Instituto Tecnológico De Culiacán

Carrera: Ingeniería en Sistemas Computacionales

Materia: Inteligencia Artificial

Zuriel Dathan Mora Félix

Título de la actividad:

Proyecto PUZZLE-8

Integrantes:

- Espíndola Leyva José Enrique
- Soto Cortez Jesús Eugenio

Horario: 09:00am – 10:00am

Localidad: Culiacán, Sinaloa

Fecha: 12/09/2025

Introducción y objetivo

A* (A-estrella) es un algoritmo común en las fuentes proporcionadas. Especialmente en relación con la solución del clásico puzzle 8. El puzzle 8 es un rompecabezas deslizante en cuadrícula con un marco de 3×3 que contiene ocho fichas numeradas del uno al ocho y una casilla vacía. El objetivo de este proyecto es mover las fichas desde el estado inicial al estado objetivo.

El algoritmo A* es un algoritmo de búsqueda informada hacia la solución óptima que aborda para encontrar el camino más corto, en términos de costo, desde el estado de inicio inicial en el estado meta final. Para lograr esto, evalúa los nodos en un grafo mediante una evaluación de la función f * n:

- f(n) = g(n) + h(n).
- g(n) representa el costo real del camino desde el nodo inicial hasta el nodo actual n.
- h(n) es una estimación heurística del costo desde el nodo actual n hasta el nodo objetivo

Heurísticamente, A* recorre primordialmente los nodos con la f más baja en n. Para propósitos de la implementación, f se define como una rutina de la Clase Nodo en el Árbol de Búsqueda. El Nodo es una de las principales componentes subyacentes del algoritmo, la misma que almacena el actual estado de la matriz, la heurística, y el costo total. La heurística se define como la distancia de Manhattan y es preferida y admisible para las instancias del juego puzzle 8 ya que no sobrestima la realidad de los requisitos para alcanzar la meta.

En un contexto más amplio de la aplicación con Tkinter, el algoritmo A* no solo se utiliza para encontrar la solución, sino que también permite animar el proceso paso a paso en la interfaz gráfica, mostrando al usuario la secuencia de movimientos óptimos. Esto resalta su papel esencial en la funcionalidad principal del software del Puzzle 8. La clase Tablero, aunque no resuelve el puzzle por sí sola, ofrece las herramientas necesarias para que un algoritmo como A* pueda hacerlo, al modelar el estado del puzzle y las reglas de movimiento.

Clase Nodo

Este script de Python implementa el algoritmo de búsqueda A* (A-estrella) para encontrar la solución óptima al clásico puzles 8. Este rompecabezas es deslizante y consta de una cuadrícula de 3x3 con 8 fichas numeradas y una ficha vacía. El objetivo es reorganizar las fichas desde una configuración inicial hasta una configuración predefinida en el menor número de movimientos posible.

El algoritmo A* es un algoritmo de búsqueda informada que garantiza encontrar el camino más corto (en términos de costo) desde un estado inicial a un estado final. Esto se logra evaluando los nodos de un grafo de búsqueda mediante una función de evaluación f(n), que combina dos elementos:

- **g(n)**: El costo real del camino desde el nodo inicial hasta el nodo actual n.
- **h(n)**: Una estimación heurística del costo desde el nodo actual n hasta el nodo objetivo.

La fórmula es: **f(n) = g(n) + h(n)**. A* explora prioritariamente los nodos con el valor f(n) más bajo.

2. Descripción de los Componentes del Código

El código está estructurado en una clase principal (Nodo) y tres funciones auxiliares (heuristica_manhattan, Intercambiar, a_estrella) que trabajan en conjunto para resolver el puzzle.

