Pontificia Universidad Javeriana

Introducción a la Inteligencia Artificial

Proyecto Aplicativo

Karol Dayan Torres Vides

Juan Sebastián Sanchez

26/02/2026

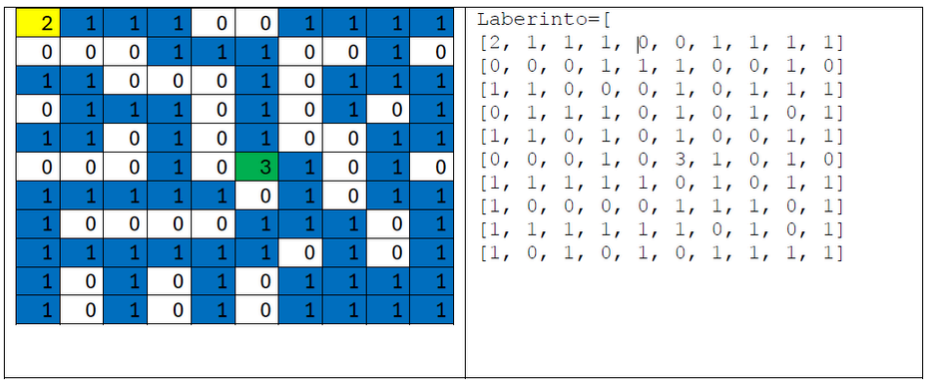
Docente Julio Omar

Problema

En este proyecto se modeló un laberinto representado como una matriz NxN, donde cada celda puede ser espacio libre (0), pared (1), inicio (2) o meta (3). A partir de esta representación se identificaron las coordenadas del nodo inicial y del nodo objetivo recorriendo la matriz. Luego, se transformó la matriz en un grafo usando una lista de adyacencia: cada celda transitable (diferente de pared) se consideró un nodo y se conectó con sus vecinos válidos en cuatro direcciones (arriba, abajo, izquierda, derecha). A cada enlace se le asignó un peso uniforme de 1, equivalente a un paso dentro del laberinto. Con el grafo construido, se aplicaron tres estrategias de búsqueda: BFS (búsqueda en anchura) utilizando una cola para explorar por niveles, DFS (búsqueda en profundidad) usando una pila para profundizar en un camino antes de retroceder, y A* como búsqueda informada, priorizando nodos según f(n) = g(n) + h(n), donde g(n) es el costo acumulado desde el inicio y h(n) es una heurística (distancia Manhattan) que estima lo que falta para llegar a la meta. Finalmente, se reconstruyó e imprimió la ruta calculada mediante un diccionario de padres (parent) que almacena el predecesor de cada nodo descubierto.

Lectura de Laberinto

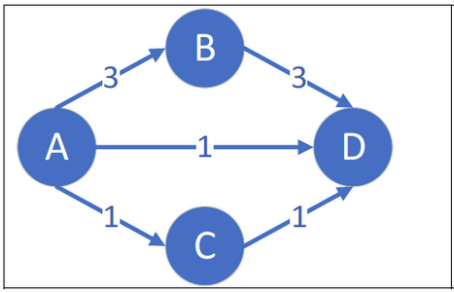
La lectura del laberinto se realizará por medio de una matriz en Python de tamaño NxN, los espacios son 0, las paredes 1, salida 2, meta 3



Lectura de Grafo

Para manipular el laberinto como uno de los algoritmo se realizará una transformación a un grafo, para esto se utilizará una lista de adyacencia, a

continuación, se presenta un ejemplo grafo ponderado como una lista de adyacencia.




```
Lista_nodos_adyacentes = {
    'A': [('B',3),('C',1),('D',1)],
    'B': [('D',3)],
    'C': [('D',1)]
}
```

