# Pràctica 1: Cerca informada

Alumno: Florian Alexandru Serb Petrusel
Asignatura: Inteligencia Artificial
Grau: Ingeniería Informática
Professor: David Sánchez
Fecha: 12/03/2025

# ÍNDICE

# 1. Tasques

## 1.1 Formalitzeu el problema definint els estats i els operadors

### 1.1.1 Definició de l'Estat

Un estado está definido por una posición (x, y) en el mapa, su altura h(x,y), el costo acumulado g(n), y el camino recorrido hasta el momento. Además, se incluye información sobre el nodo previo en el camino óptimo y el tiempo acumulado.

Cada estado debe incluir:

- Coordenadas (x, y): Ubicación en la matriz del mapa.
- Altura h(x,y): Valor de la celda en la matriz.
- Costo acumulado g(n): El tiempo total que ha tomado llegar hasta este estado.
- Camino recorrido: Lista de movimientos desde el inicio.
- Nodo previo: El estado anterior en el camino óptimo.
- Tiempo acumulado: El tiempo total acumulado desde el inicio hasta este estado.

### 1.1.2 Definició dels operadors

Los operadores son los movimientos permitidos en el mapa: arriba, abajo, izquierda y derecha. La validez de un operador depende de que la nueva posición esté dentro del mapa y no sea un precipicio. El costo del movimiento depende de la diferencia de altura entre la casilla de origen y la de destino.

Los operadores representan las acciones o movimientos permitidos en el problema. En este caso, los movimientos válidos son:

- **Arriba** (x-1, y)
- **Abajo** (x+1, y)
- **Izquierda** (x, y-1)
- **Derecha** (x, y+1)

**Reglas para aplicar los operadores:**

1. **No salir del mapa:** Las coordenadas (x, y) deben estar dentro de los límites de la matriz.
2. **No moverse a precipicios**: No se puede mover a casillas marcadas como precipicios (valor especial que indica un precipicio).
3. **Calcular el costo del movimiento:**

- Si la diferencia de altura es positiva o 0:

  costo=1+(altura destino−altura origen)

- Si la diferencia de altura es negativa:

  costo=0.5

1. **Nodo previo**: Es importante mencionar que cada estado mantiene una referencia al estado anterior en el camino óptimo. Esto es crucial para reconstruir el camino una vez que se encuentra la solución.
2. **Tiempo acumulado**: El tiempo acumulado es un atributo clave que se utiliza como costo en este problema. Debe recalcularse cada vez que se actualiza el nodo previo.
3. **Cálculo del costo**: El costo del movimiento se calcula en función de la diferencia de altura entre la casilla de origen y la de destino.
4. **Camino recorrido**: El camino recorrido se actualiza cada vez que se establece un nuevo nodo previo, lo que garantiza que siempre se tenga el camino óptimo hasta ese estado.

## 1.2 Definiu 3 heurístiques ben diferenciades

En esta sección, describimos tres heurísticas bien diferenciadas para estimar el costo de llegar desde un estado actual hasta el estado final en el mapa definido en el problema.

### 1.2.1 Heurística Euclidiana amb Diferència d'Altura Màxima

- **Definición**: Esta heurística combina la distancia Euclidiana entre el estado actual y el destino con la diferencia máxima de altura entre ambos estados. La fórmula es:

$$h(n) = \sqrt{(x_{\text{destino}} - x_{\text{actual}})^2 + (y_{\text{destino}} - y_{\text{actual}})^2} + |h_{\text{destino}} - h_{\text{actual}}|$$

- **Justificación**: La distancia Euclidiana proporciona una estimación más precisa de la distancia real en línea recta, mientras que la diferencia de altura máxima penaliza los caminos con grandes desniveles. Esta heurística es útil en terrenos con variaciones significativas de altura.
- **Implementación**: Implementada en la clase HeuristicEuclideanMaxHeight.

### 1.2.2 Heurística Manhattan amb Penalització de Precipicis

- **Definición:** Esta heurística calcula la distancia Manhattan entre el estado actual y el destino, sin considerar la altura. La fórmula es:

$$h(n) = |x_{\text{destino}} - x_{\text{actual}}| + |y_{\text{destino}} - y_{\text{actual}}|$$

- **Justificación**: Se basa únicamente en la distancia en coordenadas X e Y, sin considerar la altura del terreno. Es útil en terrenos planos o cuando la altura no es un factor determinante.
- **Implementación**: Implementada en la clase HeuristicManhattanCliffPenalty.

