lógica del algoritmo A en el 8-Puzzle

El algoritmo A* (**A-Star**) es una técnica de búsqueda informada que combina:

- **g(n)**: El costo real desde el inicio hasta el nodo actual.
- **h(n)**: Una heurística que estima el costo restante hasta la solución.
- **f(n) = g(n) + h(n)**: Suma del costo real y la heurística. Se usa para priorizar qué estados explorar.

En este caso, usamos la **heurística de Manhattan**, que mide cuántas posiciones debe moverse cada número hasta su ubicación final.

El algoritmo funciona así:

1. Se coloca el estado inicial en una **cola de prioridad** (una estructura que siempre extrae el nodo con menor f(n)).
2. Se extrae el estado con menor f(n).
3. Si el estado es el objetivo ([[1,2,3], [4,5,6], [7,8,0]]), el algoritmo termina.
4. Se generan los **estados vecinos** intercambiando la posición del "0" con sus casillas adyacentes.
5. Se calcula f(n) para cada nuevo estado y se agregan a la cola si no han sido visitados.
6. Se repite hasta encontrar la solución.

Así, el algoritmo explora primero los caminos más prometedores, evitando búsquedas innecesarias.

Explicación del programa

Estructura del código

El programa está dividido en varias partes:

- **Clase Rompecabezas**

Representa un estado del tablero del 8-Puzzle y tiene:

- **Atributos**:
    - tablero: matriz actual del estado.
    - padre: referencia al estado anterior para reconstruir la solución.
    - movimiento: último movimiento realizado.
    - profundidad: cantidad de movimientos desde el inicio.

    - costo: valor de h(n), la heurística de Manhattan.
    - posicion_vacia: coordenadas (x, y) donde está el "0".
- **Métodos**:
    - __lt__: permite comparar nodos según f(n).
    - obtener_vecinos: genera los estados vecinos posibles.
    - heuristica: calcula la distancia de Manhattan.
    - obtener_camino_tableros: reconstruye la secuencia de tableros hasta la solución.

🔹 **Función resolver_8_puzzle(tablero_inicial)**

1. Crea un nodo inicial y lo coloca en la **cola de prioridad**.
2. Usa un **conjunto visitados** para evitar repetir estados.
3. Itera extrayendo el nodo con menor f(n), verificando si es la solución.
4. Genera los vecinos del nodo actual y los agrega a la cola si no han sido explorados.
5. Si encuentra la solución, retorna la secuencia de tableros.

🔹 **Código principal**

- Define el **tablero inicial**.
- Llama a resolver_8_puzzle.
- Imprime la secuencia de tableros hasta la solución.

Explicación de la sintaxis

📌 **Librerías**

```
import heapq
import numpy as np
```

- heapq: Módulo que permite usar **colas de prioridad** (estructuras donde los elementos con menor valor salen primero).
- numpy: Librería para trabajar con **matrices** y operaciones numéricas.

📌 Constructor de la clase

```
def __init__(self, tablero, padre=None, movimiento="", profundidad=0, costo=0):
    """
    Clase que representa un estado del 8-Puzzle.
    """
    self.tablero = np.array(tablero)
    self.padre = padre
    self.movimiento = movimiento
    self.profundidad = profundidad
    self.costo = costo
    self.posicion_vacia = tuple(map(int, np.where(self.tablero == 0)))
```

- def define una **función**.
- __init__ es un **constructor**, se ejecuta automáticamente al crear un objeto de la clase.
- self: Representa al objeto mismo.
- tablero: Matriz del rompecabezas.
- padre: Estado anterior (útil para reconstruir el camino).
- movimiento: Último movimiento realizado.
- profundidad: Número de movimientos desde el inicio.
- costo: Valor de la heurística.

```
    self.tablero = np.array(tablero)
```

- Convierte tablero en una **matriz de NumPy** para facilitar cálculos.

```
    self.posicion_vacia = tuple(map(int, np.where(self.tablero == 0)))
```

- np.where(self.tablero == 0): Encuentra la posición del número 0 en la matriz.
- map(int, ...): Convierte el resultado en números enteros.
- tuple(...): Lo convierte en una **tupla** ((x, y)) para que sea fácil de usar.

📌 Método para comparación en la cola de prioridad

```
def __lt__(self, otro):
    """
    Método para comparar nodos basado en la función de costo f(n) = g(n) + h(n)
    """
    return (self.costo + self.profundidad) < (otro.costo + otro.profundidad)
```

- __lt__ (less than): Se usa para comparar objetos de la clase *puzzle_8* en la cola de prioridad.
- Devuelve True si self tiene menor f(n) (costo + profundidad) que otro.