Heurística

Implementación

1. **Lectura del laberinto (matriz)**

Se toma el laberinto como una matriz de tamaño NxN (matriz cuadrada). La variable r hará referencia a las filas mientras que la variable c se referirá a las columnas, con ayuda de la función **in_bounds** (ver fig. 1) se garantizará que tanto r como c siempre tengan valores mayores a cero pero menores al tamaño n de la matriz.

```python
# --------------------------
# A) Lectura / utilidades
# --------------------------

# --------------------------
# Funcion para verificar limites, retorna true si (r,c) esta dentro de la matriz n x n.
# Ni las filas o las columnas pueden ser negativas y/o sobrepasar el tamaño de la matriz.
# --------------------------
def in_bounds(n, r, c):
    return 0 <= r < n and 0 <= c < n
```

Fig. 1 Función in_bounds

La siguiente función retornará una lista de posiciones, éstas posiciones se irán agregando a la estructura pos con ayuda de append. Dichas coordenadas serán en las que encontremos el valor que buscamos al hacer la llamada de esta función. (Fig. 2)

```python
# --------------------------
# Si el valor coincide con el que estamos buscando (el pasado por parametro como value),
# guardamos sus coordenadas en 'pos' con append y retornamos la lista de posiciones encontradas.
# --------------------------
def find_value_positions(grid, value):
    """Devuelve lista de (r,c) donde grid[r][c] == value."""
    n = len(grid)
    pos = []
    for r in range(n):
        for c in range(n):
            if grid[r][c] == value:
                pos.append((r, c))
    return pos
```

Fig. 2 Función find_value_positions

¿Pero exactamente para qué la necesitamos?, vamos a usarla en la función a continuación para avisar de un error si se encuentra que hay mas de un par de coordenadas para el inicio y/o más de un par de coordenadas para la meta. (Fig. 3)

```python
# -------------------------
# Con esta funcion nos aseguramos de que solo haya una celda de inicio y una celda de meta.
# Se lanza un error en el caso contrario. Retorna las coordenadas de start y goal.
# -------------------------
def get_start_goal(grid):
    """Encuentra exactamente un START=2 y un GOAL=3."""
    starts = find_value_positions(grid, START)
    goals = find_value_positions(grid, GOAL)

    if len(starts) != 1:
        raise ValueError(f"Se esperaba exactamente 1 salida (2), encontré {len(starts)}.")
    if len(goals) != 1:
        raise ValueError(f"Se esperaba exactamente 1 meta (3), encontré {len(goals)}.")

    return starts[0], goals[0]
```

Fig. 3 Función get_start_goal

2. **Transformación matriz a grafo (nodos, enlaces, pesos)**

La estructura de datos directions = [(1,0), (-1,0), (0,1), (0,-1)] es una lista que guarda los movimientos posibles desde una celda del laberinto en una matriz. Cada tupla representa un cambio en la posición como (dr, dc), donde dr es el cambio en la fila y dc el cambio en la columna. En una matriz, aumentar la fila (dr = 1) significa bajar, disminuir la fila (dr = -1) significa subir; aumentar la columna (dc = 1) significa moverse a la derecha y disminuirla (dc = -1) significa moverse a la izquierda. Por eso: (1,0) = abajo, (-1,0) = arriba, (0,1) = derecha y (0,-1) = izquierda. En el código se usa con un ciclo for dr, dc in directions: para generar los vecinos de una celda (r,c) calculando nr = r + dr y nc = c + dc. Luego se filtra con una condición como in_bounds(n, nr, nc) y grid[nr][nc] != WALL para quedarnos solo con los vecinos que están dentro del tablero y no son pared. Esto permite construir el grafo del laberinto (o explorar el laberinto) conectando cada celda transitable con sus vecinos válidos. (Fig. 4)

```python
def matrix_to_graph(grid):
    """
    Construye lista de adyacencia:
    graph[(r,c)] = [((nr,nc), cost), ...]
    Solo incluye celdas != WALL.
    """
    n = len(grid)
    graph = {}

    directions = [(1,0), (-1,0), (0,1), (0,-1)]  # 4-dir

    for r in range(n):
        for c in range(n):
            if grid[r][c] == WALL:
                continue

            node = (r, c)
            graph[node] = []

            for dr, dc in directions:
                nr, nc = r + dr, c + dc
                if in_bounds(n, nr, nc) and grid[nr][nc] != WALL:
                    graph[node].append(((nr, nc), 1))  # costo 1 por paso

    return graph
```

