1.3. Expansión del algoritmo: Waypoints

Continuando con las expansiones tras el correcto funcionamiento de las casillas peligrosas, solo me faltaba implementar el uso de waypoints.

Para ello, comenzamos definiendo un nuevo tipo de celda que toma el siguiente valor de cara a la lógica del programa:

Casilla de Waypoint	5

Con ello también tuve que modificar el algoritmo para establecer una "ruta de búsqueda". El funcionamiento consiste en que cada vez que se añada un waypoint a la tabla, su valor de fila y columna se añadirán al final de un array que contiene la ruta a seguir. De esta manera, el primer waypoint que se ponga, será el primero en ser recorrido. Finalmente, la casilla final se añadirá al final del array.

Establecida la ruta, el seguimiento será sencillo: Aplicaremos el algoritmo desde el *Nodo* inicial hasta el primer waypoint, luego desde el primer waypoint al segundo y así hasta llegar al final. Una ejecución con 3 waypoints tomaría la siguiente ruta:

I - w1; w1 - w2; w2 - w3; w3 - F

Dónde **I** es la casilla inicial, **F** la final y **w?** el waypoint correspondiente.

Esto, si bien era funcional, no proporcionaba una buena visión del recorrido de la tabla, ya que a veces ocurría que una casilla se visitaba más de una vez. Para ello decidí añadir un nuevo tipo de casilla llamada casilla revisitada, la cual tiene un color diferente. Su identificador en el código es el siguiente:

Casilla de Revisitada	-2

2. Manual de uso:

Requisitos:

- Esta entrega no cuenta con un archivo ejecutable por lo que es imprescindible tener jdk 17 o superior instalado en tu sistema operativo.

Ejecución:

- La ejecución podrá llevarse a cabo desde eclipse o IntellIJ, importando zip el proyecto y ejecutando el archivo **Main.java**.

Como usar:

Tras ejecutar la aplicación, inicialmente aparecerá para ingresar las dimensiones, en formato fila y columna, de la tabla sobre la que vamos a aplicar el algoritmo. Este valor ha de ser mayor que 2 y no se recomienda trabajar con más de 15 debido a problemas de visibilidad, aunque el algoritmo funcionará. (Imagen 1)

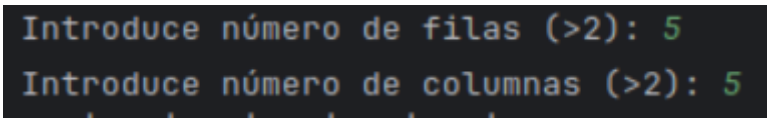


Imagen 1: Requisito de filas y columnas

En caso de introducir erróneamente los datos, estos serán pedidos hasta su cumplimiento (Imagen 2).

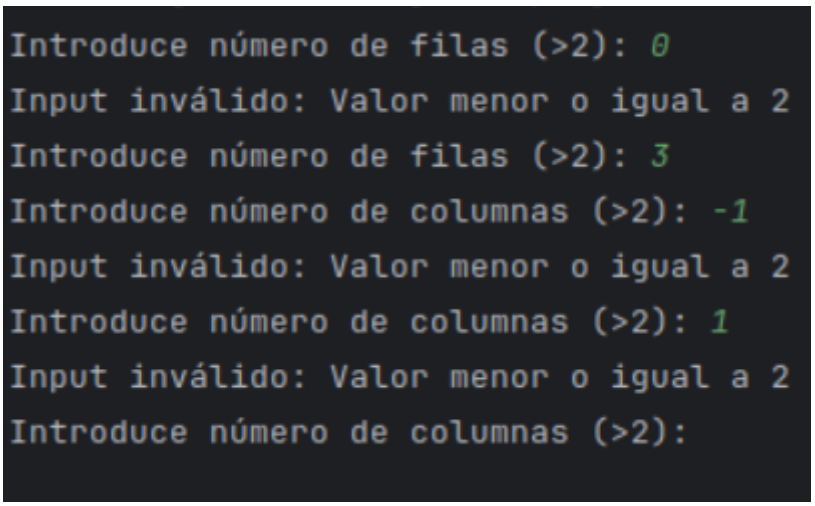


Imagen 2: Requisito de filas y columnas incorrecto

Pasado el proceso de creación de la tabla, se mostrará el estado inicial de la misma y se preguntará que se desea hacer, ofreciendo un abanico de opciones (Imagen 3)