📌 Generación de estados vecinos

```
def obtener_vecinos(self):
    """
    Genera los estados vecinos moviendo la casilla vacía.
    """
    vecinos = []
```

- Se define una lista vacía vecinos para almacenar los estados generados.

```
x, y = self.posicion_vacia
```

- x, y obtienen la posición del 0 en la matriz.

```
movimientos = {"Arriba": (x - 1, y), "Abajo": (x + 1, y), "Izquierda": (x, y - 1), "Derecha": (x, y + 1)}
```

- Se define un **diccionario** con los movimientos posibles y sus coordenadas resultantes.

```
for mov, (nx, ny) in movimientos.items():
```

- for recorre cada **clave (mov)** y **valor (nx, ny)** en el diccionario.

```
if 0 <= nx < 3 and 0 <= ny < 3:
```

- Se asegura de que (nx, ny) esté dentro de los límites de la matriz (3x3).

```
nuevo_tablero = self.tablero.copy()
```

- Se crea una copia del tablero para modificarlo sin afectar el original.

```
nuevo_tablero[x, y], nuevo_tablero[nx, ny] = nuevo_tablero[nx, ny], nuevo_tablero[x, y]
```

- Se intercambian las posiciones del 0 y el número en (nx, ny).

```
vecinos.append(puzzle_8(nuevo_tablero, self, mov, self.profundidad + 1, self.heuristica(nuevo_tablero)))
```

- Se crea un nuevo objeto Rompecabezas y se agrega a vecinos.

📌 Función heurística (Manhattan)

```
def heuristica(self, tablero):
    """
    Calcula la heurística de Manhattan: la suma de las distancias de cada número a su posición objetivo.
    """
```

- Define una función para calcular la **heurística**.

```
objetivo = {num: (i, j) for i, fila in enumerate([[1, 2, 3], [4, 5, 6], [7, 8, 0]]) for j, num in enumerate(fila)}
```

- Crea un **diccionario** con la posición final de cada número.

```
return sum(abs(x - objetivo[num][0]) + abs(y - objetivo[num][1]) for x, fila in enumerate(tablero) for y,
        num in enumerate(fila) if num)
```

- Calcula la **distancia de Manhattan** para cada número.
- abs(x - objetivo[num][0]): Diferencia en filas.
- abs(y - objetivo[num][1]): Diferencia en columnas.
- sum(...): Suma total.

📌 Construcción del camino hasta la solución

```
def obtener_camino_tableros(self):
    """
    Retorna la secuencia de tableros desde el estado inicial hasta la solución.
    """
```

- Devuelve la secuencia de tableros desde el estado inicial hasta la solución.

```
camino, nodo = [], self
```

- camino almacena la secuencia de tableros.

```
while nodo:
    camino.append(nodo.tablero)
    nodo = nodo.padre
return camino[::-1]
```

- Se recorre la cadena de nodos hasta llegar al inicial.
- [::-1] invierte la lista para que muestre el camino en orden correcto.

📌 Algoritmo principal A*

```
def resolver_8_puzzle(tablero_inicial):
    """
    Resuelve el 8-Puzzle utilizando el algoritmo A*.
    """
```

- Función que resuelve el 8-Puzzle.

```
nodo_inicial = puzzle_8(tablero_inicial, costo=0)
cola_prioridad = [nodo_inicial]
visitados = set()
```

- Se crea el **nodo inicial**.
- cola_prioridad: Lista que almacenará los estados a explorar.
- visitados: Conjunto para evitar estados repetidos.

python

CopiarEditar

```
while cola_prioridad:
    actual = heapq.heappop(cola_prioridad)

    # Verifica si se ha alcanzado el estado objetivo
    if np.array_equal(actual.tablero, [[1, 2, 3], [4, 5, 6], [7, 8, 0]]):
        return actual.obtener_camino_tableros()
```

- Repite mientras haya estados por explorar.
- heapq.heappop(...) saca el nodo con menor f(n).
- if np.array_equal(…) si el tablero es el objetivo, retorna la solución.

```
visitados.add(tuple(map(tuple, actual.tablero)))
```

- Convierte la matriz en una **tupla de tuplas** para almacenarla en visitados.

```
for vecino in actual.obtener_vecinos():
    if tuple(map(tuple, vecino.tablero)) not in visitados:
        heapq.heappush(cola_prioridad, vecino)
```

- Agrega los vecinos a la cola de prioridad si no han sido visitados.

📌 Código principal

```
# Ejemplo de uso
tablero_inicial = [[8, 5, 7], [6, 1, 4], [2, 3, 0]]
```

- Se define el tablero inicial.

```
solucion = resolver_8_puzzle(tablero_inicial)
```

- Se llama a la función resolver_8_puzzle.

```python
if solucion:
    print("Secuencia de tableros hasta la solución:")
    for paso, tablero in enumerate(solucion):
        print(f"Paso {paso}:")
        print(tablero, "\n")
else:
    print("No se encontró una solución.")
```

- Se imprime la **secuencia de tableros** hasta la solución.