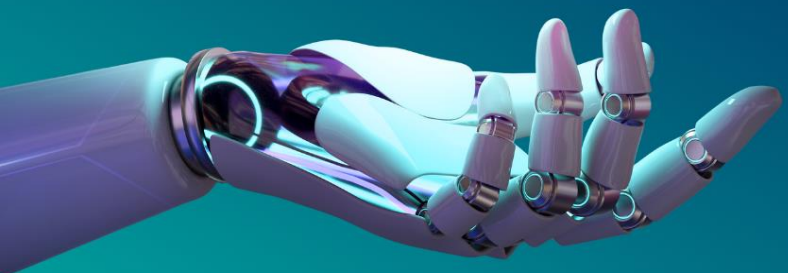
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

UNIVERSIDAD DE GUADALAJARA

PRÁCTICA 2

PUZZLE - PROFUNDIDAD Y PROFUNDIDAD ITERATIVA



Alumnos:

Carbajal Armenta Yessenia Paola

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 - Profundidad y Profundidad Iterativa

Instrucciones

Problema 1 del documento Practica 2_SI. pdf utilizando **la búsqueda en profundidad y profundidad iterativa**.

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 profundidad iterativa para encontrar la solución del 8-puzzle
def iddfs(matriz):
    profundidad_max = 0
    while True:
        visitados = set()
        camino = []
        resultado, subcamino = dfs(matriz, 0, profundidad_max, visitados, camino)
        if resultado != -1:
            return resultado, subcamino
        profundidad_max += 1

# Realizar búsqueda en profundidad con límite de profundidad para encontrar la solución del 8-puzzle
def dfs(matriz, profundidad, profundidad_max, visitados, camino):
    cadena_actual = convertir_a_cadena(matriz)
    if cadena_actual in visitados:
        return -1, []

    visitados.add(cadena_actual)

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

    if profundidad == profundidad_max:
        return -1, []

    for siguiente_matriz in expandir(matriz):
        nuevo_camino = camino.copy()
        nuevo_camino.append(siguiente_matriz)
        resultado, subcamino = dfs(siguiente_matriz, profundidad + 1, profundidad_max, visitados, nuevo_camino)
        if resultado != -1:
            return resultado, subcamino

    return -1, []
```

En esta sección del código se implementa la búsqueda en profundidad iterativa para encontrar la solución del 8-puzzle.

La función **iddfs** tiene como parámetro una matriz que representa el estado inicial del 8-puzzle. Esta función utiliza un bucle while para realizar una búsqueda en profundidad iterativa, es decir, se realiza una búsqueda en profundidad con límites de profundidad crecientes hasta encontrar la solución.

En cada iteración del bucle, se inicializan un conjunto de nodos visitados y una lista de nodos visitados llamada "camino". Luego, se llama a la función **dfs** con los parámetros de la matriz inicial, la profundidad actual (que es 0), la profundidad máxima (que es 0 en la primera iteración), el conjunto de nodos visitados y la lista de nodos visitados. La función **dfs** busca la solución del 8-puzzle utilizando búsqueda en profundidad limitada.

La función **dfs** toma como parámetros la matriz actual, la profundidad actual, la profundidad máxima, el conjunto de nodos visitados y la lista de nodos visitados. Primero, la función convierte la matriz actual en una cadena y comprueba si ya ha sido visitada. Si la matriz ya ha sido visitada, la función devuelve **-1** y una lista vacía. Si la matriz actual es igual a la matriz objetivo, la función devuelve la profundidad actual y la lista de nodos visitados.

Si la profundidad actual es igual a la profundidad máxima, la función devuelve **-1** y una lista vacía. En caso contrario, la función expande la matriz actual para obtener todas las posibles matrices siguientes, y para cada una de ellas, llama a la función **dfs** con la profundidad actual incrementada en 1, la profundidad máxima, el conjunto de nodos visitados actualizado y la lista de nodos visitados actualizada con la nueva matriz. Si la función **dfs** devuelve una profundidad mayor que -1, significa que se ha encontrado la solución y se devuelve la profundidad y la lista de nodos visitados. En caso contrario, se sigue iterando.

En resumen, la función **iddfs** realiza una búsqueda en profundidad iterativa y la función **dfs** realiza una búsqueda en profundidad limitada.

