TP2

August 21, 2025

1 Temas Tratados en el Trabajo Práctico 2

- Conceptos de Búsqueda no Informada y Búsqueda Informada.
- Concepto de Heurística.
- Abstracción de Problemas como Gráficos de Árbol.
- Estrategias de Búsqueda no Informada: Primero en Amplitud, Primero en Profundidad y Profundidad Limitada.
- Estrategias de Búsqueda Informada: Búsqueda Voraz, Costo Uniforme, A*.

1.1 Ejercicios Teóricos

1. ¿Qué diferencia hay entre una estrategia de búsqueda Informada y una estrategia de búsqueda No Informada?

La diferencia entre estrategias de búsqueda no informadas y estrategias de búsqueda informadas es la siguiente:

Búsqueda no informada (o ciega): El agente no tiene información adicional sobre los estados más allá de la definición del problema. Solo sabe si un estado es objetivo o no, y va explorando el espacio de estados sin guía. Ejemplos: búsqueda en amplitud, búsqueda en profundidad, búsqueda de costo uniforme. Es como buscar un objeto en una habitación oscura sin ninguna pista, probando caminos al azar hasta encontrarlo.

Búsqueda informada (o heurística): Es un sistema de búsqueda que puede encontrar soluciones de manera mas eficiente. Utiliza el conocimiento específico del problema mas allá de la definición en si misma. El agente usa información adicional que lo orientan sobre qué tan cerca está un estado de la solución. Esto le permite ser más eficiente, reduciendo el espacio de búsqueda. Ejemplos: búsqueda voraz primero el mejor, A*. Es como buscar un objeto en una habitación pero con una linterna que te da pistas de dónde está más cerca.

2. ¿Qué es una heurística y para qué sirve?

La función heurística representa el costo estimado del camino mas barato desde el nodo en que se encuentra a un nodo objetivo. Sirve como una “pista” o un “consejo” para guiar la búsqueda. Aunque no garantiza que sea la ruta real más corta, ayuda a priorizar los caminos que parecen más prometedores.

Un ejemplo podría ser: suponiendo que un robot debe ir desde un punto A a un punto B, la heurística podría ser la distancia en línea recta entre A y B. No siempre será el camino real más

corto (porque puede haber obstáculos), pero es una aproximación rápida que ayuda al robot a explorar primero los estados más cercanos al objetivo.

3. ¿Es posible que un algoritmo de búsqueda no tenga solución?

Si, un algoritmo de búsqueda puede no tener solucion ya sea porque el problema no tiene estados objetivo alcanzables, porque el algoritmo no es completo (como en profundidad con bucles infinitos), o porque existen limitaciones de tiempo y memoria que impiden explorar todo el espacio de estados. Por esta razon, al momento de generar los códigos, hay que tener esta posibilidad en cuenta

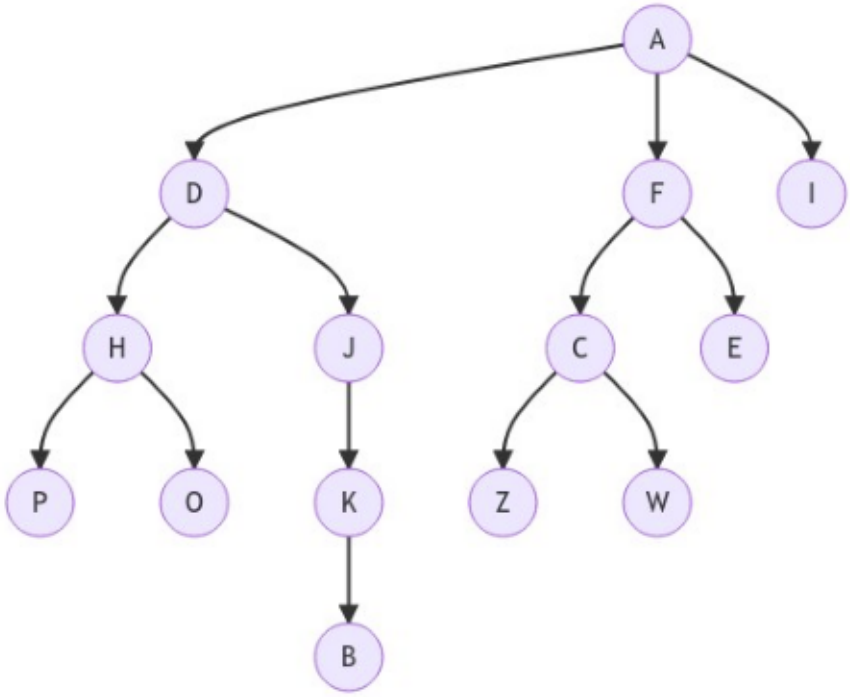
4. Describa en qué secuencia será recorrido el Árbol de Búsqueda representado en la imagen cuando se aplica un Algoritmo de Búsqueda con la estrategia:

