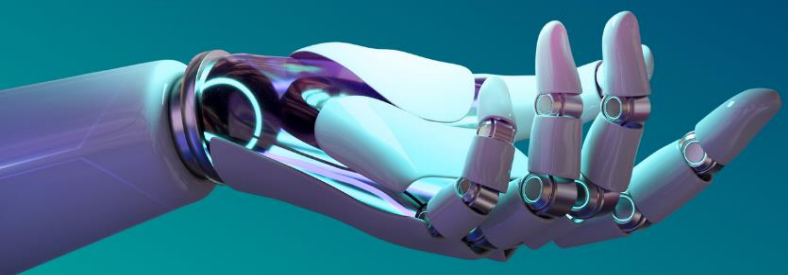
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

UNIVERSIDAD DE GUADALAJARA

PRÁCTICA 2

PUZZLE - AMPLITUD



Alumnos:

Carbajal Armenta Yessenia Poala

Leon Estrada Rafael

Medina Bolaños Danna Paola

Códigos:

220286482

220286121

220286938

Maestro:

Oliva Navarro Diego Alberto

Materia:

Inteligencia Artificial

Sección:

D05

Guadalajara, Jal. a 21 de abril del 2023

Práctica 2 – Puzzle - Amplitud

Instrucciones

Problema 1 del documento Practica 2_SI. pdf utilizando **la búsqueda en amplitud**.

Código

```
import numpy as np
from collections import deque
from copy import deepcopy
import pygame
import random
```

Aquí se importan las bibliotecas necesarias para el juego, como numpy, deque y pygame. Numpy es una biblioteca para Python que se utiliza para realizar operaciones matemáticas, como matrices y vectores. Deque es una estructura de datos que se utiliza en este juego para mantener una lista ordenada de las matrices a explorar en el algoritmo de búsqueda en amplitud. deepcopy se utiliza para crear una copia de una matriz y evitar que se modifique la matriz original. Pygame es una biblioteca que se utiliza para crear interfaces gráficas de usuario en Python. Por último, random se utiliza para generar números aleatorios.

```
# Constantes para seleccionar el algoritmo de búsqueda
BUSQUEDA_AMPLITUD = 0
BUSQUEDA_PROFUNDIDAD = 1
IDDFS = 2
ALGORITMO = IDDFS # Cambia esta constante para elegir el algoritmo:
BUSQUEDA_AMPLITUD, BUSQUEDA_PROFUNDIDAD o IDDFS
```

Aquí se definen las constantes para los diferentes algoritmos de búsqueda no informada que se pueden utilizar. **BUSQUEDA_AMPLITUD** se refiere a la búsqueda en amplitud, **BUSQUEDA_PROFUNDIDAD** a la búsqueda en profundidad y **IDDFS** a la búsqueda en profundidad iterativa. La constante **ALGORITMO** se utiliza para seleccionar qué algoritmo se utilizará para resolver el juego.

```
# Inicializar Pygame, crear una ventana de 300x300 y establecer el título
pygame.init()
pantalla = pygame.display.set_mode((300, 300))
pygame.display.set_caption("Resolviendo problema 8-puzzle")
fuente = pygame.font.Font(None, 36)
reloj = pygame.time.Clock()
```

Aquí se definen las constantes que se utilizarán para seleccionar el algoritmo de búsqueda. Las opciones son búsqueda en amplitud, búsqueda en profundidad y búsqueda en profundidad iterativa (IDDFS). Por defecto, se utiliza IDDFS.

```
# Definir colores como constantes para usar en el dibujo
BLANCO = (255, 255, 255)
NEGRO = (0, 0, 0)
```

La función **convertir_a_cadena** se utiliza para convertir una matriz en una cadena única para facilitar la comparación. En este caso, se utiliza para comparar las matrices y determinar si se ha alcanzado la solución.

```
# Convertir la matriz en una cadena única para facilitar la comparación
def convertir_a_cadena(matriz):
    return ''.join(str(e) for fila in matriz for e in fila)
```

