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?

En una busqueda no informada o tambien llamada a ciegas, no se cuenta con ningun tipo de informacion adicional sobre los estados que no son objetivos, por otro lado la estragia de busqueda informada o tambien llamada heuristica cuenta con estrategias que le permiten saber si un estado objetivo es mas prometedor que otro. Es decir que en la no informada, no se dispondra de ninguna pista sobre que tan cerca esta cada estado de la meta, por eso se explora el espacio sin guia mas alla de la estructura. En cambio en la informada, se usa un conocimiento adicional que estima el costo de la ruta mas barata desde el estado n hasta la meta, esta pista se utiliza para dirigirse hacia los estados mas prometedores

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

Una heuristica es una funcion h(n) que dado cierto nodo en el espacio, devuelve una estimacion del costo minimo restante para alcanzar un estado meta desde n, no garantiza exactitud pero orienta la busqueda. Sirve para guiar la busqueda, reducir costo computacional, mejorar eficiencia, balancear precision y rapidez, etc.

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

Si es posible que un algoritmo de busqueda no tenga solucion, puede deberse a que el problema en si no tiene solucion o que se haya definido incorrectamente el problema o que el algoritmo tenga limitaciones o que los recursos sean insuficientes.

Como por ejemplo en la Busqueda de Profundidad Limitada, si definimos un limite de profunidad menor al necesario, no llegara a ninguna solucion

O tambien en la Busqueda A*, como guarda en memoria todos los nodos generados, se puede llegar a ocupar toda la memoria disponible antes de hallar una solucion

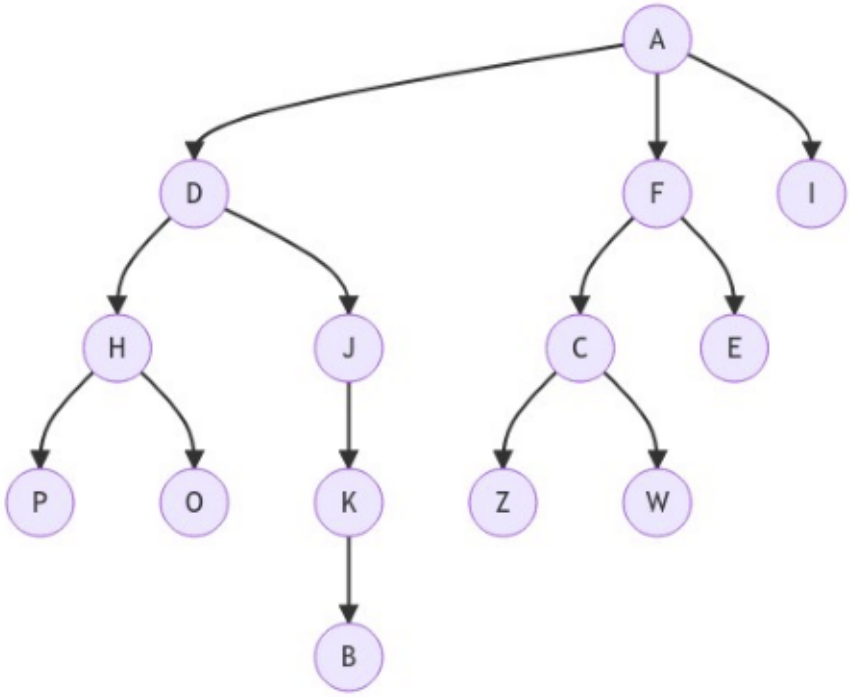
4. Describa en qué secuencia será recorrido el Árbol de Búsqueda representado en la imagen cuando se aplica un Algoritmo de Búsqueda con la estrategia:

   4.1 Primero en Amplitud.

   4.2 Primero en Profundidad.

   4.3 Primero en Profundidad con Profundidad Limitada Iterativa (comenzando por un nivel de

[1]:
```
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.

Punto A:

[8]:
```
from PIL import Image
import matplotlib.pyplot as plt