### 1.2.3 Heurística Manhattan amb Diferència d'Altura Mínima

- **Definición**: Esta heurística combina la distancia Manhattan con la diferencia mínima de altura entre el estado actual y el destino. La fórmula es:

$$h(n) = |x_{\text{destino}} - x_{\text{actual}}| + |y_{\text{destino}} - y_{\text{actual}}| + |h_{\text{destino}} - h_{\text{actual}}|$$

- **Justificación**: La distancia Manhattan proporciona una estimación de la distancia en el plano, mientras que la diferencia de altura mínima penaliza los caminos con desniveles. Esta heurística es útil en terrenos con variaciones moderadas de altura.
- **Implementación**: Implementada en la clase HeuristicManhattanMinHeight.

## 1.3 Per cada heurística, indiqueu si són o no admissibles respecte al temps

Una heurística es admisible si nunca sobreestima el coste real del camino más corto en términos de tiempo. Analizamos cada heurística:

### 1.3.1 Heurística Euclidiana amb Diferència d'Altura Màxima

- **Función heurística:**

$$h(n) = \sqrt{(x_{\text{destino}} - x_{\text{actual}})^2 + (y_{\text{destino}} - y_{\text{actual}})^2} + |h_{\text{destino}} - h_{\text{actual}}|$$

- **Admisibilidad**: No es admisible
  La distancia Euclidiana subestima el coste real, pero la diferencia de altura máxima puede sobrestimar el coste en casos donde el camino óptimo incluya descensos que reduzcan el tiempo.
  En terrenos con grandes desniveles, esta heurística puede sugerir un coste mayor que el real.

### 1.3.2 Heurística Manhattan amb Penalització de Precipicis

- **Función heurística**:

$$h(n) = |x_{\text{destino}} - x_{\text{actual}}| + |y_{\text{destino}} - y_{\text{actual}}|$$

- **Admisibilidad**: No es admisible
  Aunque la distancia Manhattan no sobrestima el coste en términos de distancia, esta heurística no tiene en cuenta la altura, lo que puede llevar a subestimar el coste real en terrenos con desniveles.
  En casos donde el camino óptimo requiere movimientos con diferencias de altura significativas, esta heurística no es admisible.

### 1.3.3 Heurística Manhattan amb Diferència d'Altura Mínima

- **Función heurística**:

$$h(n) = |x_{\text{destino}} - x_{\text{actual}}| + |y_{\text{destino}} - y_{\text{actual}}| + |h_{\text{destino}} - h_{\text{actual}}|$$

- **Admisibilidad**: Sí, es admisible
  La distancia Manhattan nunca sobrestima el coste real en términos de distancia, y la diferencia de altura mínima tampoco sobrestima el coste real en términos de tiempo, ya que solo considera la diferencia de altura entre el estado actual y el destino.
  Esta heurística no sobrestima el coste real del camino más corto, ya que siempre considera el mínimo coste posible en términos de distancia y altura.

Resumen:

| Heurística | ¿Es admisible respecto al tiempo? | Razón |
|---|---|---|
| Euclidiana con Diferencia de Altura Máxima | No | Puede sobrestimar el coste en terrenos con grandes desniveles. |
| Manhattan con Penalización de Precipicios | No | No tiene en cuenta la altura, lo que puede llevar a subestimaciones. |
| Manhattan con Diferencia de Altura Mínima | Sí | No sobrestima el coste real, ya que considera la distancia y la altura mínima. |

La heurística **Manhattan** con Diferencia de Altura Mínima es la única que garantiza ser admisible en todos los casos, ya que no sobrestima el coste real del camino más corto.

Las otras dos heurísticas no son admisibles debido a que pueden sobrestimar o subestimar el coste real en función de las diferencias de altura en el terreno.

## 1.4 Implementeu en Java els algorismes de cerca Best-first i A*

El algoritmo **Best-First** es un algoritmo de búsqueda informada que utiliza una heurística para guiar la exploración del espacio de estados. En lugar de explorar todos los estados posibles, este algoritmo selecciona en cada paso el estado que parece más prometedor según la heurística, es decir, el que tiene el menor valor heurístico.

**Fórmula clave**:
El algoritmo selecciona el estado con el menor valor heurístico h(n), donde h(n) es el valor calculado por la heurística para el estado *n*.