   4.1 Primero en Amplitud.

   4.2 Primero en Profundidad.

   4.3 Primero en Profundidad con Profundidad Limitada Iterativa (comenzando por 1).

[2]:
```
import requests
from PIL import Image
from io import BytesIO
import matplotlib.pyplot as plt

# URL directa de Google Drive
url = "https://drive.google.com/uc?
 ↪export=view&id=1IJDEKWhfMEzXnzr28RgTNOuKBER2NsuP"

# Descargar la imagen
response = requests.get(url)
img = Image.open(BytesIO(response.content))

# Mostrar la imagen
plt.imshow(img)
plt.axis('off')  # Ocultar ejes
plt.show()
```



Muestre la respuesta en una tabla, indicando para cada paso que da el agente el nodo que evalúa actualmente y los que están en la pila/cola de expansión según corresponda.

[ ]:
```
url = "https://drive.google.com/uc?
  ↪export=view&id=1fW_BgT5muzQffMVRcIiB2mM2Zf66Nb-m"
response = requests.get(url)
img = Image.open(BytesIO(response.content))

# Mostrar con figura más grande
plt.figure(figsize=(img.width / 80, img.height / 80))  # ajustá el divisor␣
  ↪según densidad deseada
plt.imshow(img)
plt.axis('off')
plt.show()
```

Estado Actual	↓							

4.1: Primero en Amplitud

Estado actual								
A	D	F	I					
D	F	I	H	J				
F	I	H	J	C	E			
I	H	J	C	E				
H	J	C	E	P	O			
J	C	E	P	O	K			
C	E	P	O	K	Z	W		
E	P	O	K	Z	W			
P	O	K	Z	W				
O	K	Z	W					
K	Z	W	B					
Z	W	B						
W	B							
B								

4.2: Primero en profundidad

Estado actual					
A	D	F	I		
D	H	J	F	I	
H	P	O	J	F	I
P	O	J	F	I	
O	J	F	I		
J	K	B	F	I	
K	B	F	I		
B	F	I			
F	C	E	I		
C	Z	W	E	I	
Z	W	E	I		
W	E	I			
E	I				
I					

4.3 Primero en Profundidad con Profundidad Limitada Iterativa (comenzando por 1).

Estado actual					
Limite 1:					
A	D	F	I		
D	F	I			
F	I				
I					
Límite 2:					
A	D	F	I		
D	H	J	F	I	
H	J	F	I		
J	F	I			
F	C	E	I		
C	E	I			
E	I				
I					
Límite 3:					
A	D	F	I		
D	H	J	F	I	
H	P	O	J	F	I
P	O	J	F	I	
O	J	F	I		
J	K	F	I		
K	F	I			
F	C	E	I		
C	Z	W	E	I	
Z	W	E	I		
W	E	I			
E	I				
I					
Límite 4:					
A	D	F	I		
D	H	J	F	I	
H	P	O	J	F	I
P	O	J	F	I	
O	J	F	I		
J	K	B	F	I	
K	B	F	I		
B	F	I			
F	C	E	I		
C	Z	W	E	I	
Z	W	E	I		
W	E	I			
E	I				
I					

1.2 Ejercicios de Implementación

5. Represente el tablero mostrado en la imágen como un árbol de búsqueda y a continuación programe un agente capaz de navegar por el tablero para llegar desde la casilla I a la casilla F utilizando:

