**UNIVERSIDAD FRANCISCO DE VITORIA**

**ESCUELA POLITÉCNICA SUPERIOR**

# Ingeniería del Conocimiento

# Memoria de la Práctica de Fin de Curso – Exploración del laberinto

**Juan Carlos García Ventura**

**14/05/2025**

## Tabla de contenido



## Tabla de ilustraciones

# 1. Introducción

En esta práctica vamos a abordar la resolución de un problema donde tenemos que encontrar la salida de un laberinto usando cuatro técnicas: BFS, DFS, Voraz y A*.

El laberinto se va a estructurar como una matriz binaria donde cada celda libre (0) o bloqueada (1) corresponde a un estado, y las acciones permitidas son desplazamientos ortogonales. En la implementación en MATLAB se puede distinguir:

1. La gestión de la frontera
2. La expansión de nodos y generación de sucesores
3. El test objetivo
4. La reconstrucción y visualización de la ruta
5. La definición de la heuristica, que en este problema va a ser la distancia de Manhattan. Además, se va a demostrar que esta heurística es admisible y consistente, algo necesario para asegurar que A* es óptima.

La memoria va a estar estructurada en 3 apartados principales:

1. Definición formal del problema, donde se especifican espacios de estados, acciones función de coste y test objetivo.
2. Descripción de las funciones implementadas, donde se va a detallar cada función en el script y el uso de las funciones Búsqueda-Árbol y Búsqueda-Grafo.
3. Justificación de la heurística

# 2. Definición formal del problema

## 2.1. Espacio de estados

- Cada estado s $\epsilon$ S es una posición (i, j) en el laberinto, con
$$1 \leq i \leq n_f, 1 \leq j \leq n_c$$
- El laberinto se modela como una matriz $L \in \{0,1\}^{n_f \times n_c}$, donde:
    - $L(i,j) = 0$ es una celda libre
    - $L(i,j) = 1$ es una pared

## 2.2. Estado inicial y objetivo

- Estado inicial $s_0 = (1,1)$, esto es la esquina superior izquierda
- Estado objetivo $s_g = (n_{f,} n_c)$, esto es la esquina inferior derecha

## 2.3. Acciones y transición

Para $s = (i, j)$, el conjunto de acciones aplicables es: A(s) = {arriba, abajo, izquierda, derecha}

La función de transición $T(s, a)$ devuelve el vecino si está dentro de límites y es libre (L = 0), en caso contrario no genera sucesor:

- $T\big((i,j), arriba\big) = (i-1, j)\ si\ i > 1\ y\ L(i-1, j) = 0$
- $T\big((i,j), abajo\big) = (i+1, j)\ si\ i < n_f\ y\ L(i+1, j) = 0$
- $T\big((i,j), izquierda\big) = (i, j-1)\ si\ j > 1\ y\ L(i, j-1) = 0$
- $T\big((i,j), derecha\big) = (i, j+1)\ si\ j < n_c\ y\ L(i, j+1) = 0$

## 2.4. Función de coste

Cada acción tiene una constante $c(s, a, s') = 1$.El coste de un camino cualquiera $\alpha = (s_0, \dots, s_k)$ es $g(\alpha) = \sum_{t=1}^{k} c\big(s_{t-1,} a_{t,} s_t\big) = k$. Donde $s_k$es el estado final (queremos que el estado final sea el objetivo)

## 2.5. Test objetivo

$$testObjetivo(s) = \begin{cases} true, & s = s_g \\ false, & en\ otro\ caso \end{cases}$$

# 3. Descripción de las funciones implementadas

## 3.1. Funciones script

1. hacerCola: crea una cola vacía
2. vacia: comprueba si la pilo o cola está vacía
3. primeroCola: devuelve el primer elemento
4. borrarPrimero: borra el primer elemento de la pila o cola
5. inserta: inserta un elemento al final de la cola
6. insertarTodo: inserta todos los elementos de una lista al final de la cola
7. expandir: genera sucesores válidos aplicando las acciones al estado actual
8. moverArriba: mueve hacia arriba si la celda destino está libre
9. moverAbajo: mueve hacia abajo si la celda destino está libre
10. moverIzquierda: mueve hacia la izquierda si la celda destino está libre
11. moverDerecha: mueve hacia la derecha si la celda destino está libre
12. testObjetivo: comprueba si un estado es el objetivo
13. reconstruirRuta: reconstruye la ruta óptima a partir del mapa de padres
14. mostrarLaberinto: dibuja el laberinto
15. solucion: muestra paso a paso la ruta hallada

16. busquedaAnchura: implementa BFS
17. busquedaProfundidad: implementa DFS
18. primeroElMejor: búsqueda voraz implementando la heurística de Manhattan
19. aEstrella: búsqueda A* con función f(n) = g(n) + h(n)
20. heuristica: calcula la heurística, que en este caso es la distancia de Manhattan

## 3.2. Función Búsqueda-Árboles

**función** BÚSQUEDA-ÁRBOLES(*problema*,*frontera*) **devuelve** una solución o fallo
  *frontera* ← INSERTA(HACER-NODO(ESTADO-INICIAL[*problema*]), *frontera*)
  **hacer bucle**
    **si** VACIA?(*frontera*) **entonces devolver** fallo.
    *nodo* ← BORRAR-PRIMERO(*frontera*)
    **si** TEST-OBJETIVO[*problema*] aplicado al ESTADO[*nodo*] es cierto
      **entonces devolver** SOLUCIÓN(*nodo*)
    *frontera* ← INSERTAR-TODO(EXPANDIR(*nodo*,*problema*),*frontera*)