**Implementación en el código**:

- Se utiliza una lista de estados pendientes (pending) y una lista de estados tratados (treated).
- En cada iteración, se selecciona el estado con el menor valor heurístico de la lista pending.
- Se exploran los vecinos del estado actual y se añaden a la lista pending si no han sido tratados previamente.
- La lista pending se ordena en cada iteración según el valor heurístico.

```
public class BestFirst extends Algorithm {
    public void bestFirst(State[][] map, State ini, State end, Heuristic h) {
        // Lista para manejar los estados pendientes
        ArrayList<State> pending = new ArrayList<>();
```

```
        // Lista para almacenar los estados ya tratados
        ArrayList<State> treated = new ArrayList<>();

        pending.add(ini); // Agrega el estado inicial a la lista de pendientes
        boolean found = false;

        // Bucle principal de búsqueda
        while (!found && !pending.isEmpty()) {
            // Extrae el estado con el menor valor heurístico (el primero en la
lista ordenada)
            State st = pending.get(0);
            pending.remove(0);

            // Si se alcanza el estado objetivo, se finaliza la búsqueda
            if (st.getPosition().cmp(end.getPosition())) {
                found = true;
                st.setTime(); // Calcula el tiempo acumulado
                printResults("Best First", st, treated, h, map, found);
                st.resetTime(); // Reinicia el tiempo para futuras búsquedas
                return;
            }

            // Explora los estados vecinos del estado actual
            for (State neighbour : succesors(st, map)) {
                if (!treated.contains(neighbour) && !pending.contains(neighbour)) {
                    pending.add(neighbour); // Agrega el vecino a la lista de
pendientes
                }
            }

            // Ordena la lista de pendientes según la heurística
            Collections.sort(pending, Comparator.comparingDouble(state ->
h.checkStates(state, end)));

            treated.add(st); // Marca el estado actual como tratado
        }

        // Si no se encuentra solución, imprimir los resultados con el estado
inicial
        printResults("Best First", ini, treated, h, map, found);
    }
}
```

**Ejemplo de uso**:

- Si utilizamos la heurística **Manhattan con Diferencia de Altura Mínima**, el algoritmo seleccionará en cada paso el estado que minimice la

distancia Manhattan más la diferencia de altura mínima con respecto al estado final.

El algoritmo **A*** es una extensión del algoritmo **Best-First** que combina el costo real desde el estado inicial (g(n)) con una estimación heurística del costo hasta el estado final (h(n)). La función de evaluación f(n) se define como:

$$f(n)=g(n)+h(n)$$

Donde:

- g(n): Costo acumulado desde el estado inicial hasta el estado actual.
- h(n): Estimación heurística del costo desde el estado actual hasta el estado final.

**Fórmula clave**:

f(n)=g(n)+h(n)

**Implementación en el código**:

- Se utiliza una cola de prioridad (pending) para mantener los estados ordenados por su valor f(n).
- Se utiliza un mapa (bestCosts) para almacenar el mejor costo encontrado para cada estado.
- En cada iteración, se selecciona el estado con el menor valor f(n) de la cola de prioridad.
- Se exploran los vecinos del estado actual y se actualizan sus costos si se encuentra un camino más óptimo.

```
public class Astar extends Algorithm {
    public void astar(State[][] map, State ini, State end, Heuristic heuristic) {
        // Cola de prioridad para manejar los estados pendientes, ordenados por el costo estimado F
        PriorityQueue<State> pending = new PriorityQueue<>(Comparator.comparingDouble(State::getF));
        // Mapa para almacenar los mejores costos encontrados para cada estado
        Map<State, Double> bestCosts = new HashMap<>();
        // Inicialización del estado inicial
        ini.setFirstF(); // Calcula el valor inicial de F
        ini.setTime(); // Establece el tiempo inicial
        bestCosts.put(ini, ini.getTime()); // Guarda el costo inicial en el mapa de mejores costos
        pending.add(ini); // Agrega el estado inicial a la cola de prioridad

        boolean found = false; // Bandera para indicar si se ha encontrado el objetivo
```

```
        // Bucle principal de búsqueda
        while (!found && !pending.isEmpty()) {
            State st = pending.poll(); // Extrae el estado con el menor costo
estimado F

            // Si se alcanza el estado objetivo, se finaliza la búsqueda
            if (st.getPosition().cmp(end.getPosition())) {
                found = true;
                printResults("A*", st, new ArrayList<>(bestCosts.keySet()),
heuristic, map, found);
                break;
            }

            // Explora los estados vecinos del estado actual
            for (State neighbour : succesors(st, map)) {
                double newCost = st.getTime() + neighbour.getCost(); // Calcula el
nuevo costo acumulado

                // Si el vecino no ha sido visitado o se encuentra un mejor costo,
se actualiza
                if (!bestCosts.containsKey(neighbour) || newCost <
bestCosts.get(neighbour)) {
                    neighbour.setPrevious(st); // Establece el estado anterior para
reconstruir el camino
                    neighbour.setTime(); // Actualiza el tiempo del estado vecino
                    neighbour.setF(newCost, heuristic, end); // Calcula el nuevo
valor F usando la heurística
                    bestCosts.put(neighbour, newCost); // Guarda el nuevo costo
en el mapa de mejores costos
                    pending.add(neighbour); // Añade el vecino a la cola de
prioridad
                }
            }
        }
        // Si no se encuentra solución, imprimir los resultados con el estado
inicial
        if (!found) {
            printResults("A*", ini, new ArrayList<>(bestCosts.keySet()), heuristic,
map, found);
        }
    }
}
```

**Ejemplo de uso**:

- Si utilizamos la heurística **Euclidiana con Diferencia de Altura Máxima**, el algoritmo calculará f(n) como la suma del costo

acumulado g(n) y la distancia Euclidiana más la diferencia de altura máxima.