2.1. Bibliotecas Utilizadas

- **numpy**: Se utiliza para representar y manipular eficientemente las matrices (tableros) de 3x3. Facilita operaciones como la búsqueda de elementos y la copia de matrices.
- **heapq**: Proporciona una implementación de cola de prioridad (min-heap), que es fundamental para gestionar la "lista abierta" en el algoritmo A*. Permite obtener de manera eficiente el nodo con el menor valor f(n) para expandir a continuación.
- **pprint**: Aunque no se usa en la lógica principal, se importa para una posible "impresión bonita" (pretty-printing) de las estructuras de datos durante la depuración.

2.2. Clase Nodo

Esta clase es muy importante en el árbol de búsqueda. Cada instancia de Nodo representa un estado específico del tablero y almacena información importante para el algoritmo.

```python
class Nodo:
    def __init__(self, matriz, heuristica=None, costo_acumulado=0, padre=None):
        self.matriz = matriz
        self.heuristica = heuristica  # h(n): costo estimado desde este nodo al objetivo
        self.costo_acumulado = costo_acumulado  # g(n): costo desde el inicio hasta este nodo
        # f(n) = g(n) + h(n): costo total estimado
        self.f = self.costo_acumulado + self.heuristica if self.heuristica is not None else float('inf')
        self.hijos = []
        self.padre = padre
```

- **Atributos:**
  - matriz: Un array de NumPy que representa la configuración del tablero 3x3 en este nodo.
  - heuristica (h(n)): El valor heurístico calculado que estima la distancia al estado objetivo.
  - costo_acumulado (g(n)): El número de movimientos realizados desde el estado inicial para llegar a este estado.
  - f (f(n)): El costo total estimado. Es la suma de g(n) y h(n). Este es el valor que A* utiliza para priorizar qué nodo explorar.
  - hijos: Una lista para almacenar los nodos sucesores que se generan a partir de este.
  - padre: Una referencia al Nodo que generó el nodo actual. Es crucial para reconstruir el camino de la solución una vez que se alcanza el objetivo.

- **Métodos:**
  - agregarHijo(...): Crea un nuevo Nodo hijo, lo establece como hijo del nodo actual y lo devuelve.
  - __lt__(self, other): Método especial ("less than") que permite a la cola de prioridad (heapq) comparar dos objetos Nodo directamente basándose en su valor f. Esto es indispensable para que heapq funcione correctamente con objetos personalizados.

2.3. Función heuristica_manhattan

Esta función calcula la distancia de Manhattan, una heurística muy eficaz y aceptable para el puzzle 8. Una heurística es aceptable si nunca le quita la importancia al coste real de alcanzar la meta.

```
def heuristica_manhattan(estado, objetivo):
    distancia = 0
    for valor in range(1, 9):  # Itera sobre las fichas del 1 al 8
        pos_actual = np.argwhere(estado == valor)[0]
        pos_objetivo = np.argwhere(objetivo == valor)[0]
        distancia += abs(pos_actual[0] - pos_objetivo[0]) + abs(pos_actual[1] - pos_objetivo[1])
    return distancia
```

- **Funcionamiento:**
  1. Itera sobre cada ficha del puzzle (del 1 al 8).
  2. Para cada ficha, encuentra sus coordenadas (fila, columna) en el estado actual y en el estado objetivo.
  3. Calcula la distancia de Manhattan para esa ficha, que es la suma de la diferencia absoluta de las filas y la diferencia absoluta de las columnas.
  4. Suma las distancias de todas las fichas. El resultado total es el valor heurístico h(n) del estado actual. Representa el número mínimo de movimientos que cada ficha necesitaría para llegar a su posición final si no hubiera otras fichas bloqueando el camino.

2.4. Función Intercambiar

Una función de utilidad simple pero importante para generar nuevos estados del tablero.

```
def Intercambiar(matriz, row1, col1, row2, col2):
    nueva_matriz = matriz.copy()
    nueva_matriz[row1, col1], nueva_matriz[row2, col2] = nueva_matriz[row2, col2], nueva_matriz[row1, col1]
    return nueva_matriz
```

- **Funcionamiento:**
  1. Recibe una matriz y las coordenadas de dos casillas del tablero.
  2. Crea una copia de la matriz. Esto es muy importante para que el nodo padre no sufra de cambios que después no se puedan restaurar.
  3. Intercambia los valores en las dos coordenadas especificadas.
  4. Devuelve la nueva matriz que tuvo cambios, para represente un nuevo estado del puzzle.

