UNIVERSIDAD COMPLUTENSE MADRID

Ingeniería del conocimiento

Práctica 1: Algoritmo A*

Álvaro del Campo Gragera
Juan Montero Gómez

12/03/2024

Índice

1. Introducción

El objetivo de la práctica es implementar el Algoritmo A*. Empezando por el lenguaje y herramientas usadas, se explicará cómo se calcula, en la implementación, la distancia óptima entre un punto inicial y uno final, considerando casillas adyacentes y obstáculos. Además, se describirá la estructura del proyecto, incluyendo la organización de archivos y la clase Celda. También se presentarán las dos funcionalidades opcionales: los peligros y los waypoints. Finalmente, se proporcionará un manual de usuario para ejecutar el programa y se mostrarán ejemplos de su funcionamiento con diferentes configuraciones de tableros.

2. Detalles de implementación

El Algoritmo A* se ha implementado en el lenguaje de programación C++, utilizando el entorno de desarrollo Visual Studio Code.

Para calcular la distancia desde un punto al final se calcula sumando su valor g, que es la distancia desde el punto de inicio hasta el punto actual, y su valor h, que es la distancia desde el punto actual hasta el punto final. De esta forma, se van comprobando los costes de las casillas adyacentes a la actual (ortogonales y diagonales) y se van guardando en una cola las posiciones con menor coste. El resultado final señalará la distancia más óptima para ir de un punto inicial a un punto final. También se pueden incluir casillas peligrosas, que son por las que el usuario no podrá pasar de ninguna forma.

A la hora de organizarlo todo, lo hemos dividido en cuatro archivos. El primero de todos, main.cpp, tiene las funciones relacionadas con la manipulación de la matriz de celdas y la interacción con el usuario. Estas son *pedirDimensiones()* para solicitar las dimensiones de la matriz, *pedirCoordenadas()* para solicitar coordenadas de puntos de inicio y destino, *printMatrix()* para imprimir la matriz en la consola, *trampas()* y *peligros()* para establecer obstáculos o peligros en ciertas celdas, y *waypoints()* para determinar los puntos intermedios en el camino.

También tenemos otro archivo aEstrella.cpp, contiene la implementación principal del algoritmo A*. La función *aEstrella()* toma la matriz de celdas, el punto de inicio y el punto de destino como entrada, y devuelve una cola de pares de coordenadas que representan el camino más corto desde el punto inicial hasta el punto final. La implementación utiliza una cola de prioridad (*priority_queue*) para explorar las celdas en orden de menor costo total, calculado como la suma de la distancia recorrida desde el punto inicial y una estimación heurística de la distancia restante hasta el punto final. Se utiliza una matriz de booleanos

para rastrear las celdas visitadas (*en_cerrado*) y una matriz adicional para rastrear las celdas que están siendo exploradas (*en_abierto*). El algoritmo se ejecuta en un bucle mientras hay celdas en la cola de prioridad, y termina cuando se encuentra el punto de destino o cuando la cola está vacía. Para cada celda explorada, se generan movimientos posibles en todas las direcciones (arriba, abajo, izquierda, derecha y diagonales) y se calcula el costo total para llegar a cada celda vecina.

Se utiliza la función *distancia()* para calcular la distancia entre dos puntos, *dentroLimites()* para verificar si una coordenada está dentro de los límites de la matriz, y *sePuedePasar()* para verificar si una celda es accesible. La función *imprimirCamino()* se encarga de imprimir en la consola el camino encontrado por el algoritmo A*, resaltando las celdas que forman parte del camino.

Finalmente, tenemos dos archivos .h: uno es *colors.h*, que sirve para representar la matriz con colores para que visualmente sea más atractivo, y el otro es *celda.h*, que contiene elementos comunes para ambos archivos .cpp. Creemos que esta forma de organizarlo es buena porque así todo está más distribuido para que sea más fácil de localizar lo que buscamos. Se podría dividir toda la parte de la vista en otro archivo, pero al no ser un código demasiado extenso no nos ha parecido necesario.

