INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO
INGENIERIA EN SISTEMAS COMPUTACIONALES

Inteligencia Artificial

Proyecto 1. Búsqueda informada

“15 puzzle Empleando búsqueda informada”

Presentan

Gómez Hernández, Alan Javier
Hernández Pérez, Juan Manuel
Jiménez Cruz, Daniel
Rivas Carrera, Diana Laura

Docente

Andrés García Floriano

noviembre de 2023

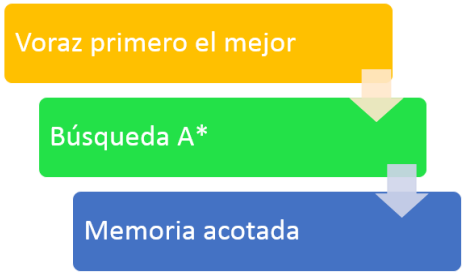
INTRODUCCIÓN

BUSQUEDA INFORMADA.

La búsqueda informada, también conocida como heurística, es un método que utiliza información específica del problema para encontrar soluciones de manera más eficiente. La búsqueda informada utiliza una heurística para obtener información del problema. La heurística mide el costo estimado más barato desde un nodo hasta un nodo objetivo. Los métodos de búsqueda heurística exploran en primer lugar los caminos más prometedores. En general, el uso de la heurística no garantiza encontrar una solución óptima, pero sí una solución de buena calidad en tiempo razonable.

Algunos algoritmos de búsqueda informados son:

- A: * Busca el camino más corto desde un estado inicial al estado meta a través de un espacio de problema.
- Voraz: Se guía exclusivamente por la función heurística.
- Escalada: Se guía por el coste real de desplazarse de un nodo a otro.



Las estrategias de búsqueda informada son mucho más eficientes que las no informadas. La búsqueda ciega o no informada sólo utiliza información acerca de si un estado es o no objetivo para guiar su proceso de búsqueda. La búsqueda informada en inteligencia artificial es un enfoque estratégico que incorpora conocimiento adicional para resolver problemas de búsqueda de manera eficiente. A diferencia de la búsqueda no informada, que explora el espacio de búsqueda sin información específica, la búsqueda informada utiliza heurísticas para dirigir la exploración hacia las áreas más prometedoras. Estas heurísticas proporcionan estimaciones aproximadas sobre la distancia o calidad de las soluciones potenciales.

¿QUÉ ES LA HEURÍSTICA?

Heurística significa encontrar, hallar o descubrir, esta palabra proviene del griego “heuriskein”.

La heurística es muy utilizado en los algoritmos de búsquedas puestos que estas ayudan a guiar el proceso de búsqueda, y permiten obtener soluciones de buena calidad, aunque no sean siempre las mejores, pero consiguen buenos resultados en la media de tiempo que las búsquedas emplea.

VENTAJAS DE UTILIZAR LA HEURÍSTICA

Debido a que la heurística, ayuda a resolver problemas complejos, se presentan las siguientes ventajas:

Generalmente para problemas complejos no necesitamos siempre obtener la solución óptima, solo se necesita resultados buenos.

Utilizando la heurística, no vamos a encontrar casos críticos, ya que siempre da soluciones.

Deducir cómo funciona la heurística, nos da un conocimiento mayor de los problemas que queremos resolver.

Distancia manhattan.

En una ciudad como Nueva York, formada en gran medida por una matriz de edificios, la distancia entre dos puntos nunca es la distancia "en línea recta". Más bien viene determinada por los bordes

de los bloques que hay que rodear. Consideremos el siguiente diagrama en el que hay dos puntos (en negro) cuya distancia queremos calcular:
Aun cuando la distancia "ordinaria" sea la correspondiente a la línea verde, si los recuadros blancos son edificios, estaremos obligados a bordearlos. En estas condiciones, tanto el camino rojo como el azul y el amarillo son equivalentes y representan la distancia Manhattan mínima entre los puntos. Esta distancia viene definida por la siguiente expresión:

$$distancia = \sum |x - y|$$

En IA tenemos su uso para:

1. **Heurística en Algoritmos de Búsqueda Informada:**
   - *A (A-star):** En el algoritmo A*, que es un algoritmo de búsqueda informada, la distancia de Manhattan se puede utilizar como una heurística para estimar el costo desde un nodo dado hasta el objetivo. Esta heurística es admissible (nunca sobreestima el costo) y es eficiente de calcular.
2. **Problemas de Optimización en IA:**
   - En problemas de optimización, donde se busca minimizar o maximizar una función objetivo, la distancia de Manhattan puede ser utilizada como parte de la función de costo. Por ejemplo, en problemas de enrutamiento o logística, puede representar la distancia total que se debe recorrer.
3. **Reconocimiento de Patrones:**
   - En problemas de reconocimiento de patrones, la distancia de Manhattan se puede utilizar para medir la similitud entre dos patrones. Esto puede ser útil en tareas como reconocimiento de imágenes o reconocimiento de voz.
4. **Juegos de Tablero:**
   - En juegos de tablero como el ajedrez, la distancia de Manhattan puede ser una medida útil para evaluar la proximidad de las piezas entre sí. Esto se puede utilizar en evaluaciones heurísticas para tomar decisiones en juegos basados en inteligencia artificial.
5. **Optimización de Rutas y Planificación de Movimiento:**
   - En robótica y sistemas autónomos, la distancia de Manhattan se puede utilizar para planificar rutas eficientes, especialmente cuando el movimiento solo está permitido en direcciones ortogonales (arriba, abajo, izquierda, derecha).

DESARROLLO

En grupos de hasta cinco personas, resuelvan alguno de los siguientes problemas:

- 15 puzzle
  - Empleando búsqueda informada
- Gato o Tic Tac Toe de 4x4
  - Empleando minimax o poda alfa beta.

Justificación.

Aunque el "15 puzzle" es un juego, su estructura de búsqueda y resolución puede extenderse para abordar problemas de la vida real que involucren estados y acciones. Esto

hace que la aplicación de la búsqueda informada en este contexto tenga relevancia en la resolución de problemas más complejos basados en estados y transiciones. Al elegir este problema, se facilita la comprensión y enseñanza de conceptos clave de búsqueda informada, haciendo que la elección sea relevante para entornos educativos.

Código

librerías que se ocupan:

```
import heapq  # Importa el módulo heapq para manejar una cola de prioridad.
import time  # Importa el módulo time para medir la duración de la ejecución.
import numpy as np  # Importa numpy para trabajar eficientemente con matrices.
```

Esta clase será utilizada para representar nodos individuales en el espacio de búsqueda del algoritmo A*

```
class Node:
    # Representa un nodo en el espacio de búsqueda.
    def __init__(self, state, parent, g, h):
        # Inicializa un nodo.
        self.state = state  # Estado actual del rompecabezas.
        self.parent = parent  # Nodo padre en el camino de búsqueda.
        self.g = g  # Costo desde el estado inicial hasta el actual.
        self.h = h  # Estimación heurística desde el estado actual hasta el objetivo.
        self.f = g + h  # Costo total estimado (f = g + h).

    def __lt__(self, other):
        # Define la comparación de menor que para la cola de prioridad.
        return self.f < other.f
```

Distancia de Manhattan=$|x1-x2|+|y1-y2|$
la distancia de Manhattan es la distancia que uno tendría que recorrer para ir desde un punto hasta otro moviéndose solo en direcciones horizontales y verticales, sin moverse en diagonal.

```
def manhattan_distance(state):
    # Calcula la distancia de Manhattan como heurística.
    distance = 0
    for i in range(4):
        for j in range(4):
            if state[i, j] != 0:
                # Para cada número, calcula la distancia a su posición objetivo.
                target_i, target_j = TARGET_POS[state[i, j]]
                distance += abs(i - target_i) + abs(j - target_j)
    return distance
```

Esta función determina los posibles movimientos que se pueden realizar desde el estado actual del rompecabezas, evitando movimientos que regresen a la posición anterior de la ficha vacía. La función devuelve una lista de tuplas, donde cada tupla contiene el nuevo estado después del

movimiento y las coordenadas de la ficha vacía en el estado anterior.

```
    def get_possible_moves(state, last_zero_pos):
        # Devuelve los posibles movimientos desde el estado actual.
        possible_moves = []
        i, j = np.where(state == 0)  # Encuentra la posición de la pieza vacía (0).
        i, j = i[0], j[0]

        # Explora los movimientos en las cuatro direcciones.
        for di, dj in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            ni, nj = i + di, j + dj
            # Verifica si el movimiento es válido y evita regresar a la posición anterior.
            if 0 <= ni < 4 and 0 <= nj < 4:
                if last_zero_pos is None or (ni, nj) != last_zero_pos:
                    new_state = np.copy(state)
                    # Realiza el movimiento intercambiando la pieza vacía con la adyacente.
                    new_state[i, j], new_state[ni, nj] = new_state[ni, nj], new_state[i, j]
                    possible_moves.append((new_state, (i, j)))
        return possible_moves
```