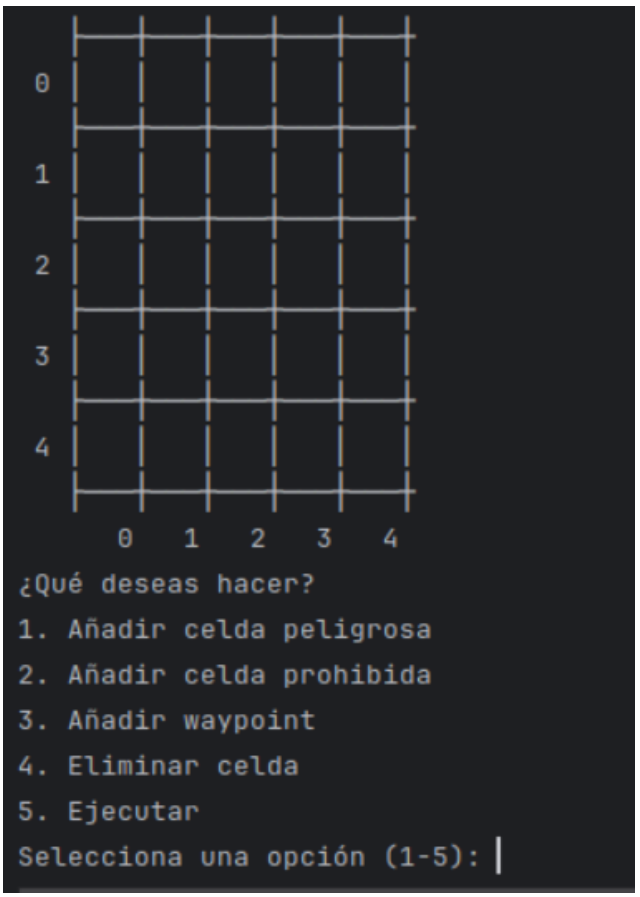
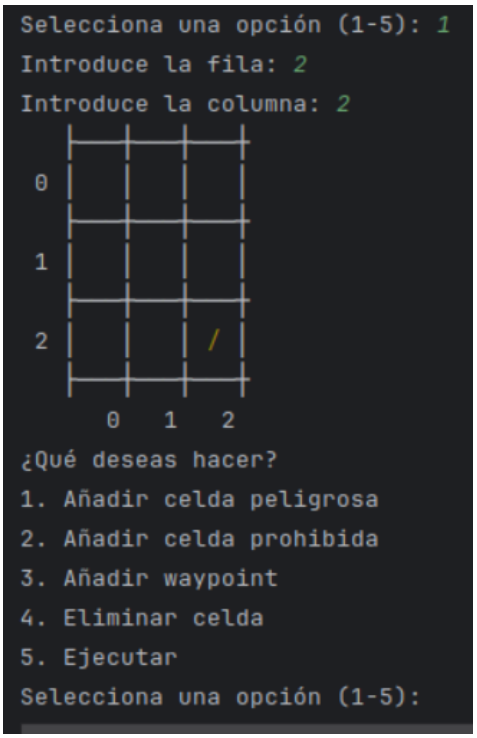


Imagen 3: Opciones iniciales

1. Añadir celda peligrosa:
   Para este apartado se preguntará por una fila y una columna válidas. (Imagen 4) En caso de error se mostrará el error acorde en pantalla (Imagen 5) y se cambiará el estado de la tabla para esa celda al indicado.
2. Añadir celda prohibida:
   Ocurrirá lo mismo que en la celda peligrosa y prohibida.
3. Añadir waypoint:
   Ocurrirá lo mismo que en la celda peligrosa y prohibida.
4. Eliminar celda:
   Ocurrirá lo mismo que en la celda peligrosa, prohibida y de waypoint pero eliminando la información de la tabla.
5. Ejecutar:
   Pasará a una pantalla donde preguntará por una casilla inicial y otra final, ambas válidas, para finalmente acabar ejecutando el algoritmo.