Para representar una celda usamos una clase de C++ que contiene todos los atributos necesarios para calcular el algoritmo A* y para mostrar el resultado en la vista. Estos atributos incluyen la letra que se muestra en la vista (letra), si la celda es parte del camino (camino), si es una trampa (paso), y varios valores utilizados por el algoritmo A* para calcular el camino más corto. También incluye atributos como la distancia heurística desde el punto actual hasta el destino (h), la distancia acumulada desde el punto inicial hasta el punto actual (g), el índice de peligrosidad (p), el índice de peligrosidad acumulado (p_acum), la suma total de h, g y p_acum (total), y las coordenadas de la celda padre (padre). Todo esto se puede observar claramente en la imagen 1.1.

```
class celda {
public:
    char letra = ' '; // La letra que se muestra en la vista
    bool camino = false; // Si es camino o no, para la vista

    bool paso = true; // Si es trampa o no
    double h = 0; // distancia desde punto actual hasta meta
    double g = 0; // distancia desde origen hasta el punto actual
    double p = 0; // indice de peligrosidad asignado (el que se muestra)
    double p_acum = 0; // indice de peligrosidad acumulado (calculo interno)
    double total = 0; // suma de h, g y p_acum
    pair<int, int> padre; // Las coordenadas x,y de la casilla padre
};
```

Imagen 1.1: Clase celda

También hemos implementado dos funcionalidades extras: los peligros y los waypoints. Los peligros son casillas por las que, en caso de cruzar por ellas, tendrán una penalización de las unidades con las que haya sido indicada previamente. Los waypoints, por otra parte, son casillas por las que tiene que pasar el algoritmo antes de llegar al final. Para implementar esto último, hemos estado llamado al Algoritmo A* por segmentos, entre el inicio y el waypoint, entre los propios waypoints, y entre el último de ellos y el punto final. Un ejemplo de cómo se ve el tablero con todos los elementos se puede ver en la imagen 1.2.

Imagen 1.2: Ejemplo de visualización de tablero

3. Manual de usuario

El primer paso es ejecutar el fichero, que se puede ejecutar tanto en Visual Studio como en Visual Studio Code. En caso de hacerlo en Visual Studio, la opción más sencilla, tan solo hay que crear un proyecto en el que se encuentren los cuatro archivos y ejecutar el programa con el botón de la flecha verde en la parte superior. En caso de hacerlo en Visual Studio Code, el proceso es un poco más laborioso. Tras haber creado un proyecto con los cuatro archivos en el mismo directorio, se tendrá que abrir la terminal (atajo con CTRL + Ñ) e introducir el siguiente comando (asumiendo que el sistema tiene instalado el compilador g++):

```
g++  -o main main.cpp aEstrella.cpp -o main.exe
```

Ahora sí, para ejecutar el fichero en terminal, se puede hacer con el siguiente comando:

```
./main.exe
```

Al iniciar, habrá que indicar poco a poco toda la información para crear el tablero. Primero se pide el número de filas y de columnas, con un tope máximo de 9. A continuación, se pedirán las posiciones de la casilla de inicio y la casilla final, teniendo en cuenta que el eje de las *x* es el horizontal y el de las *y* es el vertical, como se puede apreciar en la imagen 1.3 tras introducir 4 filas y 6 columnas.

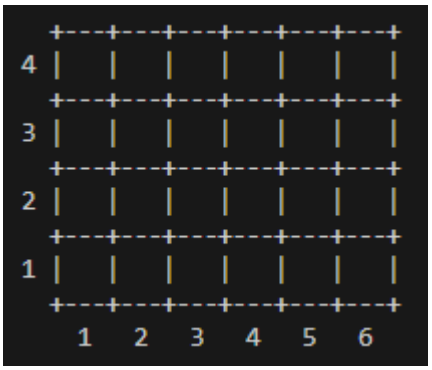


Imagen 1.3: Visualización del tablero vacío

Después de establecer las dimensiones del tablero, se le pedirá al usuario que especifique la posición inicial y final en el laberinto. Introduce las coordenadas (posición en eje x, posición en eje y) para ambos puntos. En caso de que los valores no se encuentren dentro de los límites, el programa repetirá la pregunta. En la imagen 1.4 se puede ver un ejemplo de introducir (2, 1) como punto inicial y (4, 4) como punto final.