2.5. Función a_estrella

Es el corazón del programa, donde se lleva a cabo la lógica del algoritmo A*.

```
def a_estrella(inicio, objetivo):
```

- Estructuras de Datos Clave:

○ **lista_abierta:** Una cola de prioridad (min-heap) que contiene los nodos que han sido descubiertos pero que aún no se han evaluado. El nodo con el menor valor f siempre está en la parte superior.

○ **lista_cerrada**: Un conjunto (set) que guarda los estados que ya han sido evaluados. Usar un conjunto permite verificar la existencia de un estado de manera muy eficiente (tiempo O(1) en promedio).

○ **costo_g_abierta:** Un diccionario que asocia un estado (convertido a tupla para que sea "hashable") a su costo g más bajo encontrado hasta el momento. Esto es fundamental para evitar rutas subóptimas y ciclos.

- **Flujo del Algoritmo:**

1. **Inicialización:** Se crea el nodo inicial, se calcula su heurística y se añade a la lista_abierta. Su costo g es 0.
2. **Bucle Principal:** Mientras la lista_abierta no esté vacía:
   a. **Selección:** Se extrae el nodo con el menor f de la lista_abierta. Este se convierte en nodo_actual.
   b. **Prueba de Objetivo**: Se verifica si nodo_actual es el estado objetivo. Si lo es, hemos encontrado la solución y se devuelve el nodo.
   c. **Expansión:** Se mueve el estado de nodo_actual a la lista_cerrada para no volver a evaluarlo.
   d. Generación de Sucesores:
   i. Se encuentra la posición del espacio vacío (0).
   ii. Se determinan todos los movimientos válidos (arriba, abajo, izquierda, derecha).
   iii. Para cada movimiento válido, se genera una nueva_matriz usando la función Intercambiar.
   e. **Evaluación de Sucesores:** Para cada sucesor generado:
   i. Verificar en Lista Cerrada: Si el sucesor ya está en la lista_cerrada, se ignora.
   ii. Calcular Cost

3. Bloque de Ejecución

Esta es la sección final del script donde se define el problema y se invoca al algoritmo.

```
# Configuración inicial
matriz_inicial = np.array([
    [0, 6, 3],
    [2, 5, 4],
    [7, 1, 8]
])

matriz_objetivo = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
])

# Ejecutar A*
solucion = a_estrella(matriz_inicial, matriz_objetivo)
```

Se establecen las matrices de NumPy para representar el estado inicial y el estado objetivo del puzzle.

Luego, se invoca la función a_estrella utilizando estas dos matrices.

La variable solución almacenará el Nodo objetivo final si se encuentra una solución. Desde este nodo, se puede reconstruir el camino completo hacia atrás siguiendo los punteros .padre hasta llegar al nodo inicial. Si no se encuentra solución, solución será None.

Clase Interfaz

En esta sección se explica de manera clara el funcionamiento e iteración del código proporcionado para una aplicación de escritorio del "Puzzle 8", desarrollada con la librería tkinter de Python. La aplicación permite al usuario configurar un estado inicial y un estado objetivo para el puzzle, y luego utiliza el algoritmo de búsqueda A* para encontrar y animar la solución.

Estructura General del Proyecto

El código depende de varias clases que trabajan en conjunto:

1. **interfaz.py:** Contiene toda la lógica de la interfaz gráfica de usuario (GUI) construida con tkinter. Esta clase se encarga de la interacción con el usuario y la interfaz, la visualización del tablero y la orquestación de la solución.
2. **Nodo3.py:** Debe contener la implementación del algoritmo a_estrella, así como las matrices por defecto matriz_objetivo y matriz_inicial.
3. **Tablero.py:** Esta clase contiene una clase Tablero que encapsula la lógica y el estado del puzzle, aunque en el código principal su uso es mínimo.