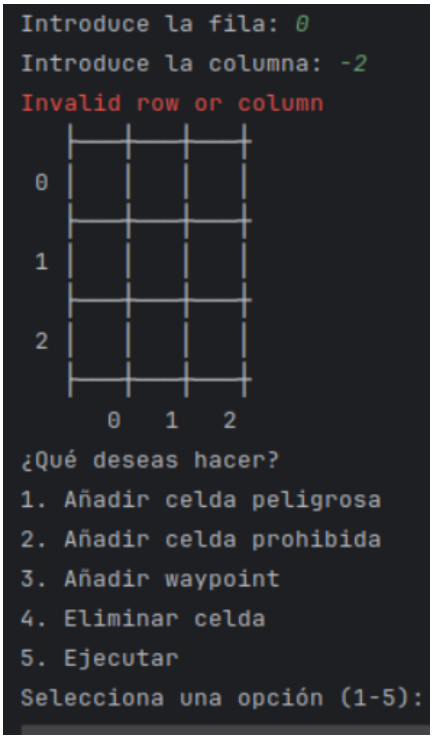


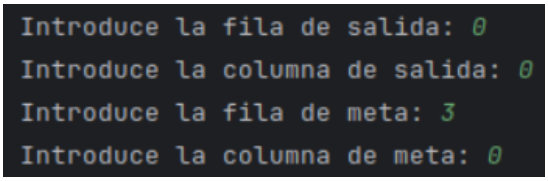
Imagen 4(izq): Dato ingresado correctamente.
Imagen 5 (dcha): Error al ingresar columna

Los simbolismos mostrados en la tabla serán los siguientes:

Casilla Normal	
Casilla de Salida	I
Casilla de Meta	F
Casilla Peligrosa	/
Casilla Prohibida	X
Casilla de Waypoint	W

El fondo negro se utiliza para marcar los colores que acompañan a cada caracter. Mencionar también que la casilla normal contiene el carácter *espacio* (“ “) por lo que este no será visible.

Cuándo se quiera ejecutar el algoritmo, seleccione la opción 5. Ésto pasará a pedir las filas y columnas para la casilla de inicio y la casilla de meta (Imagen 6), garantizando la inicialización correcta y única de cada una (Imagen 7). Añadida esta información, el algoritmo se ejecutará. Acabado el proceso, si se desea repetir el algoritmo, se ha de empezar desde el principio



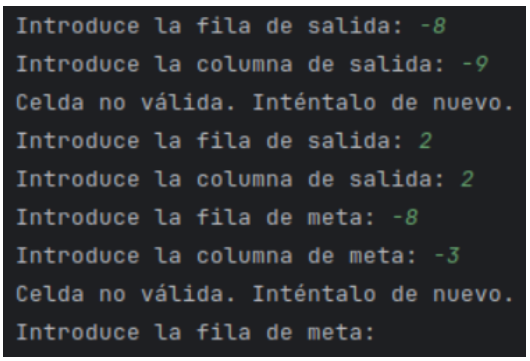


Imagen 6 (izq): Petición de casilla de salida y meta
Imagen 7 (dcha): Garantía de funcionamiento de la casilla de salida y de meta

Salida:

Al finalizar la ejecución del programa, existen dos casos pensados:

a) Existe camino: En este caso, se mostrará de nuevo en la terminal la tabla pero esta vez contando con un camino marcado. Esta ruta será identificada de la siguiente manera:

Celda Recorrida	#
Celda re-Recorrida	#

b) No existe camino: En este caso se mostrará un texto en la terminal de color rojo indicando eso mismo. El resultado esperado es el siguiente:

```
No existe camino posible para esta disposición. Contempla cambiar la posición de las celdas prohibidas si es posible.
```

En ambos casos, tras mostrar el mensaje pertinente en la consola, se detendrá la ejecución y en el supuesto de querer probar otro caso se tendrá que reiniciar la aplicación como se mencionó anteriormente.

3. Ejemplos de uso

Durante este apartado vamos a mostrar los resultados para cada uno de los casos e implementaciones que hemos estado haciendo. Denotar que la ejecución se está realizando con la última versión del programa y que hubo fases del proyecto donde las opciones “Añadir celda peligrosa” o “Añadir Waypoint” no existían.

3.1. Camino básico:

Para esta ejecución inicializamos una tabla de 5x8. Pasamos directamente a la ejecución de la misma optando por la opción 5, y elegimos como casilla inicial la (0,0) y como casilla final la (4, 7).

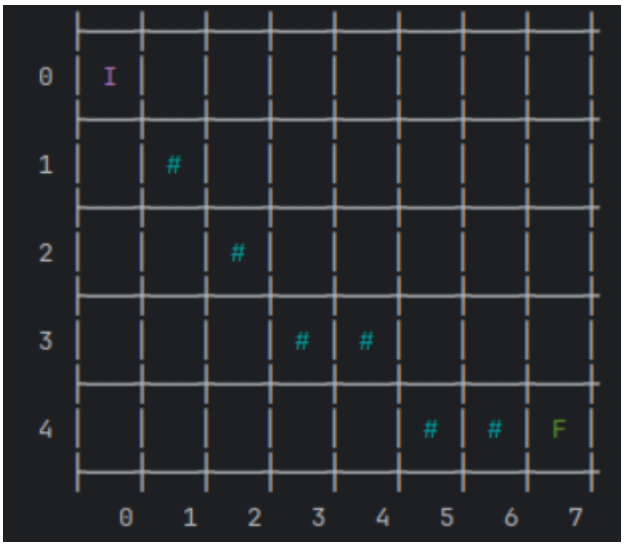


Imagen 8: Ejecución básica

3.2. Camino cortado:

Para esta ejecución inicializamos una tabla de 5x5. Marcamos como prohibida la casilla central eligiendo la opción 2 y marcando las casillas (2, 2). Pasamos la ejecución del algoritmo optando por la opción 5, y elegimos como casilla inicial la (0,0) y como casilla final la (4, 4).