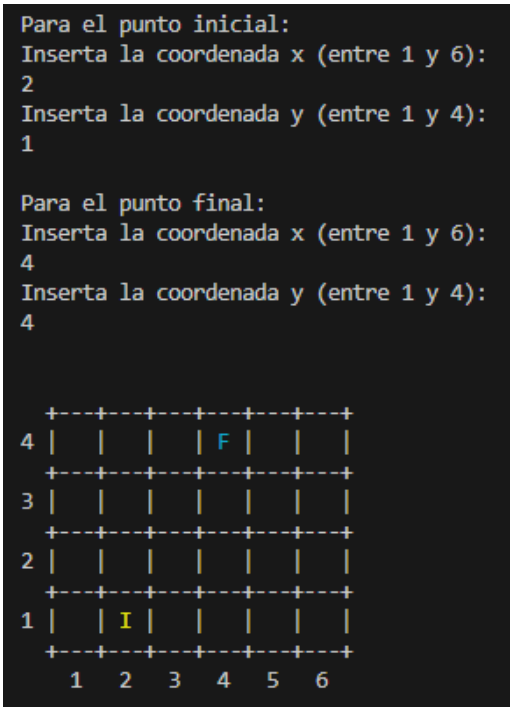


Imagen 1.4: puntos inicial y final

A continuación, se dará la opción de agregar trampas al tablero, que son áreas peligrosas por las que no se pueden pasar. Para agregar una trampa, ingresa las coordenadas (posición en eje x, posición en eje y) donde deseas colocarla. En la imagen 1.5 se puede ver cómo se han agregado trampas en las posiciones (3, 1), (3, 2) y (3, 3)

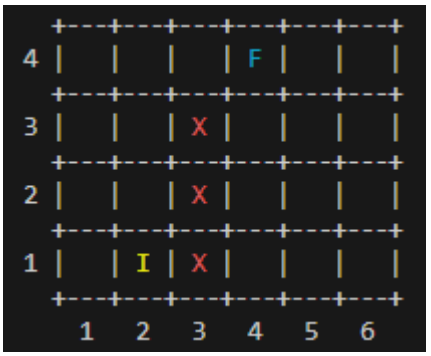


Imagen 1.5: Trampas agregadas

Después de agregar las trampas, se puede agregar también peligros al tablero, que son lugares con diferentes niveles de dificultad para cruzar. Hay que especificar las coordenadas (eje x, eje y) donde se desea colocar el peligro. Luego, se pedirá que se ingrese un índice de peligro en una escala del 1 al 9 para este lugar. En la imagen 1.6 se puede ver un ejemplo.