**Conclusión**

- **Best-First** es más rápido, pero no garantiza la optimización, ya que solo considera la heurística.
- **A*** es más lento pero, garantiza la solución óptima si la heurística es admisible.
- Las heurísticas implementadas permiten guiar la búsqueda de manera eficiente, aunque su admisibilidad varía según el caso.

### 1.4.1 El mapa d'entrada s'ha de llegir de fitxer de text

El mapa se carga desde un archivo de texto que sigue un formato específico. El archivo contiene:

1. **Lectura de las dimensiones del mapa**: Se lee la primera línea del archivo para obtener el número de filas y columnas.
2. **Lectura de las posiciones de inicio y fin**: Se leen las siguientes dos líneas para obtener las coordenadas de la posición inicial y final.
3. **Lectura de los valores de las celdas**: Se lee el resto del archivo línea por línea, y se procesa cada valor para crear los estados correspondientes en la matriz del mapa. Las casillas con 'X' se marcan como no válidas con el valor -999.
4. **Retorno del mapa cargado**: Se devuelve un objeto MapData que contiene la matriz del mapa, la posición de inicio y la posición de fin.

```
public class MapLoader {
    private static final int INVALID_VALUE = -999; // Valor que indica una
casilla no válida (representada como 'X' en  el archivo)
    public static MapData loadMapState(String filePath) throws IOException {
        BufferedReader reader = new BufferedReader(new
FileReader(filePath)); // Abrir el archivo para lectura
        String lineContent; // Variable para almacenar cada línea leída
        String[] lineArray; // Arreglo para dividir el contenido de las líneas

        // Leer las dimensiones del mapa (número de filas y columnas)
        lineArray = reader.readLine().split(" ");
        int numRows = Integer.parseInt(lineArray[0]); // Número de filas del
mapa
        int numCols = Integer.parseInt(lineArray[1]); // Número de columnas del
mapa
```

```
        // Inicializar la matriz del mapa con las dimensiones especificadas
        State[][] map = new State[numRows][numCols];
        // Leer la posición de inicio (coordenadas de la celda inicial)
        lineArray = reader.readLine().split(" ");
        Position startPosition = new Position(Integer.parseInt(lineArray[0]),
Integer.parseInt(lineArray[1]));
        // Leer la posición de destino (coordenadas de la celda final)
        lineArray = reader.readLine().split(" ");
        Position endPosition = new Position(Integer.parseInt(lineArray[0]),
Integer.parseInt(lineArray[1]));
        // Leer el contenido del mapa y llenar la matriz de estados
        int rowIndex = 0; // Índice de fila en la matriz
        while ((lineContent = reader.readLine()) != null) { // Leer línea a línea
hasta el final del archivo
            lineArray = lineContent.split(" "); // Separar los valores de la línea por
espacios
            for (int colIndex = 0; colIndex < lineArray.length; colIndex++) {
                // Si la casilla contiene 'X', se asigna el valor de casilla no válida
                int value = lineArray[colIndex].equals("X") ? INVALID_VALUE :
Integer.parseInt(lineArray[colIndex]);
                // Se crea un estado en la posición correspondiente con el valor
obtenido
                map[rowIndex][colIndex] = new State(value, new
Position(rowIndex, colIndex));
            }
            rowIndex++; // Pasar a la siguiente fila
        }
        reader.close(); // Cerrar el archivo después de la lectura
        // Retornar el mapa cargado junto con las posiciones de inicio y fin
        return new MapData(map, startPosition, endPosition);
    }
}
```

Así es como se verá en el terminal, después de cargar el mapa:

- - Donde la S indica START él inició de la búsqueda.
- - Donde E indica END el final de la búsqueda.

```
MAPA BASE
|S 1 2 3 3 X 0 1 2 2|
|2 1 2 4 3 2 1 3 3 3|
|2 2 X 4 5 3 2 X 4 4|
|3 3 X 4 6 4 2 3 3 3|
|2 2 3 3 5 3 3 2 3 3|
|2 1 1 3 4 X 2 2 3 4|
|2 0 0 1 2 1 1 1 1 X|
|X 1 0 1 2 0 2 3 2 3|
|2 2 2 1 1 0 2 3 3 4|
|4 3 2 2 X 1 1 2 4 E|
```