   `5.1 La estrategia Primero en Profundidad.`

   `5.2 La estrategia Avara.`

   `5.3 La estrategia A*.`

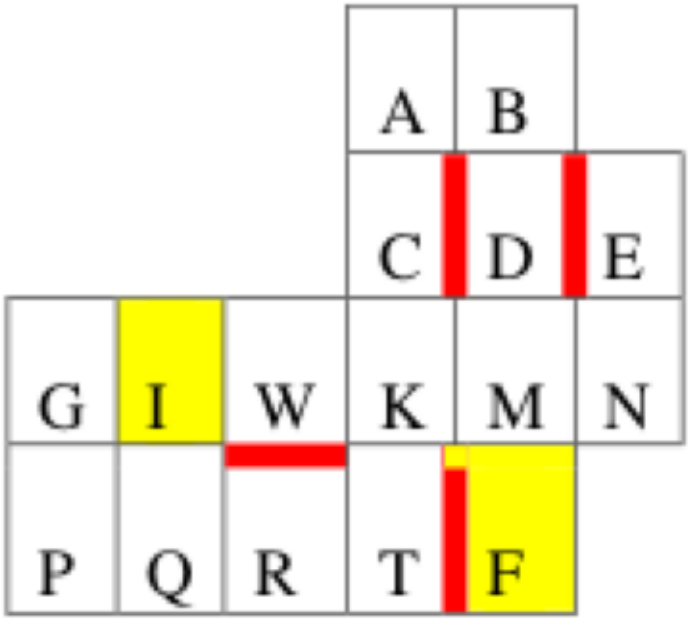
Considere los siguientes comportamientos del agente:

- El agente no podrá moverse a las casillas siguientes si las separa una pared.
- La heurística empleada en el problema es la Distancia de Manhattan hasta la casilla objetivo (el menor número de casillas adyacentes entre la casilla actual y la casilla objetivo).
- El costo de atravesar una casilla es de 1, a excepción de la casilla W, cuyo costo al atravesarla

es 30.

- En caso de que varias casillas tengan el mismo valor para ser expandidas, el algoritmo eligirá en orden alfabético las casillas que debe visitar.

[9]:
```
url = "https://drive.google.com/uc?
 ↪export=view&id=1FajYiBQ507o6yiE7MndL-PQXyoyELtuD"
response = requests.get(url)
img = Image.open(BytesIO(response.content))
plt.imshow(img)
plt.axis('off')
plt.show()
```



(D no puede ser hijo de C… Hay que ver como llenarlo) No es lo mismo encontrar el camino mas corto que el camino mas barato Hay que ver como pasar de las casillas a un arbol de busqueda (Dice de izquierda a derecha en orden alfabetico, siempre debe haber un criterio) Estado inicial: I —> Expandir (G, Q y W); G —> Expandir (I y P) (Como ya se paso por I, hay que hacer la poda del arbol de búsqueda…. hay que enforcarse si fue nodo padre o si solo ya estuvo) P —> Expandir (Q)

Primero en Profundidad

Estado Actual							
I	G	Q	W				
G	P	Q	W				
P	Q	W					
Q	R	W					
R	T	W					
T	K	W					
K	C	M	W				
C	A	M	W				
A	B	M	W				
B	D	M	W				
D	M	W					
M	F	N	W				
F							

Avara

Estado Actual							
I	W	Q	G				
W	K	Q	G				
K	M	C	T	Q	G		
M	F	D	N	C	T	Q	G
F							

A*

Estado Actual							
I	Q	G	W				
Q	R	P	G	W			
R	T	P	G	W			
T	K	P	G	W			
K	M	C	P	G	W		
M	F	D	N	C	P	G	W
F							