```
# 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()
```

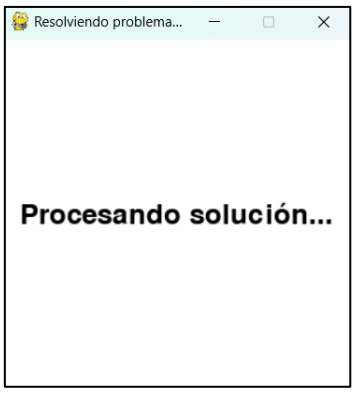
Esta parte del código ejecuta el programa completo del 8-puzzle utilizando el algoritmo seleccionado (que se define previamente en la constante ALGORITMO).

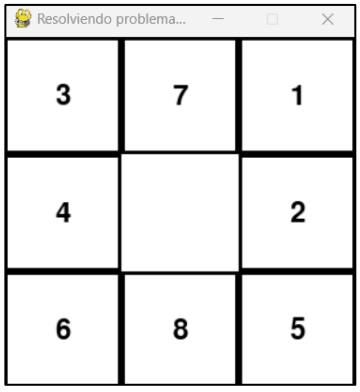
Primero se llama a la función resolver_puzzle() para obtener la matriz inicial y la solución del puzzle utilizando el algoritmo seleccionado. Luego, se inserta la matriz inicial en la lista de solución y se llama a la función animar_solucion() para mostrar la solución paso a paso.

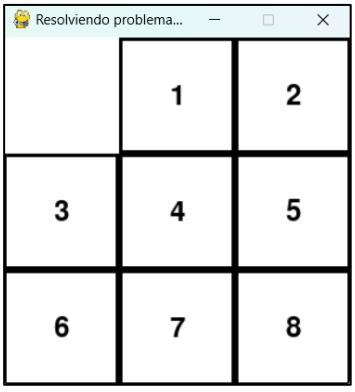
Después, se inicia un bucle principal que mantiene la ventana abierta y muestra la solución final del puzzle. Este bucle actualiza constantemente la pantalla con la última matriz de la solución y utiliza un reloj para mantener una tasa de refresco constante de 60 FPS. El bucle termina cuando el usuario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

Ejecución

Búsqueda en Profundidad:

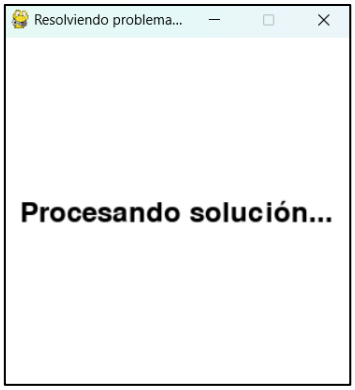


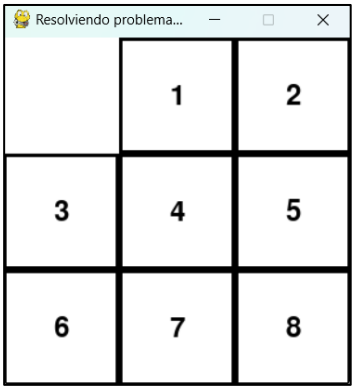





```
PS D:\UDG\7mo Semestre\Inteligencia Artificial I\Práctica 2 Puzzle> & D:/Python/python.exe "d:/UDG/7mo Semestre/Inteligencia Artificial I/Práctica 2 Puzzle/Practica_
2_Puzzle_BFS_DFS_IDDFS.py"
pygame 2.3.0 (SDL 2.24.2, Python 3.10.7)
Hello from the pygame community. https://www.pygame.org/contribute.html
Matriz inicial:
[[6 5 7]
 [1 4 0]
 [2 8 3]]
Número de movimientos: 37
```

Búsqueda en Profundidad Iterativa:






```
PS D:\UDG\7mo Semestre\Inteligencia Artificial I\Práctica 2 Puzzle> & D:/Python/python.exe "d:/UDG/7mo Semestre/Inteligencia Artificial I/Práctica 2 Puzzle/Practica_
2_Puzzle_BFS_DFS_IDDFS.py"
pygame 2.3.0 (SDL 2.24.2, Python 3.10.7)
Hello from the pygame community. https://www.pygame.org/contribute.html
Matriz inicial:
[[0 3 4]
 [1 7 5]
 [2 6 8]]
Número de movimientos: 24
```

Conclusiones

La búsqueda en profundidad es un algoritmo de búsqueda no informado que recorre el grafo en profundidad, es decir, explora el máximo posible cada rama del árbol antes de retroceder. En este algoritmo se utiliza una pila para almacenar los nodos visitados. Entre sus ventajas se encuentra su simplicidad de implementación y su capacidad para encontrar soluciones rápidamente en espacios de búsqueda con profundidades pequeñas. Sin embargo, una de sus principales desventajas es que puede quedarse atascado en ciclos infinitos y que no siempre encuentra la solución óptima.

Por otro lado, la búsqueda en profundidad iterativa es una variante de la búsqueda en profundidad que utiliza una búsqueda iterativa con profundidades incrementales, en lugar de una profundidad fija. De esta manera, el algoritmo es capaz de encontrar soluciones en espacios de búsqueda más grandes sin quedarse atascado en ciclos infinitos. Entre sus ventajas se encuentran su capacidad para encontrar la solución óptima y su menor uso de memoria en comparación con otros algoritmos. Sin embargo, su principal desventaja es su lentitud en espacios de búsqueda profundos.

En el caso específico del juego del 8-puzzle, ambos algoritmos pueden ser utilizados para encontrar la solución del puzzle. Sin embargo, dado que el espacio de búsqueda es relativamente pequeño, no es necesario utilizar un algoritmo tan complejo como el IDDFS. En general, se recomienda utilizar la búsqueda en profundidad si se busca una

solución rápida y no se requiere la solución óptima, mientras que se recomienda utilizar el IDDFS si se busca la solución óptima y el espacio de búsqueda es relativamente grande.