Esta función se utiliza para convertir una matriz en una cadena única de caracteres. Se utiliza para comparar las matrices en la exploración de los algoritmos de búsqueda.

```
# Generar todas las matrices posibles a partir de la matriz actual moviendo
una ficha adyacente a la posición vacía
def expandir(matriz):
    # Encontrar la posición de la ficha vacía (0) en la matriz
    vacio_x, vacio_y = np.where(matriz == 0)
    vacio_x, vacio_y = int(vacio_x), int(vacio_y)

    # Definir los posibles movimientos de la ficha vacía: arriba, abajo,
izquierda, derecha
    movimientos = [
        (-1, 0),  # Arriba
```

```
        (1, 0),   # Abajo
        (0, -1),  # Izquierda
        (0, 1)    # Derecha
    ]

    # Iterar sobre los posibles movimientos
    for dx, dy in movimientos:
        # Calcular la nueva posición de la ficha vacía después de moverla
        nuevo_x, nuevo_y = vacio_x + dx, vacio_y + dy

        # Comprobar si la nueva posición está dentro de los límites de la
matriz (3x3)
        if 0 <= nuevo_x < 3 and 0 <= nuevo_y < 3:
            # Crear una copia de la matriz actual para no modificar la
matriz original
            nueva_matriz = deepcopy(matriz)
             # Intercambiar la posición de la ficha vacía con la posición
adyacente en la dirección del movimiento
            nueva_matriz[vacio_x, vacio_y] = nueva_matriz[nuevo_x, nuevo_y]
            nueva_matriz[nuevo_x, nuevo_y] = 0
            # Devolver la nueva matriz generada después de mover la ficha
vacía
            yield nueva_matriz
```

La función **expandir(matriz)** genera todas las posibles matrices que se pueden obtener a partir de la matriz dada **matriz** moviendo una ficha adyacente a la posición vacía.

Primero se encuentra la posición de la ficha vacía en la matriz. Luego, se definen los posibles movimientos que se pueden hacer con la ficha vacía: arriba, abajo, izquierda, derecha.

Después, se itera sobre estos posibles movimientos y se calcula la nueva posición de la ficha vacía después de hacer el movimiento. Se verifica que la nueva posición esté dentro de los límites de la matriz (3x3).

Si es así, se crea una copia de la matriz original para no modificarla y se intercambia la posición de la ficha vacía con la posición adyacente en la dirección del movimiento. Se genera una nueva matriz a partir de esta operación, utilizando el generador **yield**.

```
# Realizar búsqueda en amplitud para encontrar la solución del 8-puzzle
def busqueda_amplitud(matriz):
    visitados = set()
```

```
    cola = deque([(matriz, 0, [])])

    while cola:
        matriz_actual, profundidad, camino = cola.popleft()
        cadena_actual = convertir_a_cadena(matriz_actual)

        if cadena_actual not in visitados:
            visitados.add(cadena_actual)

            if np.all(matriz_actual == matriz_objetivo):
                return profundidad, camino

            for siguiente_matriz in expandir(matriz_actual):
                nuevo_camino = camino.copy()
                nuevo_camino.append(siguiente_matriz)
                cola.append((siguiente_matriz, profundidad + 1,
nuevo_camino))

    return -1, []
```