Análisis del Código Principal

A continuación, se detalla cada sección de la clase interfaz:

1. Importaciones y Variables Globales

```
import tkinter as tk
from tkinter import simpledialog, messagebox
import random
import numpy as np
from Nodo3 import a_estrella, matriz_objetivo, matriz_inicial
from Tablero import Tablero
```

- tkinter: Es la biblioteca estándar de Python que se utiliza para crear interfaces gráficas de usuario.
- simpledialog, messagebox: Son módulos de tkinter que permiten mostrar ventanas emergentes para diálogos e información.
- random: Se emplea para generar un estado inicial aleatorio en el tablero.
- numpy: Se utiliza para transformar listas de Python en arrays de NumPy, un formato que probablemente necesite la función a_estrella.
- Nodo3 y Tablero: Son módulos personalizados del proyecto que contienen la lógica del algoritmo y la estructura de datos del tablero.

2. Clase App

```
class App(tk.Tk):
    def __init__(self, *args, **kwargs):
        tk.Tk.__init__(self, *args, **kwargs)
        self.geometry('800x800')
        self.container = tk.Frame(self, bg='red')
        self.container.place(relx=0, rely=0, relheight=1, relwidth=1)
        fTab = Frame_Tablero(self.container, self)
        fTab.tkraise()
```

- Propósito: Esta es la clase principal de la aplicación. Hereda de tk.Tk, convirtiéndose en la ventana raíz.
- __init__: El constructor de la clase.
- tk.Tk.__init__(self, ...): Llama al constructor de la clase padre para inicializar la ventana.
- self.geometry('800x800'): Define el tamaño inicial de la ventana.
- self.container: Crea un Frame (un contenedor) que ocupa toda la ventana y servirá como base para otros widgets.
- fTab = Frame_Tablero(...): Crea una instancia de Frame_Tablero, que es donde se encuentra toda la lógica del juego.
- fTab.tkraise(): Asegura que el frame del tablero esté visible por encima de cualquier otro.

3. Clase Ficha

```
class Ficha:
    contador = 0
    def __init__(self, r, c, n, frame):
        self.frame = frame
        self.r = r
        self.c = c
        self.n = n
        self.contador = Ficha.contador
        Ficha.contador += 1
        if n != 0:
            self.button = tk.Button(self.frame, text=str(self.n), font=("Impact", 100),
                                    command=lambda: frame.move(self.contador, self.r, self.c))
        else:
            self.button = tk.Button(self.frame, text='', font=("Impact", 100),
                                    command=lambda: frame.move(self.contador, self.r, self.c))
        self.button.place(relx=1/26+self.c*(4/13), rely=0.05+self.r*(1/4), relheight=1/4, relwidth=4/13)
```

- Propósito: Representa una única ficha (o baldosa) del puzzle en la interfaz gráfica.
- contador: Es una variable de clase que asigna un ID único a cada ficha que se crea.
- __init__(self, r, c, n, frame):

- r, c: Se refieren a la fila (row) y columna (column) de la ficha en la matriz.
- n: Es el número que aparece en la ficha (0 indica que es el espacio vacío).
- frame: Es la instancia de Frame_Tablero a la que pertenece esta ficha.
- self.button: Cada ficha se representa como un tk.Button.
  - Si n no es 0, el botón mostrará el número.
  - Si n es 0, el botón no tendrá texto.
  - command=lambda: frame.move(...): Al hacer clic en el botón, se invoca el método move del Frame_Tablero, pasándole su ID, fila y columna.
- self.button.place(...): Coloca el botón en el frame utilizando coordenadas relativas para que se ajuste al tamaño de la ventana.