# Abrir la imagen desde la carpeta del proyecto
img = Image.open("tp2_1.png")

# Mostrar la imagen con matplotlib
plt.figure(figsize=(img.width/80, img.height/80))  # Ajustá el divisor para el␣
↪tamaño
plt.imshow(img)
plt.axis("off")
plt.show()
```

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								

Punto B:

[ ]:
```
from PIL import Image
import matplotlib.pyplot as plt

# Abrir la imagen desde la carpeta del proyecto
img = Image.open("tp2_2.png")

# Mostrar la imagen con matplotlib
plt.figure(figsize=(img.width/80, img.height/80))  # Ajustá el divisor para el␣
  ↪tamaño
plt.imshow(img)
plt.axis("off")
plt.show()
```

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	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							

Punto C: con 1 nivel de profundidad

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

# Abrir la imagen desde la carpeta del proyecto
img = Image.open("tp2_3a.png")

# Mostrar la imagen con matplotlib
plt.figure(figsize=(img.width/80, img.height/80))  # Ajustá el divisor para el␣
↪tamaño
plt.imshow(img)
plt.axis("off")
plt.show()
```

Estado Actual						
A	D	F	I			
D	F	I				
F	I					
I						

Punto c: con 2 niveles de profundidad

[3]:
```
from PIL import Image
import matplotlib.pyplot as plt

# Abrir la imagen desde la carpeta del proyecto
img = Image.open("tp2_3b.png")

# Mostrar la imagen con matplotlib
plt.figure(figsize=(img.width/80, img.height/80))  # Ajustá el divisor para el␣
 ↪tamaño
plt.imshow(img)
plt.axis("off")
plt.show()
```

Estado Actual					
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					

Punto c: con 3 niveles de profundidad

[4]:
```
from PIL import Image
import matplotlib.pyplot as plt

# Abrir la imagen desde la carpeta del proyecto
img = Image.open("tp2_3c.png")

# Mostrar la imagen con matplotlib
plt.figure(figsize=(img.width/80, img.height/80))  # Ajustá el divisor para el␣
 ↪tamaño
plt.imshow(img)
plt.axis("off")
plt.show()
```

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	F	I			
K	F	I				
F	C	E	I			
C	Z	W	E	I		
Z	W	E	I			
W	E	I				
E	I					
E						

Punto c: si hay 4 o mas niveles de profundidad, se llega a lo mismo que el punto 4.2

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. orden alfabetico con camino mas largo por como`

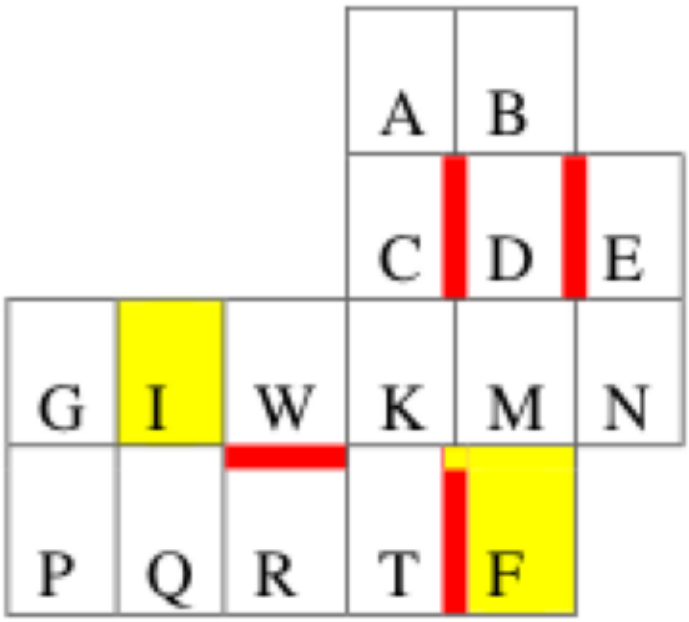
   `5.2 La estrategia Avara. camino mas corto sin importar el costo`

   `5.3 La estrategia A*. costo uniforme, paso por menor costo no menor costo total`

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.