Este bloque de código realiza la búsqueda en amplitud para encontrar la solución del 8-puzzle. Se utiliza una cola para implementar la búsqueda en amplitud. En cada iteración, se obtiene la matriz actual de la cabeza de la cola y se expande para obtener todas las matrices posibles que se pueden obtener moviendo una ficha adyacente a la posición vacía. Luego se comprueba si la matriz actual es igual a la matriz objetivo y se devuelve la profundidad y el camino si es así. Si la matriz actual no es igual a la matriz objetivo, se añaden todas las matrices posibles obtenidas a la cola, junto con su profundidad y su camino. Este proceso continúa hasta que se encuentra la matriz objetivo o la cola se vacía. Si no se encuentra la matriz objetivo, se devuelve un valor negativo para la profundidad y una lista vacía para el camino.

```
# Dibujar la matriz en la pantalla utilizando Pygame
def dibujar_matriz(matriz):
    pantalla.fill(BLANCO)

    for i, fila in enumerate(matriz):
        for j, valor in enumerate(fila):
            if valor != 0:
                rect = pygame.Rect(j * 100 , i * 100, 100, 100)
                pygame.draw.rect(pantalla, NEGRO, rect, 3)
                texto = fuente.render(str(valor), True, NEGRO)
```

```
                rect_texto = texto.get_rect(center=rect.center)
                pantalla.blit(texto, rect_texto)

    pygame.display.flip()

# Mostrar un mensaje en la pantalla mientras se procesa la solución
def mostrar_mensaje_procesando():
    pantalla.fill(BLANCO)
    texto = fuente.render("Procesando solución...", True, NEGRO)
    rect_texto = texto.get_rect(center=(150, 150))
    pantalla.blit(texto, rect_texto)
    pygame.display.flip()
```

Estas dos funciones están relacionadas con la interfaz gráfica de usuario del juego. La función **dibujar_matriz(matriz)** se encarga de dibujar la matriz en la pantalla utilizando la librería Pygame. Recibe como parámetro una matriz de 3x3 que representa el estado actual del juego y dibuja cada número en su respectiva celda. El valor 0 se considera como la celda vacía y no se dibuja nada en ella. También dibuja los bordes de cada celda utilizando la función **pygame.draw.rect()**.

La función **mostrar_mensaje_procesando()** muestra un mensaje en la pantalla indicando que se está procesando la solución del juego. Se utiliza para dar feedback al usuario mientras se busca la solución y se actualiza la pantalla con el mensaje utilizando la función **pygame.display.flip()**.

```
# Animar la solución del 8-puzzle paso a paso
def animar_solucion(solucion):
    for matriz in solucion:
        dibujar_matriz(matriz)
        pygame.display.update()
        reloj.tick(5)

        for evento in pygame.event.get():
            if evento.type == pygame.QUIT:
                pygame.quit()
                return
```

La función **animar_solucion** recibe como parámetro una lista de matrices que representan la solución del 8-puzzle y las dibuja en la pantalla utilizando la función **dibujar_matriz**. Luego de cada dibujo, se actualiza la pantalla con la función

pygame.display.update() y se espera un tiempo de 5 milisegundos con **reloj.tick(5)**. Además, se verifica si el usuario ha presionado el botón de salir de la ventana de Pygame.

```
# Generar una matriz inicial aleatoria para el 8-puzzle
def generar_matriz_inicial():
    numeros = list(range(9))
    random.shuffle(numeros)
    return np.array(numeros).reshape(3, 3)
```

la función **generar_matriz_inicial** genera una matriz aleatoria para iniciar el juego del 8-puzzle. Primero crea una lista con los números del 0 al 8, luego los desordena con **random.shuffle(numeros)** y finalmente crea una matriz de 3x3 con los números desordenados utilizando la función **np.array(numeros).reshape(3, 3)**.

```
# Definir la matriz objetivo del 8-puzzle
matriz_objetivo = np.array([
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
])
```

Esta parte del código define la matriz objetivo del 8-puzzle, la cual representa el estado final que se desea alcanzar en la solución del juego. La matriz objetivo es una matriz 3x3 que contiene los números del 0 al 8, donde el 0 representa la casilla vacía y los demás números representan las casillas ocupadas por las fichas.

```
# Resolver el 8-puzzle utilizando el algoritmo seleccionado
def resolver_puzzle(algoritmo):
    while True:
        matriz_inicial = generar_matriz_inicial()
        print("Matriz inicial:")
        print(matriz_inicial)

        mostrar_mensaje_procesando()

        if algoritmo == BUSQUEDA_AMPLITUD:
            num_movimientos, solucion = busqueda_amplitud(matriz_inicial)
```

```
        elif algoritmo == BUSQUEDA_PROFUNDIDAD:
            num_movimientos, solucion = iddfs(matriz_inicial)
        elif algoritmo == IDDFS:
            num_movimientos, solucion = iddfs(matriz_inicial)

        if num_movimientos != -1:
            break

    print("Número de movimientos:", num_movimientos)
    return matriz_inicial, solucion
```