Este bloque de código implementa el algoritmo de búsqueda A* para resolver el problema del "15 puzzle".

```
def a_star_search(initial_state):
    # Implementa el algoritmo de búsqueda A*.
    pq = []  # Cola de prioridad para los nodos a explorar.
    initial_node = Node(initial_state, None, 0, manhattan_distance(initial_state))
    heapq.heappush(pq, initial_node)  # Agrega el nodo inicial a la cola.
    explored = set()  # Conjunto para almacenar los estados ya explorados.

    while pq:
        current_node = heapq.heappop(pq)  # Toma el nodo con el menor costo f.
        current_state = current_node.state

        # Verifica si el estado actual es el estado objetivo.
        if np.array_equal(current_state, TARGET):
            path = []
            # Reconstruye el camino desde el estado inicial hasta el objetivo.
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]  # Devuelve el camino en orden correcto.

        # Agrega el estado actual a los explorados.
        explored.add(str(current_state.flatten()))
```

```
28
29          # Determina la posición de la pieza vacía en el nodo padre.
30          last_zero_pos = None
31          if current_node.parent:
32              parent_zero_pos = np.where(current_node.parent.state == 0)
33              if parent_zero_pos[0].size > 0:
34                  last_zero_pos = (parent_zero_pos[0][0],
35  parent_zero_pos[1][0])
36
37          # Explora los movimientos posibles desde el estado actual.
38          for move, _ in get_possible_moves(current_state,
39  last_zero_pos):
                if str(move.flatten()) in explored:
                    continue
                # Crea un nuevo nodo y lo agrega a la cola de prioridad.
                new_node = Node(move, current_node, current_node.g + 1,
    manhattan_distance(move))
                heapq.heappush(pq, new_node)

      return None  # Retorna None si no se encuentra una solución.
```

Aquí dará la información sobre la solución encontrada para el "15 puzzle" con el estado inicial proporcionado, incluyendo cada paso de la solución y el tiempo total de ejecución del algoritmo.

```
   if __name__ == '__main__':
1      # Define el estado inicial y ejecuta la búsqueda A*.
 2     initial_state = np.array([[11, 2, 13, 8], [5, 6, 0, 4], [9, 10, 7,
 3  12], [3, 14, 1, 15]])
 4
 5     start_time = time.time()  # Inicia el cronómetro.
 6     solution_path = a_star_search(initial_state)  # Ejecuta la
 7  búsqueda A*.
 8     end_time = time.time()  # Detiene el cronómetro.
 9
10     # Imprime cada paso de la solución y estadísticas de la ejecución.
11     print("Pasos para llegar al objetivo: \n")
12     for state in solution_path:
13         print(state, "\n")
14
15     print(f"Solución encontrada en {len(solution_path)} pasos")
       print(f"Tiempo de ejecución: {end_time - start_time} segundos")
```