[ ]:
```python
import heapq
from collections import deque

# Grafo representado como diccionario de adyacencia
# Cada clave es un nodo y el valor es otro diccionario con sus vecinos y el costo de ir hacia ellos
graph = {
    "A": {"B": 1, "C": 1},
    "B": {"A": 1, "D": 1},
    "C": {"A": 1, "K": 1},
    "D": {"B": 1, "M": 1},
    "E": {"N": 1},
    "G": {"I": 1, "P": 1},
    "I": {"G": 1, "Q": 1, "W": 1},
```

```
    "W": {"I": 30, "K": 30},  # pesos altos simulan que este camino no es ideal
    "K": {"W": 1, "M": 1, "T": 1, "C": 1},
    "M": {"K": 1, "N": 1, "F": 1, "D": 1},
    "N": {"M": 1, "E": 1},
    "P": {"G": 1, "Q": 1},
    "Q": {"I": 1, "P": 1, "R": 1},
    "R": {"Q": 1, "T": 1},
    "T": {"K": 1, "R": 1},
    "F": {"M": 1},
}

goal = "F"

# Conjunto de "paredes": aristas que existen en el grafo pero que no deben␣
 ↪usarse
# Se agregan como restricciones para los algoritmos de búsqueda
walls = {("C","D"), ("D","E"), ("W","R"), ("T","F")}


def maze_distance(start, goal):
    """
    Calcula la distancia real más corta entre 'start' y 'goal'
    usando BFS, considerando las paredes como prohibidas.
    Esto se usa como heurística para Avara y A*.
    """
    queue = deque([(start, 0)])  # Cola para BFS (nodo, distancia)
    visited = {start}            # Conjunto de visitados para no repetir nodos

    while queue:
        node, dist = queue.popleft()
        if node == goal:
            return dist  # Se encontró el objetivo, devolvemos la distancia

        for neighbor in graph[node]:
            # Ignorar vecinos que estén bloqueados por paredes
            if (node, neighbor) in walls or (neighbor, node) in walls:
                continue
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, dist+1))

    # Si no hay camino posible, devolvemos infinito
    return float("inf")

# Diccionario de heurística: para cada nodo precomputamos su distancia mínima␣
 ↪al objetivo
heuristic = {n: maze_distance(n, goal) for n in graph}
```

```
print(heuristic)


def dfs(start, goal):
    """
    Algoritmo DFS clásico (con pila).
    Explora caminos hasta encontrar la meta.
    No garantiza ser el mas optimo.
    """
    stack = [(start, [start], 0)]  # (nodo actual, camino recorrido, costo␣
 ↪acumulado)
    visited = set()

    while stack:
        node, path, cost = stack.pop()
        if node == goal:
            return path, cost
        if node in visited:
            continue
        visited.add(node)

        # Se recorre en orden alfabético invertido para mantener consistencia␣
 ↪en el recorrido
        for neighbor, w in sorted(graph[node].items(), reverse=True):
            stack.append((neighbor, path + [neighbor], cost + w))
    return None

def avara(start, goal):
    """
    Algoritmo avara.
    Siempre expande el nodo cuya heurística es menor.
    NO garantiza el camino óptimo porque ignora el costo real recorrido.
    """
    frontier = [(heuristic[start], start, [start], 0)]  # (h(n), nodo, camino,␣
 ↪costo real)
    visited = set()

    while frontier:
        _, node, path, cost = heapq.heappop(frontier)  # Elige siempre el de␣
 ↪menor heurística
        if node == goal:
            return path, cost

        if node in visited:
            continue
        visited.add(node)
```