## 1.5 Proveu ambdós algorismes i les 3 heurístiques per a diferents problemes

### 1.5.1 Algoritme Best First

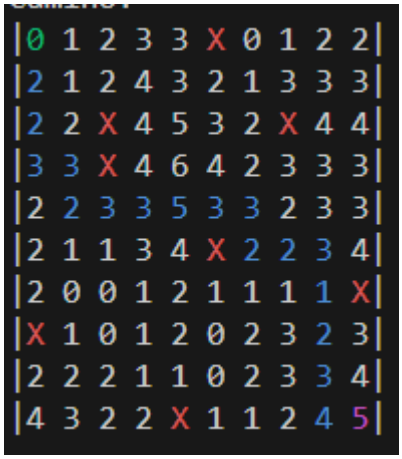
Mapa1 con la configuración del enunciado:

**Ejecutando Best-First con HeuristicEuclideanMinHeight**
Número de nodos tratados: 22
Coste: 27.0
Camino:

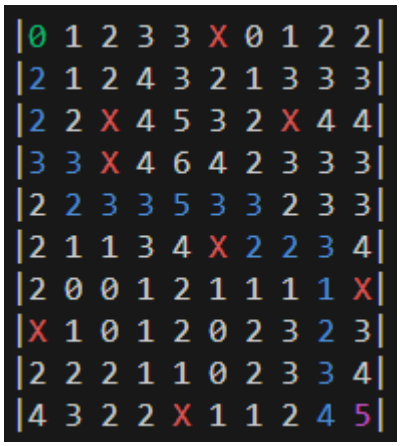
```
|0 1 2 3 3 X 0 1 2 2|
|2 1 2 4 3 2 1 3 3 3|
|2 2 X 4 5 3 2 X 4 4|
|3 3 X 4 6 4 2 3 3 3|
|2 2 3 3 5 3 3 2 3 3|
|2 1 1 3 4 X 2 2 3 4|
|2 0 0 1 2 1 1 1 1 X|
|X 1 0 1 2 0 2 3 2 3|
|2 2 2 1 1 0 2 3 3 4|
|4 3 2 2 X 1 1 2 4 5|
```

**Ejecutando Best-First con HeuristicEuclideanMaxHeight**

Número de nodos tratados: 22
Coste: 27.0
Camino:




**Ejecutando Best-First con HeuristicManhattanCliffPenalty**
Número de nodos tratados: 20
Coste: 27.0
Camino:




**Conclusión**

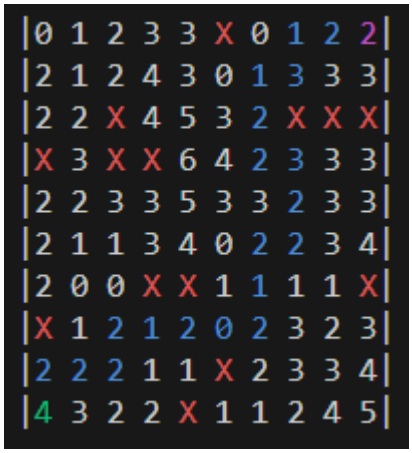
El algoritmo **Best-First** no considera el costo real acumulado (g(n)) para llegar a un estado, sino que se guía únicamente por la heurística (h(n)), que es una estimación del costo restante hasta el objetivo. Esto está provocando que, en el Mapa1, todas las heurísticas guíen al algoritmo hacia el mismo camino, resultando en el mismo costo final.

La razón es que el **Mapa1** tiene un único camino viable o las diferencias en las heurísticas no son lo suficientemente significativas como para influir en la selección de estados. Por ejemplo, al no haber precipicios y tener diferencias de altura mínimas, las
heurísticas EuclideanMinHeight, EuclideanMaxHeight y ManhattanCliffPenalty e stán convergiendo hacia el mismo camino, lo que explica por qué el costo final es idéntico en los tres casos.

Mapa2 con la configuración ajustada:

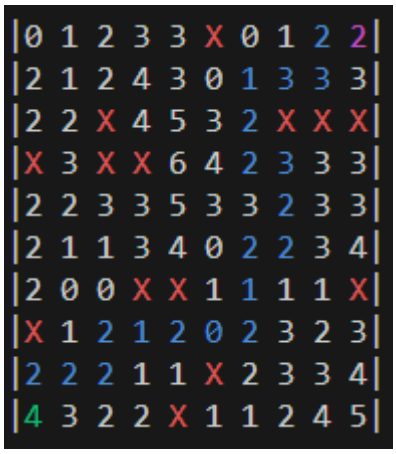
**Ejecutando Best-First con HeuristicEuclideanMinHeight**

Número de nodos tratados: 24
Coste: 24.5
Camino:




**Ejecutando Best-First con HeuristicEuclideanMaxHeight**

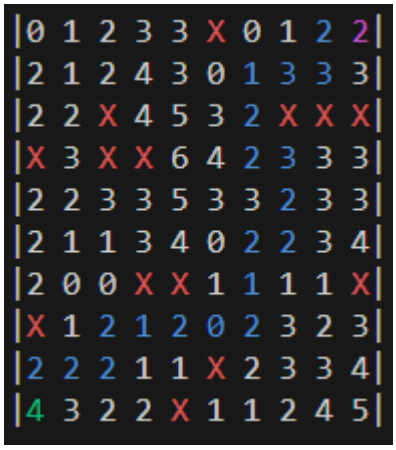
Número de nodos tratados: 22
Coste: 23.5
Camino:




**Ejecutando Best-First con HeuristicManhattanCliffPenalty**

Número de nodos tratados: 20
Coste: 23.5

Camino:




El algoritmo **Best-First** se guía únicamente por la heurística (h(n)) para seleccionar el siguiente estado a explorar, sin tener en cuenta el costo acumulado (g(n)). En el **Mapa2**, los resultados muestran que los costes y los caminos varían ligeramente entre las diferentes heurísticas.

**Diferencias en las heurísticas**:

- **HeuristicEuclideanMinHeight**: Combina la distancia Euclidiana con la diferencia de altura mínima. Esta heurística tiende a priorizar caminos con menores desniveles, lo que puede resultar en un coste ligeramente mayor (**24.5**) debido a que evita subidas pronunciadas.
- **HeuristicEuclideanMaxHeight**: Combina la distancia Euclidiana con la diferencia de altura máxima. Esta heurística puede priorizar caminos más directos, incluso si implican subidas más pronunciadas, lo que resulta en un coste menor (**23.5**).
- **HeuristicManhattanCliffPenalty**: Utiliza la distancia Manhattan sin considerar la altura. Esta heurística tiende a ser más agresiva en la búsqueda de caminos directos, lo que también resulta en un coste menor (**23.5**).

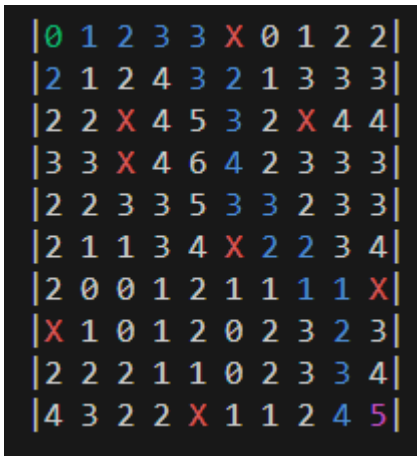
En el **Mapa2**, el algoritmo **Best-First** muestra resultados ligeramente diferentes en función de la heurística utilizada. Esto se debe a que el mapa tiene una configuración que permite que las heurísticas influyan en la selección de estados, resultando en caminos y costes diferentes.
La **HeuristicEuclideanMinHeight** prioriza caminos con menores desniveles, mientras que las otras dos heurísticas priorizan caminos más directos, incluso con subidas pronunciadas. Esto explica por qué los costes y los caminos varían ligeramente entre las heurísticas.

### 1.5.2 Algoritme A*

Mapa1 con la configuración del enunciado:

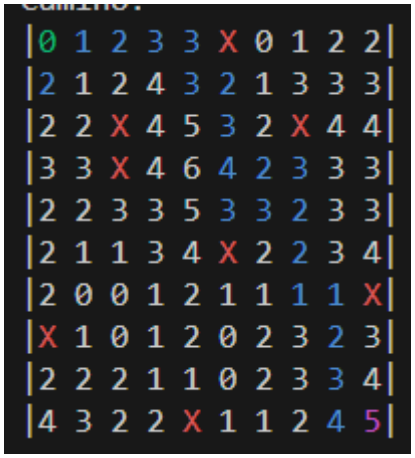
**Ejecutando A* con HeuristicManhattanMinHeight**

Número de nodos tratados: 92
Coste: 28.5
Camino:




**Ejecutando A* con HeuristicEuclideanMaxHeight**

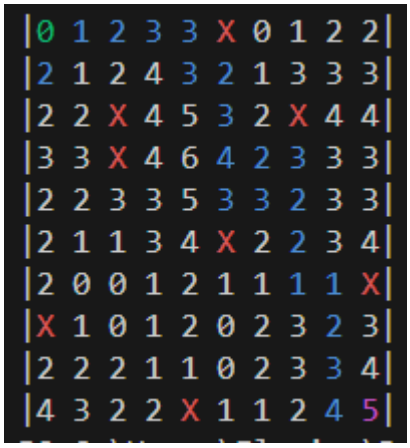
Número de nodos tratados: 92
Coste: 31.0
Camino:




**Ejecutando A* con HeuristicManhattanCliffPenalty**

Número de nodos tratados: 92
Coste: 31.0

Camino:




La heurística **HeuristicEuclideanMinHeight** es admisible, lo que significa que nunca sobrestima el costo real del camino más corto. Esto permite que el algoritmo **A*** encuentre la solución óptima con un coste de **28.5**, que es mejor que los costes obtenidos con las heurísticas no admisibles (**31.0**).