[ ]:
```
import requests
from PIL import Image
from io import BytesIO
# Descargar y mostrar una imagen desde Google Drive
import matplotlib.pyplot as plt
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()
```



[2]:
```
from collections import deque
import heapq

# ========================
# Clases (tus adyacencias)
# ========================
class A:
    adyacentes = ["B", "C"]
    costo = 1
    inicio = False
```

```
    objetivo = False

class B:
    adyacentes = ["A", "D"]
    costo = 1
    inicio = False
    objetivo = False

class C:
    adyacentes = ["A", "K"]   # pared con D
    costo = 1
    inicio = False
    objetivo = False

class D:
    adyacentes = ["B", "M"]   # paredes con C y E
    costo = 1
    inicio = False
    objetivo = False

class E:
    adyacentes = ["N"]        # pared con D
    costo = 1
    inicio = False
    objetivo = False

class F:
    adyacentes = ["M"]        # pared con T
    costo = 1
    inicio = False
    objetivo = True

class G:
    adyacentes = ["I", "P"]
    costo = 1
    inicio = False
    objetivo = False

class I:
    adyacentes = ["G", "Q", "W"]  # según tu definición
    costo = 1
    inicio = True
    objetivo = False

class K:
    adyacentes = ["C", "M", "T", "W"]
    costo = 1
```

```
    inicio = False
    objetivo = False

class M:
    adyacentes = ["D", "F", "K", "N"]
    costo = 1
    inicio = False
    objetivo = False

class N:
    adyacentes = ["E", "M"]
    costo = 1
    inicio = False
    objetivo = False

class P:
    adyacentes = ["G", "Q"]
    costo = 1
    inicio = False
    objetivo = False

class Q:
    adyacentes = ["I", "P", "R"]
    costo = 1
    inicio = False
    objetivo = False

class R:
    adyacentes = ["Q", "T"]   # pared con W
    costo = 1
    inicio = False
    objetivo = False

class T:
    adyacentes = ["K", "R"]   # pared con F
    costo = 1
    inicio = False
    objetivo = False

class W:
    adyacentes = ["I", "K"]   # pared con R
    costo = 30
    inicio = False
    objetivo = False

# ========================
# Helpers y heurística
```

```python
# ========================
NODES = {cls.__name__: cls for cls in [A,B,C,D,E,F,G,I,K,M,N,P,Q,R,T,W]}

def get_start_and_goal():
    start = next(name for name, cls in NODES.items() if getattr(cls, "inicio",␣
 ↪False))
    goal  = next(name for name, cls in NODES.items() if getattr(cls,␣
 ↪"objetivo", False))
    return start, goal

def neighbors(u: str):
    # Siempre en orden alfabético (regla de desempate / decisión)
    return sorted(list(getattr(NODES[u], "adyacentes")))

def enter_cost(v: str) -> int:
    return int(getattr(NODES[v], "costo", 1))

def reconstruct_path(parent, goal):
    path = [goal]
    while path[-1] in parent:
        path.append(parent[path[-1]])
    path.reverse()
    return path

# Coordenadas para Manhattan (ajustá si tu layout es otro)
POS = {
    # y=0
    'P': (0,0), 'Q': (1,0), 'R': (2,0), 'T': (3,0), 'F': (4,0),
    # y=1
    'G': (0,1), 'I': (1,1), 'W': (2,1), 'K': (3,1), 'M': (4,1), 'N': (5,1),
    # y=2
    'C': (3,2), 'D': (4,2), 'E': (5,2),
    # y=3
    'A': (3,3), 'B': (4,3),
}
def manhattan(u: str, v: str) -> int:
    x1,y1 = POS[u]; x2,y2 = POS[v]
    return abs(x1-x2) + abs(y1-y2)

# ========================
# 1) DFS alfabético (sin costos)
# ========================
def dfs_alfabetico():
    """
    Profundidad: toma SIEMPRE el siguiente vecino en orden alfabético,␣
 ↪ignorando costos.
    Evita ciclos dentro del camino actual (pila) para no laquearse.
```

```
    """
    start, goal = get_start_and_goal()
    stack = [(start, iter(neighbors(start)))]
    parent = {}
    in_path = {start}  # nodos en la rama actual (para no volver por el mismo␣
 ↪ciclo)

    while stack:
        u, it = stack[-1]
        if u == goal:
            # costo final SOLO para informar (suma de entrada), aunque DFS no␣
 ↪decide por costo
            cost = 0
            path = reconstruct_path(parent, goal)
            for a,b in zip(path, path[1:]):
                cost += enter_cost(b)
            return path, cost

        try:
            v = next(it)  # siguiente vecino en orden alfabético
            if v not in in_path:
                parent[v] = u
                stack.append((v, iter(neighbors(v))))
                in_path.add(v)
        except StopIteration:
            # no hay más vecinos: backtrack
            stack.pop()
            in_path.discard(u)

    return None, None

# =========================
# 2) Greedy (estrategia avara)
# =========================
def greedy_best_first():
    """
    Estrategia avara: prioriza MENOR h(n) (Manhattan). Desempata␣
 ↪alfabéticamente.
    Ignora costos para decidir; calcula el costo total solo al final.
    """
    start, goal = get_start_and_goal()
    h0 = manhattan(start, goal)
    heap = [(h0, start, start)]  # (h, etiqueta, nodo)
    parent = {}
    visited = set()
    g_cost = {start: 0}          # para reportar costo del camino hallado
```

```
    while heap:
        h, _, u = heapq.heappop(heap)
        if u in visited:
            continue
        visited.add(u)

        if u == goal:
            path = reconstruct_path(parent, goal)
            return path, g_cost[u]

        for v in neighbors(u):
            if v in visited:
                continue
            # acumulamos costo real solo para informar al final
            new_g = g_cost[u] + enter_cost(v)
            if v not in g_cost or new_g < g_cost[v]:
                g_cost[v] = new_g
                parent[v] = u
            hv = manhattan(v, goal)
            heapq.heappush(heap, (hv, v, v))  # tie-break alfabético

    return None, None

# =========================
# 3) A* (mínimo costo total)
# =========================
def astar():
    """
    A*: f = g + h (Manhattan), desempate alfabético.
    Minimiza costo total (g suma costo de entrada; W=30).
    """
    start, goal = get_start_and_goal()
    g_cost = {start: 0}
    f0 = g_cost[start] + manhattan(start, goal)
    heap = [(f0, 0, start, start)]  # (f, g, etiqueta, nodo)
    parent = {}
    best_g = {start: 0}
    closed = set()

    while heap:
        f, g, _, u = heapq.heappop(heap)
        if u in closed:
            continue
        closed.add(u)

        if u == goal:
            path = reconstruct_path(parent, goal)
```