Código completo:

```
1
2  import heapq  # Importa el módulo heapq para manejar una cola de
3  prioridad.
4  import time  # Importa el módulo time para medir la duración de la
5  ejecución.
6  import numpy as np  # Importa numpy para trabajar eficientemente con
7  matrices.
8
9  # Define el estado objetivo del rompecabezas como una matriz 4x4.
```

```
TARGET = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13,
14, 15, 0]])

# Crea un diccionario para mapear cada número a su posición objetivo.
TARGET_POS = {TARGET[i, j]: (i, j) for i in range(4) for j in
range(4)}

class Node:
    # Representa un nodo en el espacio de búsqueda.
    def __init__(self, state, parent, g, h):
        # Inicializa un nodo.
        self.state = state  # Estado actual del rompecabezas.
        self.parent = parent  # Nodo padre en el camino de búsqueda.
        self.g = g  # Costo desde el estado inicial hasta el actual.
        self.h = h  # Estimación heurística desde el estado actual
hasta el objetivo.
        self.f = g + h  # Costo total estimado (f = g + h).

    def __lt__(self, other):
        # Define la comparación de menor que para la cola de
prioridad.
        return self.f < other.f

def manhattan_distance(state):
    # Calcula la distancia de Manhattan como heurística.
    distance = 0
    for i in range(4):
        for j in range(4):
            if state[i, j] != 0:
                # Para cada número, calcula la distancia a su
posición objetivo.
                target_i, target_j = TARGET_POS[state[i, j]]
                distance += abs(i - target_i) + abs(j - target_j)
    return distance

def get_possible_moves(state, last_zero_pos):
    # Devuelve los posibles movimientos desde el estado actual.
    possible_moves = []
    i, j = np.where(state == 0)  # Encuentra la posición de la pieza
vacía (0).
    i, j = i[0], j[0]

    # Explora los movimientos en las cuatro direcciones.
    for di, dj in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
        ni, nj = i + di, j + dj
        # Verifica si el movimiento es válido y evita regresar a la
posición anterior.
        if 0 <= ni < 4 and 0 <= nj < 4:
            if last_zero_pos is None or (ni, nj) != last_zero_pos:
                new_state = np.copy(state)
                # Realiza el movimiento intercambiando la pieza vacía
con la adyacente.
```

```
                new_state[i, j], new_state[ni, nj] = new_state[ni,
nj], new_state[i, j]
                possible_moves.append((new_state, (i, j)))
    return possible_moves

def a_star_search(initial_state):
    # Implementa el algoritmo de búsqueda A*.
    pq = []  # Cola de prioridad para los nodos a explorar.
    initial_node = Node(initial_state, None, 0,
manhattan_distance(initial_state))
    heapq.heappush(pq, initial_node)  # Agrega el nodo inicial a la
cola.
    explored = set()  # Conjunto para almacenar los estados ya
explorados.

    while pq:
        current_node = heapq.heappop(pq)  # Toma el nodo con el menor
costo f.
        current_state = current_node.state

        # Verifica si el estado actual es el estado objetivo.
        if np.array_equal(current_state, TARGET):
            path = []
            # Reconstruye el camino desde el estado inicial hasta el
objetivo.
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1]  # Devuelve el camino en orden
correcto.

        # Agrega el estado actual a los explorados.
        explored.add(str(current_state.flatten()))

        # Determina la posición de la pieza vacía en el nodo padre.
        last_zero_pos = None
        if current_node.parent:
            parent_zero_pos = np.where(current_node.parent.state ==
0)
            if parent_zero_pos[0].size > 0:
                last_zero_pos = (parent_zero_pos[0][0],
parent_zero_pos[1][0])

        # Explora los movimientos posibles desde el estado actual.
        for move, _ in get_possible_moves(current_state,
last_zero_pos):
            if str(move.flatten()) in explored:
                continue
            # Crea un nuevo nodo y lo agrega a la cola de prioridad.
            new_node = Node(move, current_node, current_node.g + 1,
  manhattan_distance(move))
            heapq.heappush(pq, new_node)
```

```
    return None  # Retorna None si no se encuentra una solución.

if __name__ == '__main__':
    # Define el estado inicial y ejecuta la búsqueda A*.
    initial_state = np.array([[11, 2, 13, 8], [5, 6, 0, 4], [9, 10,
7, 12], [3, 14, 1, 15]])

    start_time = time.time()  # Inicia el cronómetro.
    solution_path = a_star_search(initial_state)  # Ejecuta la
búsqueda A*.
    end_time = time.time()  # Detiene el cronómetro.

    # Imprime cada paso de la solución y estadísticas de la
ejecución.
    print("Pasos para llegar al objetivo: \n")
    for state in solution_path:
        print(state, "\n")

    print(f"Solución encontrada en {len(solution_path)} pasos")
    print(f"Tiempo de ejecución: {end_time - start_time} segundos")
```