En resumen:

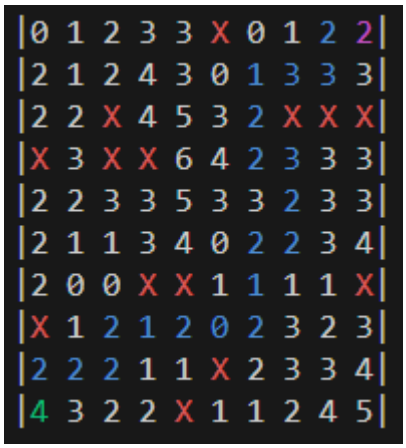
**Heurística admisible (HeuristicEuclideanMinHeight)**: Garantiza la optimización del camino, resultando en un coste menor (**28.5**).

**Heurísticas no admisibles (HeuristicEuclideanMaxHeighty** y **HeuristicManhattanCliffPenalty)**: No garantizan la optimización, lo que lleva a costes más altos (**31.0**).

Mapa2 con la configuración ajustada:

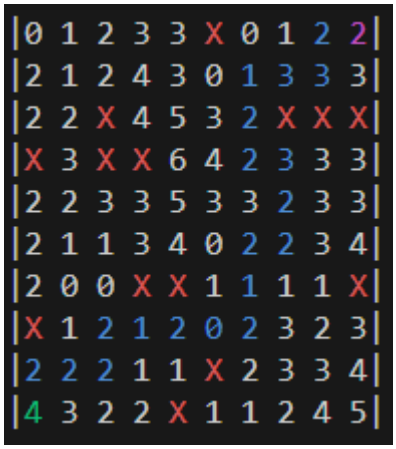
**Ejecutando A* con HeuristicManhattanMinHeight**

Número de nodos tratados: 85
Coste: 23.5
Camino:



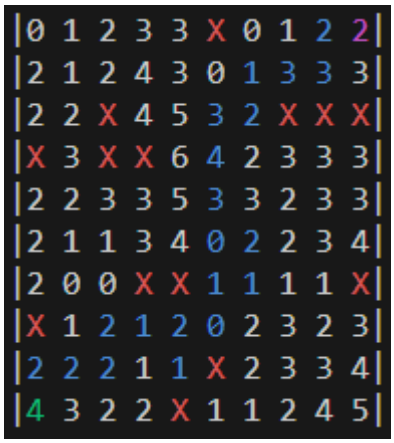
**Ejecutando A* con HeuristicEuclideanMaxHeight**

Número de nodos tratados: 85
Coste: 26.0
Camino:




Ejecutando A* con HeuristicManhattanCliffPenalty

Número de nodos tratados: 85
Coste: 25.5
Camino:



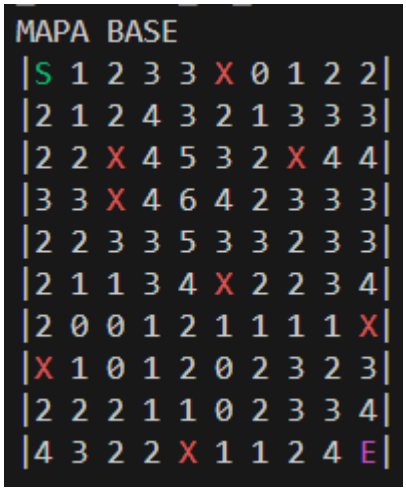

La heurística **ManhattanMinHeight** es admisible, lo que significa que nunca sobrestima el costo real del camino más corto. Esto permite que el algoritmo **A*** encuentre la solución óptima con un coste de **23.5**, que es mejor que los costes obtenidos con las heurísticas no admisibles (**26.0** y **25.5**).

Además, el **Mapa2** es más variado y complejo que el **Mapa1**, lo que permite que las heurísticas muestren su impacto real en la selección de estados y en el coste final. Esto demuestra que, en mapas con múltiples caminos viables y variaciones en las alturas, las heurísticas pueden influir significativamente en los resultados.

## 1.6. Raonament Algoritme Hill climbing

El algoritmo **Hill Climbing** es un algoritmo de búsqueda local que intenta optimizar una función objetivo (en este caso, minimizar el coste del camino) moviéndose iterativamente hacia estados vecinos que mejoren el valor de la función heurística. Sin embargo, tiene limitaciones importantes, como la posibilidad de quedar atrapado en máximos locales o mesetas, lo que puede impedirle encontrar una solución óptima o incluso cualquier solución.

Mapa1:




**Heurística ManhattanMinHeight:**

**Comportamiento**: Esta heurística es admisible y proporciona una estimación precisa del coste real. **Hill Climbing** podría encontrar una solución en este mapa, ya que la heurística guía el algoritmo hacia el objetivo de manera consistente.