4. Clase Frame_Tablero

Esta es la clase más crucial, ya que alberga toda la lógica de la aplicación.

__init__(self, parent, root)

- **Propósito:** Configura el marco principal del tablero, los botones de control y el estado inicial del rompecabezas.
- **Botones:**
  - b_solve: Botón "Resolver" que pone en marcha el algoritmo A*.
  - b_edit: Botón "Editar Inicial" que activa el modo de edición del tablero inicial.
  - b_rand: Botón "Aleatorio Inicial" que crea una configuración aleatoria.
  - b_goal: Botón "Editar Objetivo" que permite ajustar el estado final deseado.
- **Estado del Puzzle:**
  - self.nums: Una matriz 3x3 que refleja el estado actual del tablero. Se inicializa con matriz_inicial.
  - self.objetivo: Una matriz 3x3 que representa la meta final. Por defecto, es la configuración ordenada.
- **Creación de Fichas:**
  - self.fichas: Una lista que guarda las 9 instancias de la clase Ficha. Se recorre la matriz self.nums y se crea un objeto Ficha para cada celda.
- **Control de Modo:**
  - self.modo: Una variable de estado ("normal", "edicion_inicial", "edicion_objetivo") que define cómo reacciona la aplicación a los clics en las fichas.
  - self.seleccion: Almacena las coordenadas de la primera ficha seleccionada en el modo de edición.

5. Bloque de Ejecución Principal

```
if __name__ == "__main__":
    app = App()
    app.title("Puzzle 8 con IA A*")
    app.mainloop()
```

- if __name__ == "__main__":: Esta es una construcción estándar en Python que asegura que el código dentro de este bloque solo se ejecute cuando el script es el programa principal.
- app = App(): Crea una instancia de la ventana principal de la aplicación.
- app.title(...): Establece el título de la ventana.
- app.mainloop(): Inicia el bucle de eventos de tkinter, que espera las interacciones del usuario (clics, etc.) y mantiene la ventana abierta.

Clase tablero

El código comienza por establecer dos variables globales. Estas funcionan como constantes de configuración que definen el estado final del rompecabezas y las reglas de movimiento, lo que las hace fácilmente accesibles desde cualquier parte del programa. El código comienza por establecer dos variables globales. Estas funcionan como constantes de configuración que definen el estado final del rompecabezas y las reglas de movimiento, lo que las hace fácilmente accesibles desde cualquier parte del programa.

sol = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

Esta variable representa de manera canónica el estado objetivo o la solución del rompecabezas. Es el "destino" al que cualquier algoritmo de búsqueda intentará llegar.

Esta variable representa de manera canónica el estado objetivo o la solución del rompecabezas. Es el "destino" al que cualquier algoritmo de búsqueda intentará llegar.

moves_list
Esta es una estructura de datos que se ha precalculado, una optimización de diseño fundamental. Guarda todos los movimientos posibles para la casilla vacía

(0) en cada una de las 9 posiciones que puede haber en el tablero.

```
moves_list = [
    [['r', 'd'], ['l', 'r', 'd'], ['l', 'd']],
    [['r', 'u', 'd'], ['l', 'r', 'u', 'd'], ['l', 'u', 'd']],
    [['r', 'u'], ['l', 'r', 'u'], ['l', 'u']]
]
```

Esto permite:

Optimización y Eficiencia: En lugar de calcular los movimientos válidos en tiempo de ejecución (lo que implicaría hacer comprobaciones de límites como if fila > 0, if columna < 2, etc.), este método utiliza una tabla de búsqueda. Dado que las reglas de movimiento en un tablero de 3x3 son fijas, podemos precalcularlas. Esto aligera la carga computacional durante la ejecución del algoritmo de búsqueda, que llamará a esta función miles o incluso millones de veces

Ejemplo Detallado:

moves_list[0][0]: representa la esquina superior izquierda (fila 0, columna 0). Desde aquí, el espacio vacío solo puede moverse a la derecha ('r') o hacia abajo ('d').

moves_list[1][1]: se refiere al centro del tablero (fila 1, columna 1). Esta posición ofrece la máxima libertad, permitiendo movimientos en las cuatro direcciones: izquierda, derecha, arriba y abajo.