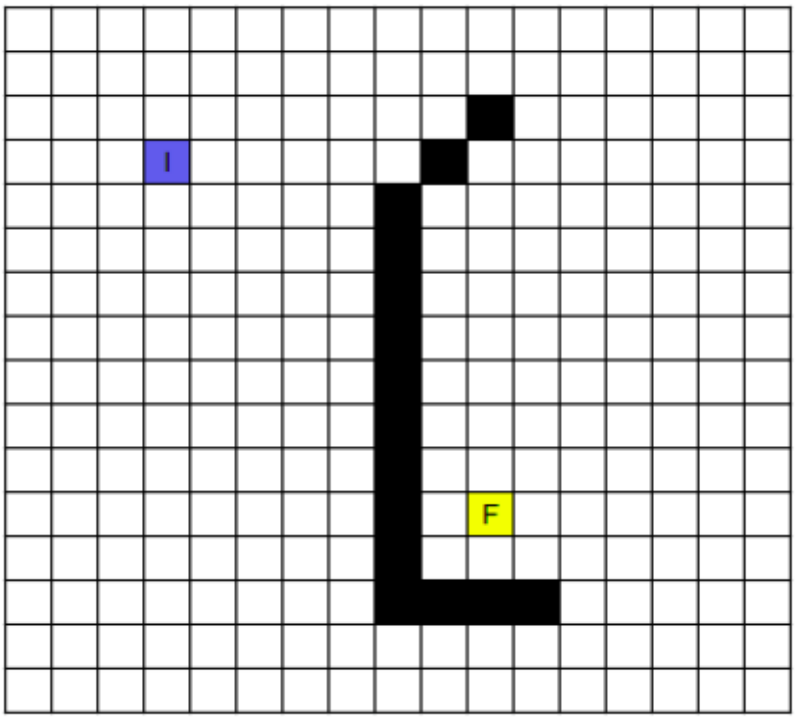
```
            return path, g_cost[u]

        for v in neighbors(u):
            new_g = g_cost[u] + enter_cost(v)
            if v not in best_g or new_g < best_g[v]:
                best_g[v] = new_g
                g_cost[v] = new_g
                parent[v] = u
                new_f = new_g + manhattan(v, goal)
                heapq.heappush(heap, (new_f, new_g, v, v))

    return None, None

# =========================
# Demo
# =========================
if __name__ == "__main__":
    print("== Profundidad ==")
    p,c = dfs_alfabetico()
    print("camino:", p, " | costo (informativo):", c)

    print("\n== Avara ==")
    p,c = greedy_best_first()
    print("camino:", p, " | costo:", c)

    print("\n== A* ==")
    p,c = astar()
    print("camino:", p, " | costo:", c)
```

```
== Profundidad ==
camino: ['I', 'G', 'P', 'Q', 'R', 'T', 'K', 'C', 'A', 'B', 'D', 'M', 'F']  |
costo (informativo): 12

== Avara ==
camino: ['I', 'Q', 'R', 'T', 'K', 'M', 'F']  | costo: 6

== A* ==
camino: ['I', 'Q', 'R', 'T', 'K', 'M', 'F']  | costo: 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.

[7]:
```
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()
```



[18]:
```
import numpy as np
import matplotlib.pyplot as plt
from heapq import heappush, heappop