Fig. 4 Función matrix_to_graph

3. **Reconstrucción de la ruta**

   Básicamente tenemos la función reconstruct_path la cual se devuelve desde goal hasta start, en ese camino va guardando los nodos para posteriormente realizar un path.reverse( ) para que las coordenadas queden en el orden esperado. Y mostramos ese camino con print_path. (Fig. 5)

```python
def reconstruct_path(parent, goal):
    """Reconstruye ruta desde goal usando parent dict, devuelve lista start->goal o None."""
    if goal not in parent:
        return None
    path = []
    cur = goal
    while cur is not None:
        path.append(cur)
        cur = parent[cur]
    path.reverse()
    return path

def print_path(path, title="Ruta"):
    if path is None:
        print(f"{title}: No hay ruta")
    else:
        print(f"{title} (len={len(path)}): {path}")
```

Fig. 5 Función reconstruct_path y print_path

4. **BFS (Búsqueda por anchura)**

En ‘cola’ se van a guardar los nodos por explorara a manera de FIFO, con visited se evitar’an las repeticiones y con el diccionario parent, se irá guardando la ruta que se toma. Mientras haya algo, en el while se irá sacando el primer elemento

con poplesft( ) y si u == goal se hace llamado a la función encargada de reconstruir el camino. Se irán recorriendo los vecinos de u. (Fig. 6)

```python
def bfs(graph, start, goal):
    cola = deque([start])
    visited = set([start])
    parent = {start: None}

    while cola:
        u = cola.popleft()
        # Si u es la meta reconstruirmos la ruta
        if u == goal:
            return reconstruct_path(parent, goal)
        # Recorremos vecinos de u, si no han sido visitados, los marcamos co
        for v, cost in graph.get(u, []):
            if v not in visited:
                visited.add(v)
                parent[v] = u
                cola.append(v)

    return None
```

Fig. 6 Función bfs

5. **DFS (Búsqueda por profundidad)**

Es parecido al de búsqueda por anchura, entre las diferencias de implementación se encuentra que se debe usar una stack puesto que funciona como una LIFO (Last in, first out), así que irá profundizando hasta llegar a ´goal´.

```python
def dfs(graph, start, goal):
    stack = [start]
    visited = set([start])
    parent = {start: None}

    while stack:
        u = stack.pop()
        if u == goal:
            return reconstruct_path(parent, goal)

        # El orden de vecinos afecta el camino que saldrá en DFS.
        for v, cost in graph.get(u, []):
            if v not in visited:
                visited.add(v)
                parent[v] = u
                stack.append(v)

    return None
```

Fig. 7 Función dfs

6. **Heurística**

La heurística Manhattan es una función que estima “qué tan lejos” está un nodo de la meta partiendo de que sólo se tienen 4 posibles direcciones (arriba, abajo, izquierda, derecha). Es útil en A* para guiar la búsqueda hacia la meta.

---> abs(r - rg) es la diferencia vertical en filas.

---> abs(c - cg) es la diferencia horizontal en columnas.

La suma es el número mínimo de pasos si no hubiera paredes.

```python
# ------------------------
# C3) Heurística
# ------------------------

def heuristic_manhattan(node, goal):
    r, c = node
    rg, cg = goal
    return abs(r - rg) + abs(c - cg)
```