2. La Clase Tablero: Modelando el Estado del Puzzle

Esta clase es una gran demostración de la Programación Orientada a Objetos (POO). Agrupa tanto los datos (como el estado del tablero, self.nums) como las operaciones que se pueden llevar a cabo sobre esos datos (los métodos como make_move, moves, etc.) en una única entidad lógica.

Método __init__ (Constructor)

Este método es el constructor de la clase. Se invoca automáticamente al crear una nueva instancia (mi_tablero = Tablero(...)) y su función es inicializar los atributos del objeto.

```
    def __init__(self, nums=[[1, 2, 3], [4, 5, 6], [7, 8, 0]]):
        """
        Inicializa el tablero del puzzle.
        :param nums: Una matriz de 3x3 que representa el estado inicial del
```

tablero.

Por defecto, es el estado de la solución.

"""

self.nums = nums

self: Es una referencia a la instancia actual del objeto. Permite acceder a los atributos y métodos de la clase dentro de ella misma (ej. self.nums).

Parámetro por defecto: El argumento nums tiene un valor por defecto que es la matriz de la solución. Esto permite crear un objeto Tablero sin proporcionar un estado inicial (t = Tablero()), lo cual es útil para pruebas. Para iniciar un juego real, se le pasaría una matriz desordenada.

Atributo self.nums: Este es el "estado" del objeto. Almacena la configuración actual de las piezas en el tablero.

Método empty()

Una función de utilidad interna crucial, ya que todas las acciones se definen en relación con la posición de la casilla vacía.

```
def empty(self):
    """
    Encuentra las coordenadas de la casilla vacía (el 0).
    :return: Una tupla (fila, columna) con la posición del 0.
    """
    for ir, r in enumerate(self.nums):
        for ic, c in enumerate(r):
            if c == 0:
                return ir, ic
```

Enumerate: Esta función de Python es una manera de recorrer una secuencia, obteniendo tanto el índice como el valor en cada paso. En este caso, "ir" representa el índice de la fila y "r" es la fila misma, que se presenta como una lista.

Método moves()

Este método es la interfaz pública para la tabla de búsqueda moves_list.

```
def moves(self):
    """
    Obtiene los movimientos válidos para la casilla vacía desde su posición
```

actual.

:return: Una lista de strings con los movimientos posibles (ej. ['r', 'd']).
"""
r, c = self.empty()
return moves_list[r][c]

Abstraccion: Este método simplifica la complejidad de cómo se determinan los movimientos. Un usuario de la clase Tablero no necesita preocuparse por moves_list; simplemente llama a .moves() y recibe la lista de acciones válidas.

Método make_move()

Este método hace la manipulación del estado del tablero.

```
def make_move(self, dir):
    """
    Realiza un movimiento en el tablero, intercambiando la casilla vacía
    con una pieza adyacente.
    :param dir: La dirección del movimiento ('l', 'r', 'u', 'd').
    """
    r, c = self.empty()
    # ... lógica de intercambio ...
```

Lógica de Intercambio (Swap): Mover una pieza siempre implica un intercambio de posiciones con la casilla vacío.

Ejemplo:

El movimiento hacia la izquierda (dir == 'l'):

1. aux = self.nums[r][c]: Guardamos el valor de la celda actual (que es 0) en una variable temporal llamada aux.

2. self.nums[r][c] = self.nums[r][c - 1]: Ahora, la posición de la celda vacía toma el valor de la celda a su izquierda.

3. self.nums[r][c - 1] = aux: La celda de la izquierda ahora recibe el valor que estaba en aux (el 0).