# ------------------------ A* básico ---------------------------

def manhattan(a, b):
    """Heurística admisible (y consistente) para 4 direcciones."""
    return abs(a[0] - b[0]) + abs(a[1] - b[1])


def vecinos4(r, c, N):
    """Generador de vecinos en 4 direcciones dentro del tablero N×N."""
    for dr, dc in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        nr, nc = r + dr, c + dc
        if 0 <= nr < N and 0 <= nc < N:
            yield (nr, nc)
```

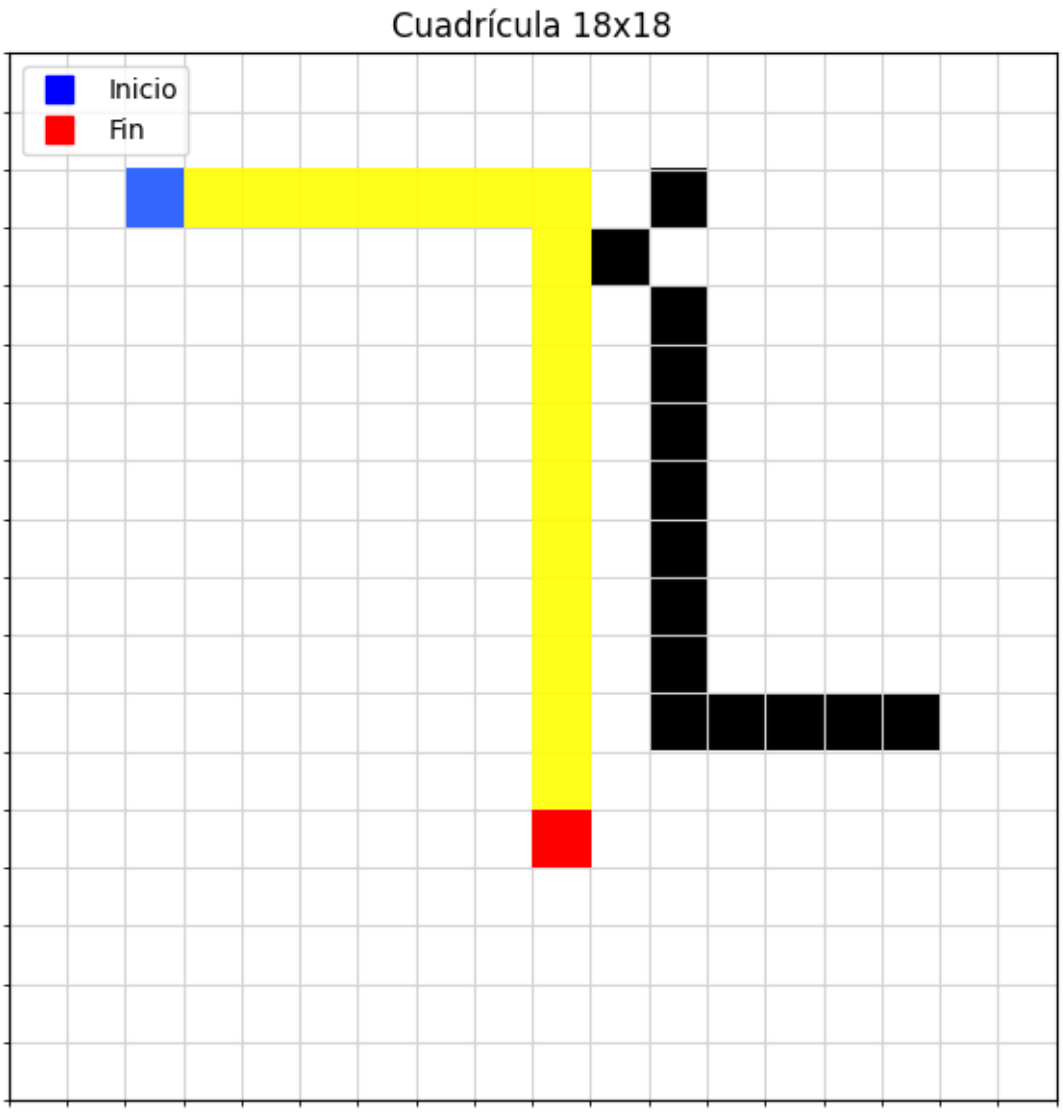
```python
def astar(grilla, inicio_rc, fin_rc):
    """
    A* sobre una grilla binaria (0=libre, 1=pared).
    Retorna lista de celdas (r, c) desde inicio a fin, o None si no hay camino.
    """
    N = grilla.shape[0]
    start, goal = inicio_rc, fin_rc