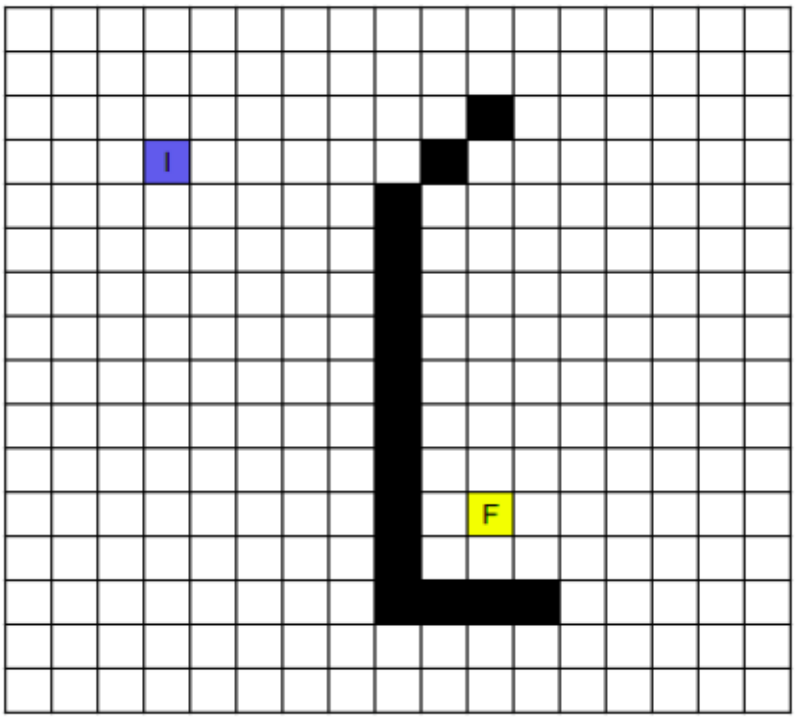
```
        for neighbor, w in graph[node].items():
            heapq.heappush(frontier, (heuristic[neighbor], neighbor,␣
 ↪path+[neighbor], cost+w))
    return None

def astar(start, goal):
    """
    Algoritmo A*.
    Combina el costo real g(n) con la heurística h(n).
    Siempre expande el nodo con menor f(n) = g(n) + h(n).
    Garantiza ser el mas optimo si la heurística es admisible.
    """
    frontier = [(heuristic[start], 0, start, [start])]  # (f, g, nodo, camino)
    visited = {}  # Guarda el costo g más bajo con el que se visitó un nodo

    while frontier:
        f, g, node, path = heapq.heappop(frontier)
        if node == goal:
            return path, g

        # Si ya encontramos un camino más barato antes, no seguimos este
        if node in visited and visited[node] <= g:
            continue
        visited[node] = g

        for neighbor, w in graph[node].items():
            g2 = g + w
            f2 = g2 + heuristic[neighbor]
            heapq.heappush(frontier, (f2, g2, neighbor, path+[neighbor]))
    return None

print("DFS:", dfs("I", "F"))
print("Avara:", avara("I", "F"))
print("A*:", astar("I", "F"))
```

```
PS C:\Users\tomas\OneDrive\Escritorio\Facu\IA\IA> & "C:/Users/tomas/OneDrive/Escritorio/Archivos Programas/Python/python.exe" c:/Use
rs/tomas/OneDrive/Escritorio/Facu/IA/IA/TP2/Ejercicio_5.py
{'A': 4, 'B': 3, 'C': 3, 'D': 2, 'E': 3, 'G': 5, 'I': 4, 'W': 3, 'K': 2, 'M': 1, 'N': 2, 'P': 6, 'Q': 5, 'R': 4, 'T': 3, 'F': 0}
DFS: (['I', 'G', 'P', 'Q', 'R', 'T', 'K', 'C', 'A', 'B', 'D', 'M', 'F'], 12)
Avara: (['I', 'W', 'K', 'M', 'F'], 33)
A*: (['I', 'Q', 'R', 'T', 'K', 'M', 'F'], 6)
```

6. Desarrolle un agente que emplee una estrategia de búsqueda A* para ir de una casilla a otra evitando la pared representada, pudiendo seleccionar ustedes mismos el inicio y el final. Muestre en una imagen el camino obtenido.

[5]:
```
url = "https://drive.google.com/uc?
 ↪export=view&id=1fD2Ws5oqFU9_RTj-yX9BIvslXJiqcLCZ"
response = requests.get(url)
```

```
img = Image.open(BytesIO(response.content))
plt.imshow(img)
plt.axis('off')
plt.show()
```