Esta parte del código resuelve el 8-puzzle utilizando el algoritmo seleccionado. Primero, se genera una matriz inicial aleatoria utilizando la función **generar_matriz_inicial()**. Luego, se muestra un mensaje en la pantalla mientras se procesa la solución utilizando la función **mostrar_mensaje_procesando()**. A continuación, se llama a la función correspondiente según el algoritmo seleccionado (**busqueda_amplitud()**, **iddfs()** o **iddfs()**). Si se encuentra una solución, se rompe el bucle y se devuelve la matriz inicial y la solución encontrada. Si no se encuentra una solución, se genera una nueva matriz inicial y se vuelve a intentar. Finalmente, se imprime el número de movimientos necesarios para resolver el puzzle y se devuelve la matriz inicial y la solución encontrada.

```
# Obtener la matriz inicial y la solución del 8-puzzle
matriz_inicial, solucion = resolver_puzzle(ALGORITMO)
solucion.insert(0, matriz_inicial)
animar_solucion(solucion)

# Bucle principal para mantener la ventana abierta y mostrar la solución
final
ejecutando = True
while ejecutando:
    for evento in pygame.event.get():
        if evento.type == pygame.QUIT:
            ejecutando = False

    dibujar_matriz(solucion[-1])
    pygame.display.update()
    reloj.tick(60)

pygame.quit()
```

Esta parte del código ejecuta el programa completo del 8-puzzle utilizando el algoritmo seleccionado (que se define previamente en la constante ALGORITMO).

Primero se llama a la función resolver_puzzle() para obtener la matriz inicial y la solución del puzzle utilizando el algoritmo seleccionado. Luego, se inserta la matriz inicial en la lista de solución y se llama a la función animar_solucion() para mostrar la solución paso a paso.

Después, se inicia un bucle principal que mantiene la ventana abierta y muestra la solución final del puzzle. Este bucle actualiza constantemente la pantalla con la última matriz de la solución y utiliza un reloj para mantener una tasa de refresco constante de 60 FPS. El bucle termina cuando el usuario cierra la ventana. Finalmente, se llama a la función pygame.quit() para cerrar la ventana y salir del programa.

Ejecución


```
Matriz inicial:
[[4 2 8]
 [7 6 1]
 [3 5 0]]
Número de movimientos: 22
PS D:\UDG\7mo Semestre\Inteligencia Artificial I\Práctica 2 Puzzle>
```

Conclusiones

El algoritmo de búsqueda en amplitud es una técnica para recorrer y buscar en un grafo o árbol. Una de sus principales ventajas es que siempre encuentra la solución óptima (en términos de profundidad o número de pasos necesarios), si existe. Además,

garantiza que, si hay varias soluciones óptimas, encontrará la que requiera menos pasos.

En el caso del juego del 8-puzzle, el algoritmo de búsqueda en amplitud es muy útil porque el espacio de búsqueda es relativamente pequeño. Dado que hay un número limitado de movimientos posibles para cada ficha, la cantidad de estados alcanzables es finita y por lo tanto es posible explorarlos todos en un tiempo razonable.

Sin embargo, una de las principales desventajas del algoritmo de búsqueda en amplitud es que puede requerir mucho espacio en memoria. En el peor de los casos, se necesitaría almacenar todos los nodos en la frontera (es decir, todos los nodos que han sido visitados pero cuyos sucesores aún no han sido explorados), lo que puede ser un problema en problemas con espacios de búsqueda muy grandes.

En general, el algoritmo de búsqueda en amplitud es una buena opción para problemas en los que el espacio de búsqueda es relativamente pequeño y se desea encontrar la solución óptima. Sin embargo, puede haber casos en los que otros algoritmos (como la búsqueda en profundidad) sean más adecuados dependiendo de la complejidad del problema.