    # Conjunto abierto como heap de (f, g, nodo)
    openh = []
    heappush(openh, (manhattan(start, goal), 0, start))

    came_from = {}          # para reconstruir al llegar a goal
    gscore = {start: 0}     # mejor g conocido para cada nodo
    cerrado = set()         # nodos ya expandidos

    while openh:
        f, g, actual = heappop(openh)
        if actual in cerrado:
            continue
        if actual == goal:
            # reconstrucción del camino
            path = [actual]
            while actual in came_from:
                actual = came_from[actual]
                path.append(actual)
            path.reverse()
            return path

        cerrado.add(actual)
        r, c = actual
        for nr, nc in vecinos4(r, c, N):
            if grilla[nr, nc] == 1:   # pared => no transitable
                continue
            nuevo_g = gscore[actual] + 1
            if (nr, nc) not in gscore or nuevo_g < gscore[(nr, nc)]:
                gscore[(nr, nc)] = nuevo_g
                came_from[(nr, nc)] = actual
                fscore = nuevo_g + manhattan((nr, nc), goal)
                heappush(openh, (fscore, nuevo_g, (nr, nc)))

    return None  # sin solución


# ------------------ Entrada y marcado de puntos ----------------

def leer_entero(mensaje, minimo, maximo):
```

```
    """Pide un entero en [minimo, maximo] hasta que sea válido."""
    while True:
        try:
            v = int(input(mensaje))
            if minimo <= v <= maximo:
                return v
            print(f"Valor fuera de rango. Debe estar entre {minimo} y {maximo}.
 ↪")
        except ValueError:
            print("Ingresá un número entero, por favor.")


def pedir_y_marcar_inicio_fin(grilla, ax):
    """
    Pide (x_ini, y_ini, x_fin, y_fin), los convierte a (fila, col),
    y pinta inicio (azul) y fin (rojo) dentro de su celda.
    Devuelve (inicio_rc, fin_rc).
    """
    N = grilla.shape[0]
    print("\nCoordenadas (x,y) en 0..17 con origen ARRIBA-IZQUIERDA.")
    xi = leer_entero("x inicial: ", 0, N - 1)
    yi = leer_entero("y inicial: ", 0, N - 1)
    xf = leer_entero("x final:   ", 0, N - 1)
    yf = leer_entero("y final:   ", 0, N - 1)

    inicio = (yi, xi)   # (fila, columna)
    fin = (yf, xf)

    if grilla[inicio] == 1:
        print("  Atención: el INICIO cae sobre una pared (se marcará igual).")
    if grilla[fin] == 1:
        print("  Atención: el FIN cae sobre una pared (se marcará igual).")