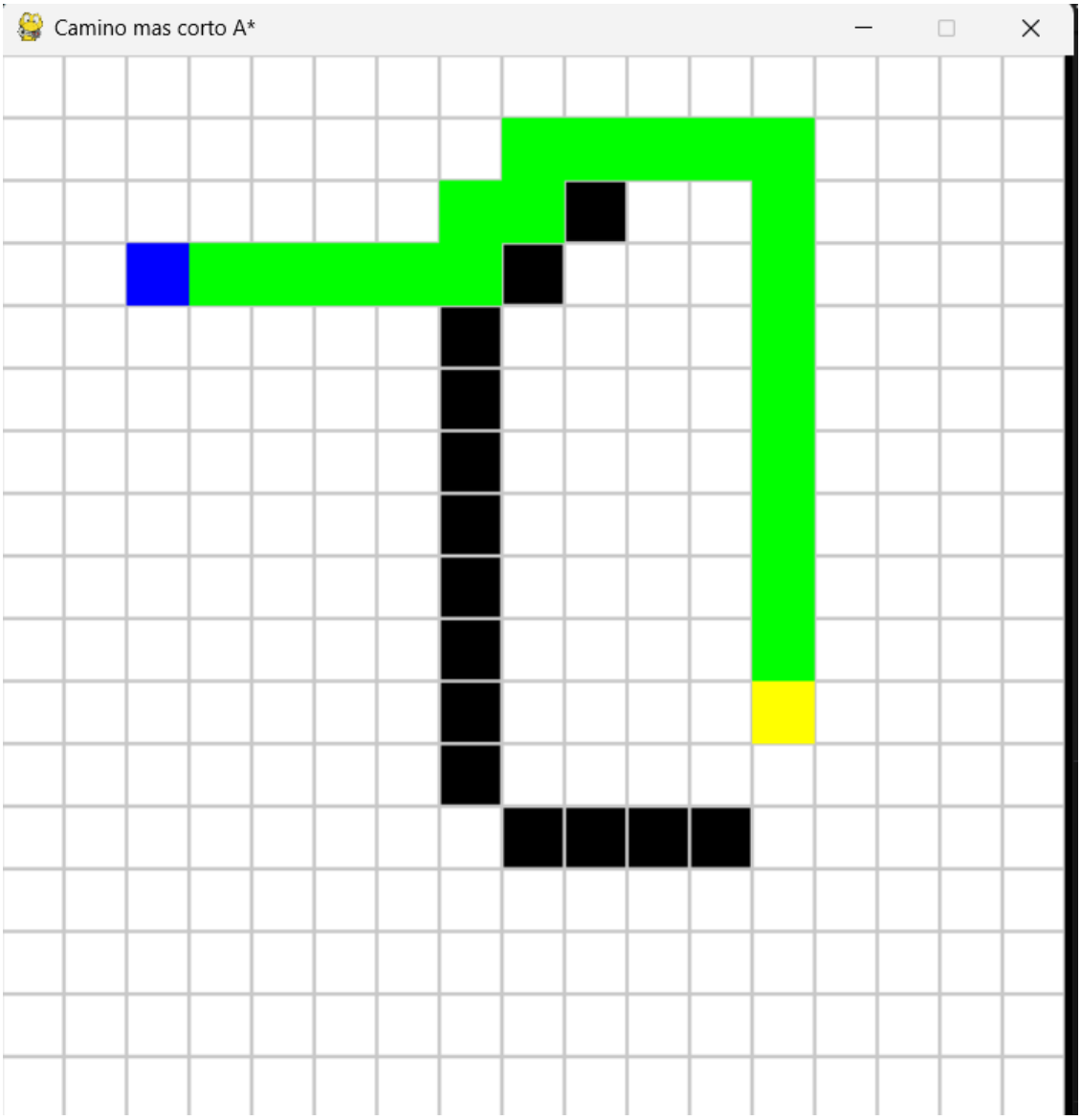


[ ]:
```
import pygame
import heapq
import time

# Configuración
WIDTH, HEIGHT = 600, 600    # Dimensiones de la ventana
ROWS, COLS = 17, 17          # Filas y columnas de la cuadrícula
CELL_SIZE = WIDTH // COLS       # Tamaño de cada celda

# Colores
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
BLUE = (0, 0, 255)
YELLOW = (255, 255, 0)
GREEN = (0, 255, 0)
GRAY = (200, 200, 200)

pygame.init()
```

```
win = pygame.display.set_mode((WIDTH, HEIGHT))  # Inicializar ventana
pygame.display.set_caption("Camino mas corto A*")   # Título de la ventana


grid = [[0 for _ in range(COLS)] for _ in range(ROWS)]  # Crear cuadrícula vacía



# Obstáculos según la imagen
obstacles = [
   (2, 9), (3, 8), (4, 7), (5, 7), (6, 7), (7, 7), (8, 7), (9, 7), (10, 7),␣
 ↪(11, 7),
    (12, 8), (12, 9), (12, 10), (12, 11), (12, 11)
]                                   #Crea los obstáculos punto a punto (Revisar)
for r, c in obstacles:   # Agregar obstáculos a la cuadrícula
    grid[r][c] = 1

start, end = None, None

# --- Algoritmo A* ---
def heuristic(a, b):            # Heurística de Manhattan
    return abs(a[0] - b[0]) + abs(a[1] - b[1])      # Calcula la distancia␣
 ↪Manhattan entre dos puntos

def astar(start, goal):             # Algoritmo A*
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {start: 0}
    f_score = {start: heuristic(start, goal)}

    while open_set:                                 # Mientras haya nodos por␣
 ↪explorar
        _, current = heapq.heappop(open_set)        # Obtiene el nodo con menor␣
 ↪costo
        if current == goal:                         # Si se llega al objetivo....
            path = []                                # Lista para almacenar el␣
 ↪camino
            while current in came_from:              # Retrocede por el camino
                path.append(current)                 # Se agrega el nodo actual␣
 ↪al camino
                current = came_from[current]         # Obtener el nodo anterior
            return path[::-1]                        # Retornar el camino␣
 ↪invertido
```

```
        neighbors = [(0,1),(1,0),(0,-1),(-1,0)]     # Indica cuales son los mov␣
↪válidos
        for dx, dy in neighbors:
            neighbor = (current[0] + dx, current[1] + dy)
            if 0 <= neighbor[0] < ROWS and 0 <= neighbor[1] < COLS:     #␣
↪Verifica los límites de la cuadrícula
                if grid[neighbor[0]][neighbor[1]] == 1:             # Si es␣
↪pared, se salta
                    continue
                tentative_g = g_score[current] + 1                  # Costo del␣
↪movimiento
                # Si encontramos un camino más barato hacia 'neighbor'...
                if tentative_g < g_score.get(neighbor, float("inf")):
                    came_from[neighbor] = current                    # Se␣
↪guarda el nodo actual como el anterior
                    g_score[neighbor] = tentative_g                  # Se␣
↪actualiza el costo g