PRUEBAS DE FUNCIONAMIENTO

1. **Definición del Estado Inicial:**
   - Se define un estado inicial específico del rompecabezas "15 puzzle" en la variable **initial_state**.
2. *Ejecución de la Búsqueda A:**
   - Se inicia el cronómetro antes de ejecutar la búsqueda A*.
   - La función **a_star_search** se llama con el estado inicial, lo que devuelve el camino de solución.
   - Se detiene el cronómetro después de completar la búsqueda.
3. **Impresión de Resultados:**
   - Se imprime cada paso del camino de solución.
   - Se imprime la longitud del camino de solución (número de pasos).
   - Se imprime el tiempo total de ejecución de la búsqueda A*.

```
 [[ 1  2  3  4]
  [ 5  6  7  8]
  [10  0 15 11]
  [ 9 13 14 12]]

 [[ 1  2  3  4]
  [ 5  6  7  8]
  [ 0 10 15 11]
  [ 9 13 14 12]]

 [[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 15 11]
  [ 0 13 14 12]]

 [[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 15 11]
  [13  0 14 12]]

 [[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 15 11]
  [13 14  0 12]]

 [[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10  0 11]
  [13 14 15 12]]

 [[ 1  2  3  4]
  [ 5  6  7  8]
  [ 9 10 11  0]
  [13 14 15 12]]
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15  0]]

Solución encontrada en 20 pasos
Tiempo de ejecución: 0.6388096809387207 segundos
```

```
# Define el estado objetivo del rompecabezas como una matriz 4x4.
TARGET = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13,
14, 15, 0]])
```

```
 # Define el estado inicial y ejecuta la búsqueda A*.
    initial_state = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11,
12], [0, 14, 15, 13]])
```

1. Clase Node:
La clase **Node** representa un nodo en el espacio de búsqueda. Un nodo contiene la siguiente información:

- **state**: El estado actual del rompecabezas.
- **parent**: El nodo padre en el camino de búsqueda.
- **g**: El costo desde el estado inicial hasta el estado actual.
- **h**: La estimación heurística desde el estado actual hasta el objetivo.
- **f**: El costo total estimado, dado por la suma de **g** y **h**.

2. Función de Heurística (manhattan_distance):
La función **manhattan_distance** calcula la distancia de Manhattan como una heurística. Para cada número en el estado actual, calcula la distancia a su posición objetivo y acumula estas distancias. Esta heurística es utilizada para estimar cuánto falta para llegar al estado objetivo.
3. Función de Posibles Movimientos (get_possible_moves):
La función **get_possible_moves** devuelve los posibles movimientos desde el estado actual. Examina la posición de la pieza vacía (representada por **0**) y considera movimientos en las cuatro direcciones posibles, evitando regresar a la posición anterior.
4. Función Principal (a_star_search):
La función **a_star_search** implementa el algoritmo de búsqueda A*:

- Se crea una cola de prioridad (**pq**) para nodos a explorar.
- Se inicializa el nodo inicial con el estado inicial y se agrega a la cola de prioridad.
- Se crea un conjunto (**explored**) para almacenar los estados ya explorados.

El algoritmo sigue un bucle principal mientras haya nodos en la cola de prioridad:

1. Se extrae el nodo con el menor costo total (**f**) de la cola de prioridad.
2. Si el estado actual es el estado objetivo, se reconstruye el camino desde el estado inicial hasta el objetivo y se devuelve.
3. Se agrega el estado actual a los estados explorados.
4. Se determina la posición de la pieza vacía en el nodo padre.
5. Se exploran los posibles movimientos desde el estado actual.
6. Para cada movimiento posible, se crea un nuevo nodo y se agrega a la cola de prioridad si no ha sido explorado.

CONCLUSIÓN

Para esta practica podemos asumir que para el algoritmo A* y la implementación de las funciones son correctas al proyecto, ya que da una solución con la longitud del camino, viendo si da resultados eficientes. Para el tiempo de ejecución podemos determinar que si es pesada la carga dada la depuración que tardo, quizás no uso tantos recursos, pero si tiempo, E igual esta la opción de probar el código con diferentes estados iniciales para evaluar la capacidad, dando una exploración mas a detalle sobre nodos con menor costo total. Podemos decir que con base al puzle 8 al expandirlo más se entendió la lógica de implementación también hay que decir que tuvo un grado de complejidad el solo hecho de implementar con los demás nodos que se ocupan es decir la lógica

de A* ya que usamos la distancia de Manhattan un algoritmo bien conocido. Para Node regresamos a los términos básicos de orientado a objetos que es básicamente lo importante para que interactúe con el algoritmo, la aplicación de la cola de prioridad en la exploración de nodos, así como la representación de los estados mediante la clase Node, han contribuido a la eficiencia del algoritmo. La capacidad de adaptarse a diversas configuraciones iniciales del rompecabezas subraya la versatilidad y generalidad del enfoque implementado.