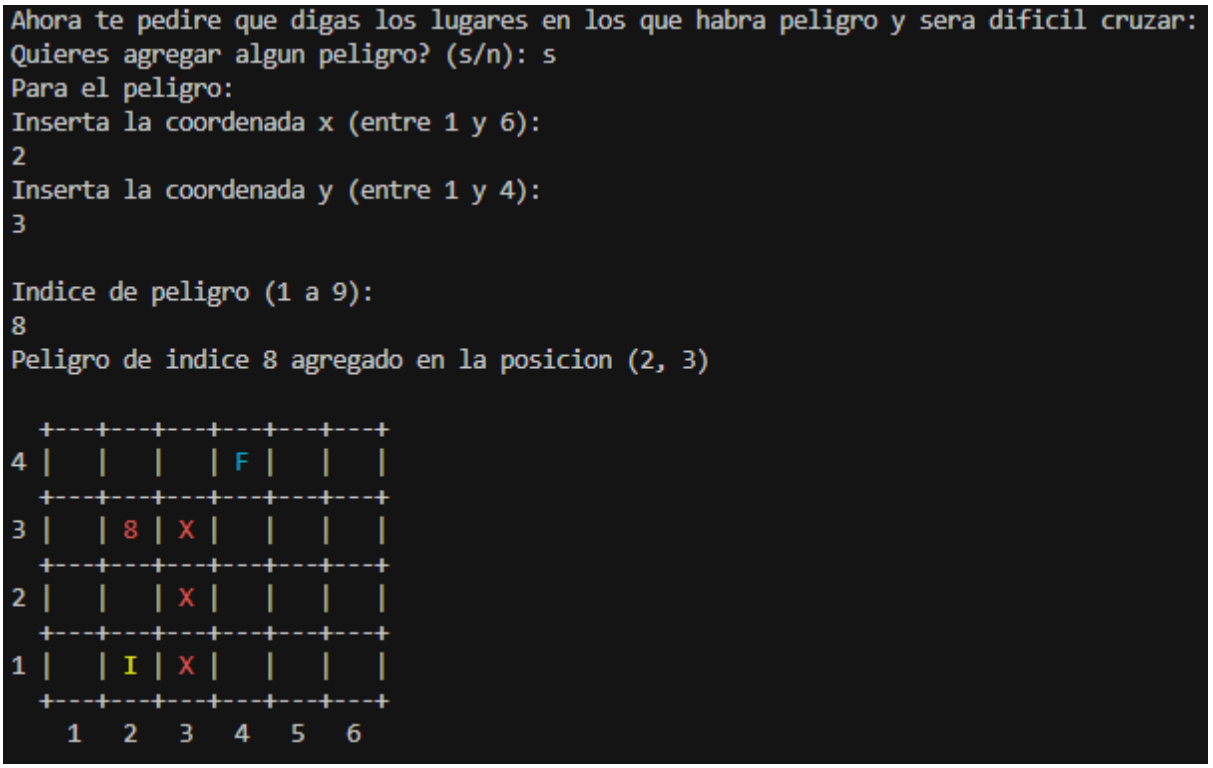


Imagen 1.6: Peligro de nivel 8 agregado

Finalmente, se preguntará si se desea agregar algún waypoint. Los waypoints son puntos de referencia o destinos intermedios en tu ruta a través del tablero. Se indican las coordenadas (eje x, eje y) donde se desea colocar cada waypoint. En el ejemplo de la imagen 1.7 se han agregado dos waypoints en las posiciones (4, 1) y (6, 3).

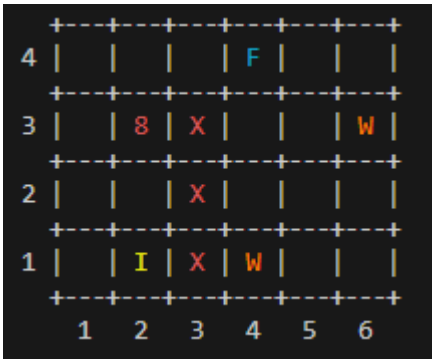


Imagen 1.7: Waypoints colocados

Una vez que se hayan configurado todas las características del tablero, la aplicación calculará la ruta menos cara para navegar desde la posición inicial hasta la posición final, pasando por todos los waypoints especificados. Esta ruta se mostrará en la consola junto con una representación visual del laberinto, resaltando la ruta en el mismo.

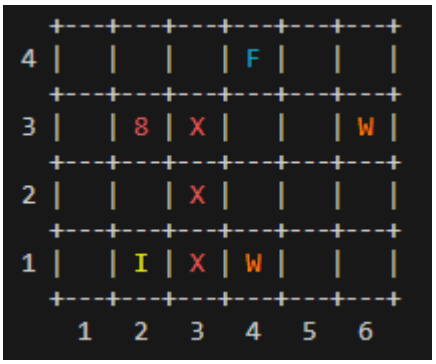
En la representación visual del tablero, las trampas se muestran con la letra 'X', los peligros con el índice de peligrosidad, y los waypoints con la letra 'W'.

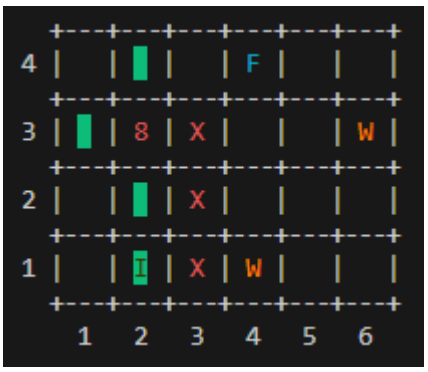
4. Ejemplos de ejecución

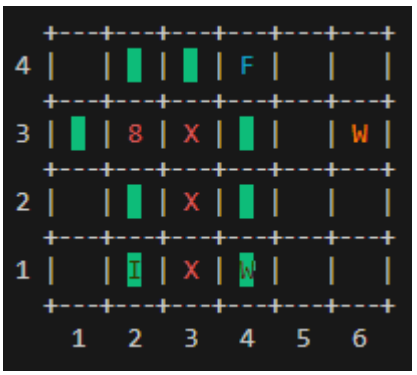
Para finalizar mostraremos varios ejemplos de la ejecución del algoritmo con tableros de distintos tamaños y probando todos los elementos que tiene el proyecto.