Imagen 9: Ejecución con camino cortado

3.3. Celda peligrosa evitable:

Para esta ejecución inicializamos una tabla de 8x8. Marcamos como peligrosa la casilla (0,1) eligiendo la opción 1. Pasamos la ejecución del algoritmo optando por la opción 5, y elegimos como casilla inicial la (0,0) y como casilla final la (0,2). Al poder evitarse la casilla peligrosa, se evita.

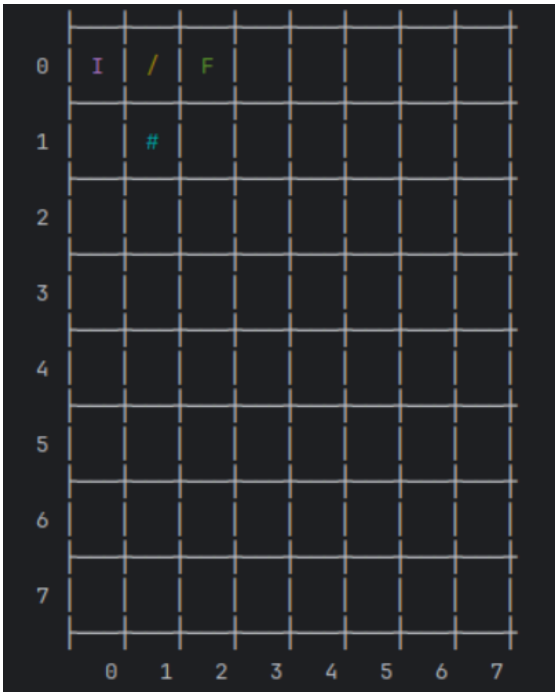


Imagen 10: Ejecución de casilla peligrosa evitable

3.4. Casilla peligrosa no evitable:

Para esta ejecución inicializamos una tabla de 3x3. Marcamos como peligrosa la casilla (1,1) eligiendo la opción 1, con la opción 2 marcamos como prohibidas las casillas (1, 0) y (0, 1). Pasamos la ejecución del algoritmo optando por la opción 5, y elegimos como casilla inicial la (0,0) y como casilla final la (0, 2). Al no existir ruta alternativa, el camino tomado pisa la casilla peligrosa.

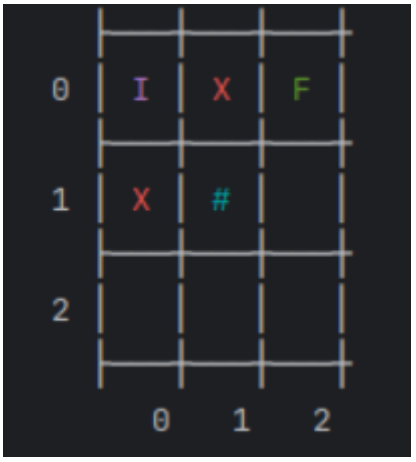


Imagen 11: Ejecución con casilla peligrosa no evitable

3.5.Waypoints sin solapamiento:

Para esta ejecución inicializamos una tabla de 3x3. Marcamos como waypoints las casillas (0, 2) y (2,0) en ese orden eligiendo la opción 3 . Pasamos la ejecución del algoritmo optando por la opción 5, y elegimos como casilla inicial la (0,0) y como casilla final la (2, 2).

Imagen 12: Ejecución con waypoints sin solapamiento

La ruta tomada en este caso es la siguiente:

(0, 0) - (0, 2); (0, 2) - (2, 0); (2, 0) - (2, 2)

3.6.Waypoints con solapamiento:

Para esta ejecución inicializamos una tabla de 5x5. Marcamos como waypoints las casillas (0, 4), (4, 0) y (2, 2) en ese orden eligiendo la opción 3. Pasamos la ejecución del algoritmo optando por la opción 5 y elegimos como casilla inicial la (0,0) y como casilla final la (4, 4).