    # Máscara RGBA del tamaño del tablero para pintar celdas completas
    mask = np.zeros((N, N, 4), dtype=float)
    mask[yi, xi] = [0.2, 0.4, 1.0, 1.0]  # azul
    mask[yf, xf] = [1.0, 0.0, 0.0, 1.0]  # rojo

    # Usamos 'extent' fijo para que todas las capas encuadren igual
    ax.imshow(mask, origin='upper', interpolation='nearest',
              extent=[-0.5, N - 0.5, N - 0.5, -0.5], zorder=3)

    # Leyenda sin dibujar puntos extra
    ax.plot([], [], 's', color='blue',  markersize=10, label='Inicio')
    ax.plot([], [], 's', color='red',   markersize=10, label='Fin')
    ax.legend(loc='upper left')
```

```
    return inicio, fin


# --------------- Pintado del camino como máscara ----------------

def dibujar_camino(ax, path, N, color=(1.0, 1.0, 0.0, 0.9), zorder=2):
    """
    Pinta el 'path' sobre una máscara RGBA N×N (amarillo por defecto).
    Usa 'extent' fijo para no recortar la figura.
    """
    if not path:
        return
    mask = np.zeros((N, N, 4), dtype=float)
    for r, c in path:
        mask[r, c] = color

    ax.imshow(mask, origin='upper', interpolation='nearest',
              extent=[-0.5, N - 0.5, N - 0.5, -0.5], zorder=zorder)


# ------------------------------ main ----------------------------

def main():
    N = 18
    grilla = np.zeros((N, N), dtype=int)  # 0 = libre, 1 = pared

    # --------- Definí tu pared (coordenadas en formato (fila, col)) ---------
    pared = [
        (2, 11), (3, 10),                 # "tapita" como en tu diseño
        (4, 11), (5, 11), (6, 11), (7, 11), (8, 11),
        (9, 11), (10, 11), (11, 11),      # tramo vertical
        (11, 12), (11, 13), (11, 14), (11, 15)  # tramo horizontal
    ]
    for (r, c) in pared:
        grilla[r, c] = 1

    # ------------------- Dibujo base de la grilla -------------------
    fig, ax = plt.subplots(figsize=(6, 6))

    # Fondo: la propia grilla (0=blanco, 1=negro)
    ax.imshow(grilla, origin='upper', interpolation='nearest',
              cmap='gray_r', vmin=0, vmax=1,
              extent=[-0.5, N - 0.5, N - 0.5, -0.5])

    # Líneas de cuadrícula
    ax.set_xticks(np.arange(-0.5, N, 1), minor=True)
    ax.set_yticks(np.arange(-0.5, N, 1), minor=True)
```

```
    ax.grid(which='minor', color='lightgray', linestyle='-', linewidth=0.8)
    ax.set_xticks([]); ax.set_yticks([])
    ax.set_title("Cuadrícula 18x18")

    # ----------------- Inicio / fin (entrada + pintado) -----------------
    inicio_rc, fin_rc = pedir_y_marcar_inicio_fin(grilla, ax)

    # -------------------- A* y pintado del camino ----------------------
    camino = astar(grilla, inicio_rc, fin_rc)
    if camino is None:
        print("No hay camino  (la pared bloquea la ruta entre inicio y fin).")
    else:
        camino_interno = [p for p in camino if p not in (inicio_rc, fin_rc)]
        dibujar_camino(ax, camino_interno, N=N)  # amarillo

    plt.tight_layout()
    plt.show()


# Punto de entrada
if __name__ == "__main__":
    main()
```

```
Coordenadas (x,y) en 0..17 con origen ARRIBA-IZQUIERDA.
Ingresá un número entero, por favor.
```



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