Fig. 8 Función heuristic_manhattan

7. **A***

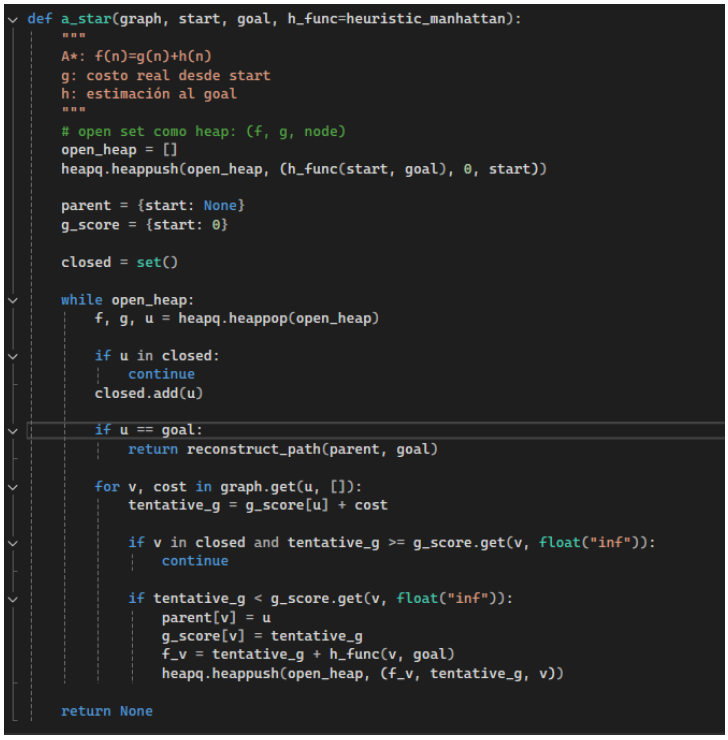
A* (A estrella) es un algoritmo de búsqueda informada que combina el costo real recorrido con una estimación de lo que falta para llegar a la meta. Su prioridad es:

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

g(n) = costo real desde el inicio hasta el nodo n

h(n) = heurística (estimación desde n a la meta)

f(n) = prioridad total para decidir qué explorar primero

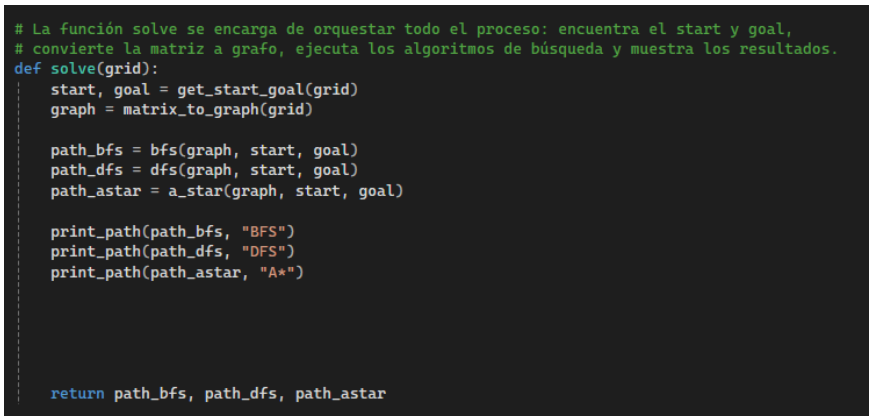


```python
def a_star(graph, start, goal, h_func=heuristic_manhattan):
    """
    A*: f(n)=g(n)+h(n)
    g: costo real desde start
    h: estimación al goal
    """
    # open set como heap: (f, g, node)
    open_heap = []
    heapq.heappush(open_heap, (h_func(start, goal), 0, start))

    parent = {start: None}
    g_score = {start: 0}

    closed = set()

    while open_heap:
        f, g, u = heapq.heappop(open_heap)

        if u in closed:
            continue
        closed.add(u)

        if u == goal:
            return reconstruct_path(parent, goal)

        for v, cost in graph.get(u, []):
            tentative_g = g_score[u] + cost

            if v in closed and tentative_g >= g_score.get(v, float("inf")):
                continue

            if tentative_g < g_score.get(v, float("inf")):
                parent[v] = u
                g_score[v] = tentative_g
                f_v = tentative_g + h_func(v, goal)
                heapq.heappush(open_heap, (f_v, tentative_g, v))

    return None
```

Fig. 9 Función a_star

Para hacerlo, mantiene una cola de prioridad (heap) con tuplas (f,g,nodo)(f,g,nodo) (f,g,nodo); en cada iteración saca el nodo con menor fff, lo marca como procesado, y revisa sus vecinos calculando un costo tentativo tentative_gtentative_gtentative_g. Si ese camino mejora el mejor costo conocido para un vecino, actualiza su parent (para reconstruir la ruta), guarda el nuevo g_score y vuelve a insertar el vecino en el heap con su nueva prioridad fff. Cuando extrae la meta del heap, reconstruye la ruta siguiendo parent desde la meta hasta el inicio.