**Problemas**: Sin embargo, **Hill Climbing** no tiene en cuenta el coste acumulado (g(n)$g(n)$), por lo que podría quedar atrapado en un máximo local si el camino óptimo requiere pasar por estados con valores heurísticos peores antes de mejorar. En este mapa, es probable que encuentre una solución, pero no necesariamente la óptima.

**Heurística EuclideanMaxHeight:**

**Comportamiento**: Esta heurística no es admisible y puede sobrestimar el coste real. **Hill Climbing** podría quedar atrapado en un máximo local, ya que la heurística no siempre guía correctamente hacia el objetivo.

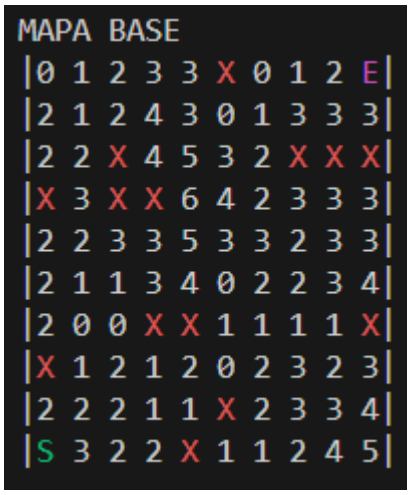
**Problemas**: En zonas con grandes desniveles, el algoritmo podría preferir caminos que parecen prometedores según la heurística, pero que en realidad son subóptimos. Es posible que no encuentre una solución en este mapa.

**Heurística ManhattanCliffPenalty:**

**Comportamiento**: Esta heurística no considera la altura, por lo que puede subestimar el coste real en terrenos con desniveles. **Hill Climbing** podría encontrar una solución, pero no garantiza que sea óptima.

**Problemas**: Al no tener en cuenta la altura, el algoritmo podría quedar atrapado en un máximo local o en una meseta, especialmente en zonas con precipicios.

Mapa2:




**Heurística ManhattanMinHeight:**

**Comportamiento**: Al ser admisible, esta heurística guía bien el algoritmo hacia el objetivo. En este mapa, **Hill Climbing** podría encontrar una solución, aunque no garantiza que sea la óptima debido a la posibilidad de quedar atrapado en un máximo local.

**Problemas**: Aunque la heurística es admisible, **Hill Climbing** no tiene una visión global del problema, por lo que podría no encontrar la solución óptima.

**Heurística EuclideanMaxHeight:**

**Comportamiento**: Esta heurística no es admisible y puede sobrestimar el coste real. En este mapa, **Hill Climbing** podría quedar atrapado en un máximo local, especialmente en zonas con grandes desniveles.

**Problemas**: Es probable que no encuentre una solución en este mapa, ya que la heurística no guía correctamente hacia el objetivo en terrenos complejos.

**Heurística ManhattanCliffPenalty:**

**Comportamiento**: Esta heurística no considera la altura, por lo que puede subestimar el coste real en terrenos con desniveles. **Hill Climbing** podría encontrar una solución, pero no garantiza que sea óptima.

**Problemas**: Al no tener en cuenta la altura, el algoritmo podría quedar atrapado en un máximo local o en una meseta, especialmente en zonas con precipicios.

**Conclusión general**

**Hill Climbing** es un algoritmo que depende en gran medida de la heurística utilizada y de la estructura del mapa. En general, es más probable que encuentre una solución cuando la heurística es admisible y el mapa tiene una estructura simple (como el **Mapa1**).

En el **Mapa1**, **Hill Climbing** podría encontrar una solución con la heurística **ManhattanMinHeight**, pero es menos probable que lo haga con las heurísticas no admisibles (**EuclideanMaxHeight** y **ManhattanCliffPenalty**).

En el **Mapa2**, que es más complejo y tiene más variaciones en las alturas y precipicios, **Hill Climbing** tiene más dificultades para encontrar una solución, especialmente con heurísticas no admisibles.

En resumen, **Hill Climbing** no garantiza encontrar una solución óptima o incluso cualquier solución, especialmente en mapas complejos y con heurísticas no admisibles.

# 2. Conclusió

Esta práctica ha permitido explorar y comparar diferentes enfoques para resolver problemas de búsqueda en entornos con restricciones. Hemos demostrado que la formalización del problema, la elección de la heurística y el algoritmo de búsqueda son factores clave que influyen en la eficiencia y optimización de las soluciones. Mientras que Best-First es adecuado para problemas donde la velocidad es prioritaria, A* es la mejor opción cuando se requiere garantía de optimización. Por otro lado, Hill Climbing es útil en contextos limitados, pero su falta de visión global lo hace menos confiable en problemas complejos. Esta experiencia ha reforzado la importancia de comprender las características del problema y seleccionar las herramientas adecuadas para abordarlo de manera efectiva.