                    f_score[neighbor] = tentative_g + heuristic(neighbor, goal)␣
↪    # Se actualiza el costo f  (f = g + h)
                    heapq.heappush(open_set, (f_score[neighbor], neighbor))  #␣
↪Se agrega el vecino a la lista abierta
    return None  # (Si no se encontró un camino)

# --- Dibujo ---
def draw_grid():
    for r in range(ROWS):
        for c in range(COLS):
            color = WHITE
            if grid[r][c] == 1:
                color = BLACK                    # Se dibujan los obstáculo
             # Rectángulo de la celda
            pygame.draw.rect(win, color, (c*CELL_SIZE, r*CELL_SIZE, CELL_SIZE,␣
↪CELL_SIZE))
            # Líneas de la cuadrícula
            pygame.draw.rect(win, GRAY, (c*CELL_SIZE, r*CELL_SIZE, CELL_SIZE,␣
↪CELL_SIZE), 1)

    if start:
        # Celda inicio en azul (nota: (x,y) en pantalla = (col, fila))
        pygame.draw.rect(win, BLUE, (start[1]*CELL_SIZE, start[0]*CELL_SIZE,␣
↪CELL_SIZE, CELL_SIZE))
    if end:
         # Celda fin en amarillo
        pygame.draw.rect(win, YELLOW, (end[1]*CELL_SIZE, end[0]*CELL_SIZE,␣
↪CELL_SIZE, CELL_SIZE))
```

```
def draw_path(path):
    if not path:
        return

    for i, (row, col) in enumerate(path):
        # Saltamos el primer nodo (inicio) y el último (destino),
        # para que sigan mostrándose con sus colores originales
        if (row, col) == start or (row, col) == end:
            continue

        # El último del camino lo pintamos amarillo
        if i == len(path) - 1:
            color = (255, 255, 0)  # Amarillo
        else:
            color = (0, 255, 0)    # Verde

        pygame.draw.rect(win, color, (col * CELL_SIZE, row * CELL_SIZE,␣
 ↪CELL_SIZE, CELL_SIZE))

# --- Loop principal ---
running = True
path = []
while running:
    draw_grid()          # Dibuja el tablero y (si existen) inicio/fin...
    if path:
        time.sleep(0.1)  # Pausa para visualizar el camino
        draw_path(path)
    pygame.display.update() # Actualiza la ventana

    for event in pygame.event.get():     # Procesa eventos de la ventana
        if event.type == pygame.QUIT:    # Cerrar ventana
            running = False

        if event.type == pygame.MOUSEBUTTONDOWN: # Click del mouse
            x, y = pygame.mouse.get_pos()       # Posición en píxeles
            r, c = y // CELL_SIZE, x // CELL_SIZE   # Lo convierte a (fila,␣
 ↪columna)
            if not start:
                start = (r, c)      # Primer click: fija el inicio
            elif not end:           # Segundo click: fija el final
                end = (r, c)
                path = astar(start, end)  # Llama a A* y guarda el camino␣
 ↪resultante

pygame.quit()
# Fin del programa
```



2 Bibliografía

Russell, S. & Norvig, P. (2004) *Inteligencia Artificial: Un Enfoque Moderno.* Pearson Educación S.A. (2a Ed.) Madrid, España

Poole, D. & Mackworth, A. (2023) *Artificial Intelligence: Foundations of Computational Agents.* Cambridge University Press (3a Ed.) Vancouver, Canada