8. LLamado de las funciones en orden y mostrar por pantalla



```python
# La función solve se encarga de orquestar todo el proceso: encuentra el start y goal,
# convierte la matriz a grafo, ejecuta los algoritmos de búsqueda y muestra los resultados.
def solve(grid):
    start, goal = get_start_goal(grid)
    graph = matrix_to_graph(grid)

    path_bfs = bfs(graph, start, goal)
    path_dfs = dfs(graph, start, goal)
    path_astar = a_star(graph, start, goal)

    print_path(path_bfs, "BFS")
    print_path(path_dfs, "DFS")
    print_path(path_astar, "A*")




    return path_bfs, path_dfs, path_astar
```

Fig. 10 Función solve

Desde la función solve, realizamos todo el proceso de determinar validar el inicio y la meta, crear el grafo, realixar la llamada de la búqueda por anchura/profundidad/A* e imprimir las correspondientes salidas.

Con la siguiente función, lo que se busca es delimitar el camino encontardo con ´*´ (asteriscos):

```python
# Lo que se hace aquí es tomar la ruta encontrada por cada algoritmo y superponerla en el laberinto original,
# marcando las celdas del camino con un símbolo (en este caso '*') para facilitar su visualización.
def overlay_path_on_grid(grid, path):
    if path is None:
        return grid
    new_grid = [row[:] for row in grid]
    for (r, c) in path:
        if new_grid[r][c] == 0:
            new_grid[r][c] = '*'
    return new_grid

def print_grid(grid):
    for row in grid:
        for cell in row:
            print(f"{str(cell):>2}", end=" ")
        print()
```

Fig. 11 Función overlay_path_on_grid

```
Laberinto con ruta BFS:
 2  1  1  1  0  0  1  1  1  1
 *  *  *  1  1  1  0  0  1  0
 1  1  *  *  *  1  0  1  1  1
 0  1  1  1  *  1  0  1  0  1
 1  1  0  1  *  1  0  0  1  1
 0  0  0  1  *  3  1  0  1  0
 1  1  1  1  1  0  1  0  1  1
 1  0  0  0  0  1  1  1  0  1
 1  1  1  1  1  1  0  1  0  1
 1  0  1  0  1  0  1  1  1  1
```

Fig. 12 Representaci'on del camino encontrado con ''*

Durante el desarrollo se presentaron varios desafíos técnicos. Primero, al generar vecinos desde una celda, fue necesario evitar accesos fuera de la matriz; esto se resolvió implementando una verificación de límites (in_bounds) antes de consultar el contenido de una celda. Segundo, para impedir ciclos y repeticiones (por ejemplo, volver a una celda ya explorada), se empleó un conjunto visited que marca nodos visitados en BFS y DFS, y un conjunto closed para nodos ya procesados en A*. Tercero, aunque los algoritmos podían detectar la

meta, inicialmente no era posible reconstruir el camino completo; se solucionó guardando en un diccionario parent el nodo desde el cual se llegó a cada vecino, permitiendo reconstruir la ruta desde la meta hasta el inicio y luego invertirla. Cuarto, en A* surgió el reto de actualizar correctamente los costos: se utilizó un diccionario g_score para almacenar el mejor costo conocido hacia cada nodo y se actualizó únicamente cuando se encontró un camino con menor costo; los nodos pendientes se gestionaron con una cola de prioridad (heapq) para siempre expandir el nodo con menor f(n). Por último, se validó que el laberinto contuviera exactamente un inicio y una meta para evitar ejecuciones ambiguas, generando mensajes de error cuando la entrada no cumplía la estructura esperada.