**función** EXPANDIR(*nodo*,*problema*) **devuelve** un conjunto de nodos
  *sucesores* ← conjunto vacío
  **para cada** (*acción*,*resultado*) **en** SUCESOR-FN[*problema*](ESTADO[*nodo*]) **hacer**
    *s* ← un nuevo NODO
    ESTADO[*s*] ← *resultado*
    NODO-PADRE[*s*] ← *nodo*
    ACCIÓN[*s*] ← *acción*
    COSTO-CAMINO[*s*] ← COSTO-CAMINO[*nodo*] + COSTO-INDIVIDUAL(*nodo*,*acción*,*s*)
    PROFUNDIDAD[*s*] ← PROFUNDIDAD[*nodo*] + 1
    añadir *s* a *sucesores*
  **devolver** *sucesores*

**Figura 3.9** Algoritmo general de búsqueda en árboles. (Notemos que el argumento *frontera* puede ser una cola vacía, y el tipo de cola afectará al orden de la búsqueda.) La función SOLUCIÓN devuelve la secuencia de acciones obtenida de la forma punteros al padre hasta la raíz.

*Ilustración 1 - Figura 3.9 (Función Búsqueda-Árboles)*

- HACER-NODO: creamos el estado inicial *inicio = [1,1]*
- frontera: nuestra cola (BFS) o pila (DFS), con las operaciones: hacerCola, inserta, vacia, primeroCola, borrarPrimero.
- EXPANDIR: *expandir(L, s),* que genera los sucesores (llama a las funciones moverArriba, moverAbajo, moverDerecha, moverIzquierda).
- TEST-OBJETIVO: llama a la función *testObjetivo(L, s)*
- INSERTAR-TODO*: insertarTodo(fr, sucesores)* para añadir en bloque todos los sucesores

## 3.3. Función Búsqueda-Grafos

```
función BÚSQUEDA-GRAFOS(problema,frontera) devuelve una solución, o fallo

  cerrado ← conjunto vacío
  frontera ← INSERTAR(HACER-NODO(ESTADO-INICIAL[problema]),frontera)
  bucle hacer
    si VACIA?(frontera) entonces devolver fallo
    nodo ← BORRAR-PRIMERO(frontera)
    si TEST-OBJETIVO[problema](ESTADO[nodo]) entonces devolver SOLUCIÓN(nodo)
    si ESTADO[nodo] no está en cerrado entonces
      añadir ESTADO[nodo] a cerrado
      frontera ← INSERTAR-TODO(EXPANDIR(nodo,problema),frontera)
```

**Figura 3.19** Algoritmo general de búsqueda en grafos. El conjunto cerrado puede implementarse como una tabla *hash* para permitir la comprobación eficiente de estados repetidos. Este algoritmo supone que el primer camino a un estado $s$ es el más barato (*véase* el texto).

*Ilustración 2 - Figura 3.19 (Función Búsqueda-Grafos)*

- Implementamos la estructura "cerrado" de dos formas:
  - BFS/DFS: un array booleano *visitado* que marca cada celda
  - A*: usamos tanto gMap como el mismo array *visitado* implícito al no reinsertar nodos con peor coste.
- La frontera en los algoritmos informados es un vector de estructuras con campos g y/o h, del que extraeremos el primer elemento según el criterio (menor h, o menor g+h).
- Tras extraer un nodo, comprobamos *testObjetivo*, marcamos en cerrado y luego hacemos *insertarTodo* de los sucesores recién generados con su coste calculado.

# 4. Heurística: admisibilidad y consistencia

Se usa la distancia de Manhattan desde $s = (i, j)$ hasta el objetivo $(n_f, n_c)$, siendo la distancia $h(i, j) = |n_f - i| + |n_c - j|$.

## 4.1. Admisibilidad

Manhattan cuenta con el mínimo número de movimientos sin considerar obstáculos. Por definición, nunca sobreestima el coste real $g^*(s \to s_g)$:

$$h(i, j) \leq g^*((i, j) \to (n_f, n_c)),$$

esto es poque los obstáculos solo pueden incrementar el camino real

### 4.2. Consistencia

Es consistente para cualquier transición válida $s \rightarrow s'$ con coste 1, $h(s) \leq 1 + h(s')$.

Moverse en fila o columna modifica la suma de diferencias en, a lo sumo, 1. Así $|n_f - i| + |n_c - j| \leq 1 + (|n_f - i'| + |n_c - j'|)$.

Es decir, h nunca salta más que el coste de un solo movimiento, garantizando que A* expanda los nodos en orden de coste creciente y encuentre rutas óptimas.

## 5. Bibliografía


1. Russell, S. y Norvig, P. (2003). *Inteligencia Artificial: un enfoque moderno*, 2ª ed., Prentice Hall.
2. Python implementation of algorithms from Russell And Norvig's "Artificial Intelligence - A Modern Approach": https://github.com/aimacode/aima-python.git