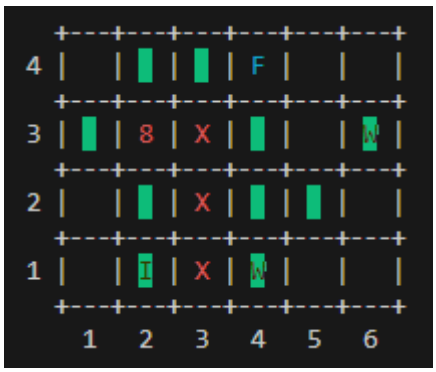
Ejemplo 1: El del apartado anterior

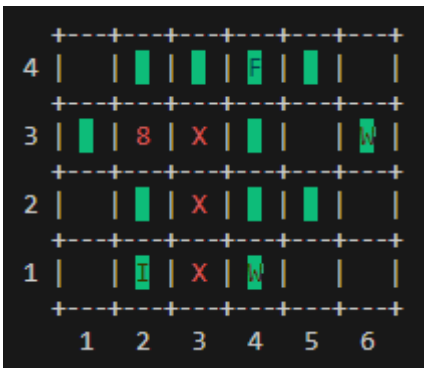
Inicio: (2, 1)
Final: (4, 4)
Trampas: (3, 1), (3, 2), (3, 3)
Peligro: (2, 3) -Nivel 8-
Waypoints: (4, 1), (6, 3)





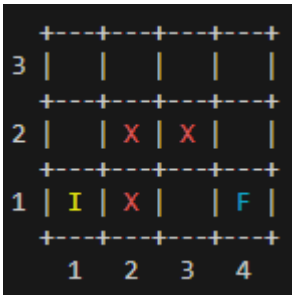


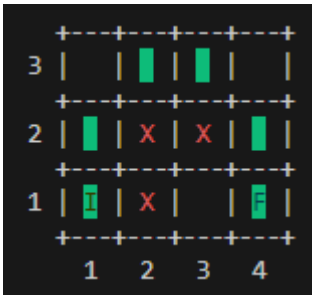




Ejemplo 2: Probando trampas

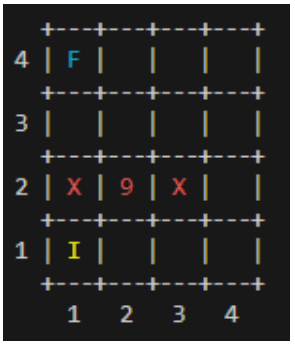
Inicio: (1, 1)
Final: (4, 1)
Trampas: (2, 1), (2, 2), (3, 2)





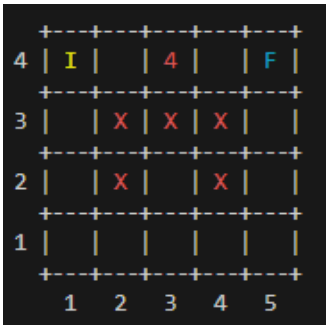
Ejemplo 3: Probando peligros (no toma el camino peligroso)

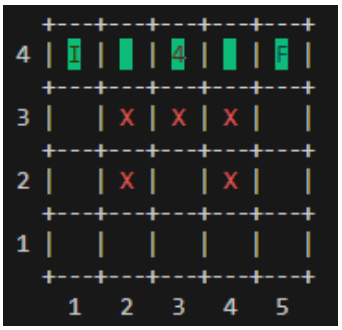
Inicio: (1, 1)
Final: (1, 4)
Trampas: (1, 2), (3, 2)
Peligro: (2, 2) -Nivel 9-



Ejemplo 4: Probando peligros (toma el camino peligroso)

Inicio: (1, 4)
Final: (5, 4)
Trampas: (2, 2), (2, 3), (3, 3)
(4, 3), (4, 2)
Peligro: (3, 4) -Nivel 4-