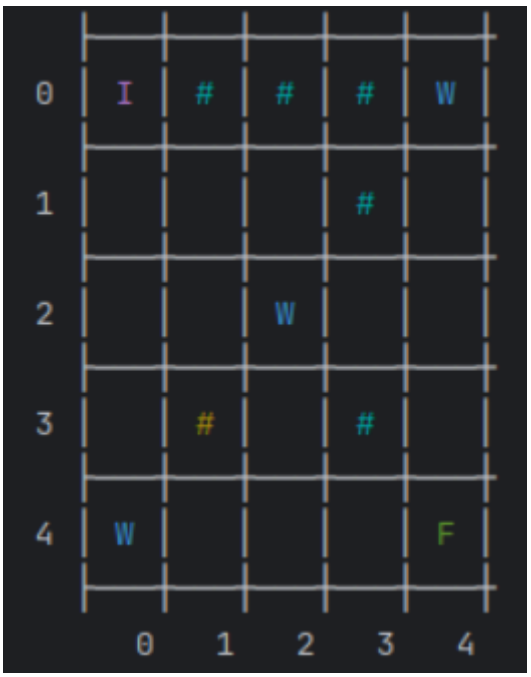


Imagen 13: Ejecución con waypoints con solapamiento

La ruta tomada en este caso es la siguiente:

(0, 0) - (0, 4); (0, 4) - (4, 0); (4, 0) - (2, 2); (2, 2) - (4, 4)

Donde apreciamos solapamiento en la ruta de la casilla (4, 0) a la casilla (2, 2) al haberse recorrido la casilla (3, 1) durante el paso de (0, 4) a (4, 0).

3.7. Uso completo:

Para esta ejecución inicializamos una tabla de 7x7. Luego marcamos como casillas prohibidas las casillas (1, 0), (0, 1) y (3, 3) mediante la opción 2. A continuación marcamos como waypoints las casillas (0, 6), (6, 0) y (2, 0) en ese orden eligiendo la opción 3. Ahora marcamos como peligrosas las casillas (6, 5) y (5, 6). Y finalmente pasamos la ejecución del algoritmo optando por la opción 5: elegimos como casilla inicial la (0,0) y como casilla final la (6, 6).

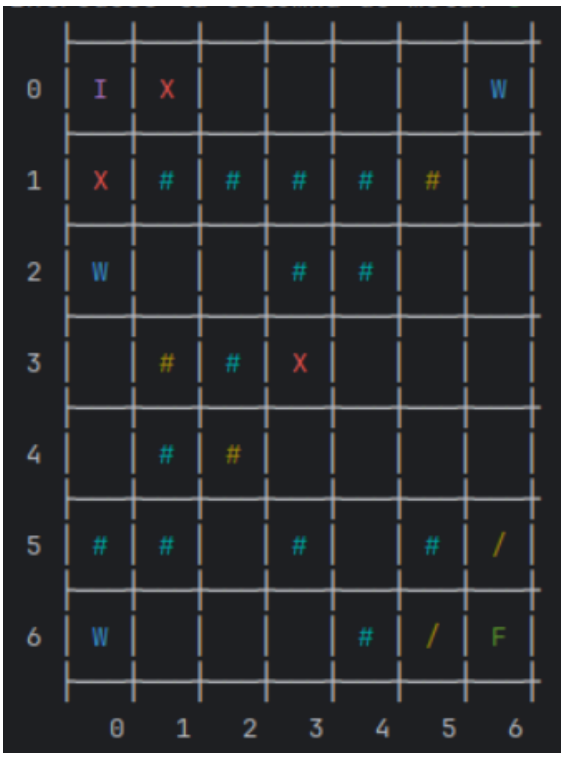


Imagen 14: Ejecución completa

Debido a la complejidad de la ruta tomada, esta ha sido indicada con líneas en la imagen mediante herramientas externas.

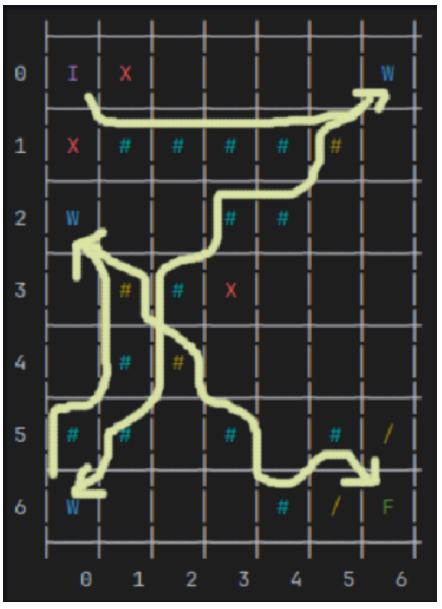


Imagen 15: Ejecución completa con la ruta marcada