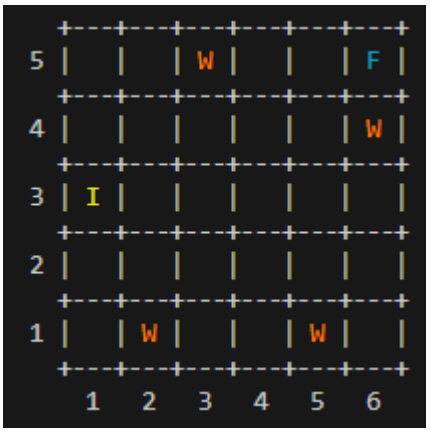


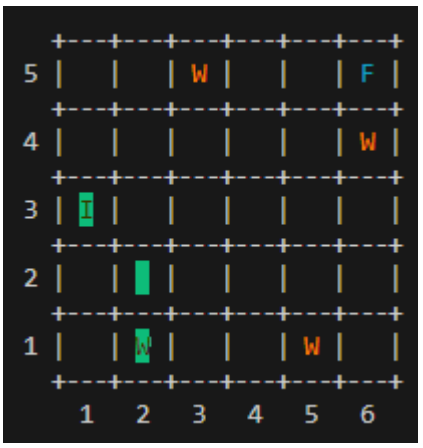
Ejemplo 5: Probando waypoints

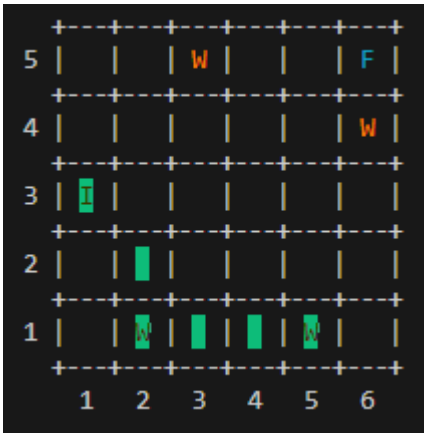
Inicio: (1, 3)

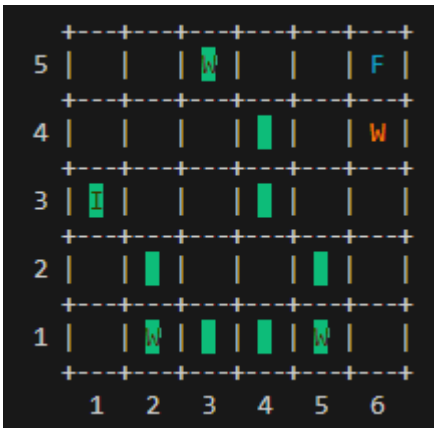
Final: (6, 5)

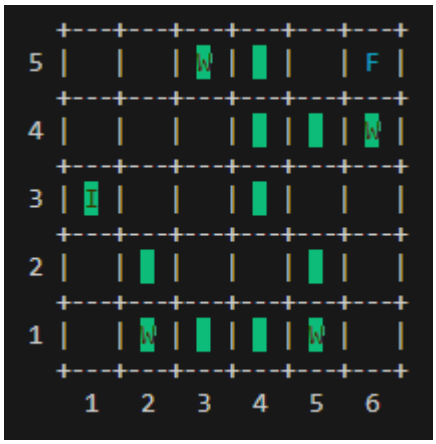
Waypoints: (2, 1), (5, 1), (3, 5), (6, 4)

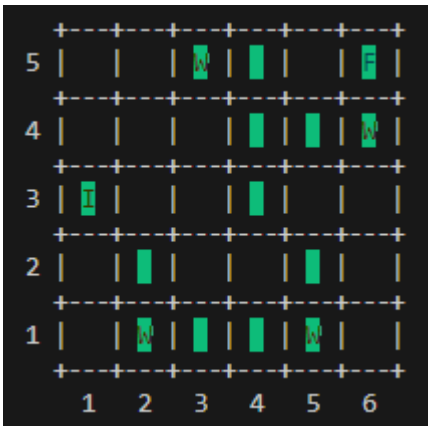












Ejemplo 6: Caso general

Inicio: (2, 2)

Final: (4, 7)

Trampas: (2, 3), (3, 3), (3, 2), (4, 8)
(5, 8), (5, 7), (5, 6), (4, 6)

Peligro: (3, 1) -Nivel 2-, (4, 5) -Nivel 7-

Waypoints: (4, 1), (8, 7), (1, 8)