Presentación:

Practica 3 Primero el mejor

Nombre: Farfán de León José Osvaldo

Código: 214796622

Materia: Inteligencia Artificial I

Sección: “D02”

Profesor: Oliva Navarro Diego Alberto

Fecha: 30/11/2022

Farfán de león José Osvaldo

Primero el mejor

Para dar solución a problemas presentados en la practica 2 se debe entregar un reporte y el código funcional.

Objetivo:

Implementar los algoritmos de búsqueda no informada en problemas de prueba para poder comparar su desempeño.

Implementación:

Desarrollar un programa que encuentre la mejor solución a los problemas planteados, usando los algoritmos de búsqueda no informada vistos en clase (búsqueda en profundidad, búsqueda en amplitud, etc).

Se debe hacer el planteamiento de los problemas, en base a los conceptos: del espacio de estados las acciones (función sucesora), test objetivo y el costo del camino. Con base a esto, se definen los árboles y la estrategia de búsqueda.

Problemas:

1. Implementar la búsqueda en amplitud y en profundidad para dar solución de forma automática al problema 8-puzzle (Fig. 1).

El estado inicial debe ser aleatorio.

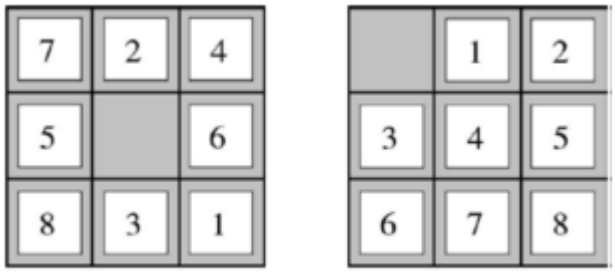


Fig.1 Ejemplo del problema 8 Puzzle

```
from copy import deepcopy
from colorama import Fore, Back, Style

DIRECTIONS = {"U": [-1, 0], "D": [1, 0], "L": [0, -1], "R": [0, 1]}     # estas
son las instrucciones para mover el espacio vacio
END = [[0, 1, 2],       # este es el resultado esperado
       [3, 4, 5],
       [6, 7, 8]]
```

```
# unicode para dibujar la cajita
left_down_angle = '\u2514'
right_down_angle = '\u2518'
right_up_angle = '\u2510'
left_up_angle = '\u250C'

middle_junction = '\u253C'
top_junction = '\u252C'
bottom_junction = '\u2534'
right_junction = '\u2524'
left_junction = '\u251C'

bar = Style.BRIGHT + Fore.CYAN + '\u2502' + Fore.RESET + Style.RESET_ALL
dash = '\u2500'

first_line = Style.BRIGHT + Fore.CYAN + left_up_angle + dash + dash + dash +
top_junction + dash + dash + dash + top_junction + dash + dash + dash +
right_up_angle + Fore.RESET + Style.RESET_ALL
middle_line = Style.BRIGHT + Fore.CYAN + left_junction + dash + dash + dash +
middle_junction + dash + dash + dash + middle_junction + dash + dash + dash +
right_junction + Fore.RESET + Style.RESET_ALL
last_line = Style.BRIGHT + Fore.CYAN + left_down_angle + dash + dash + dash +
bottom_junction + dash + dash + dash + bottom_junction + dash + dash + dash +
right_down_angle + Fore.RESET + Style.RESET_ALL


# funcion que imprime el puzzle
def print_puzzle(array):
    print(first_line)
    for a in range(len(array)):
        for i in array[a]:
            if i == 0:
                print(bar, Back.RED + ' ' + Back.RESET, end=' ')
            else:
                print(bar, i, end=' ')
        print(bar)
        if a == 2:
            print(last_line)
        else:
            print(middle_line)

#crea el nodo para el puzzle

class Node:
```

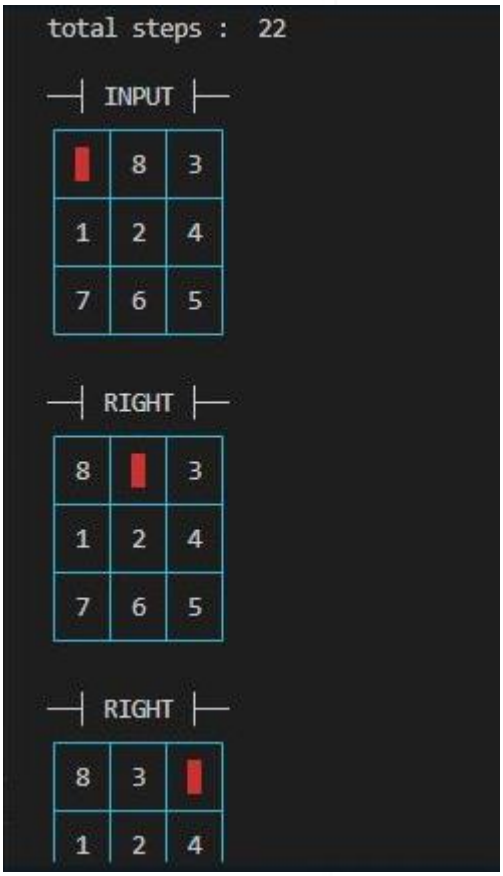
```
    def __init__(self, current_node, previous_node, g, h, dir):
        self.current_node = current_node
        self.previous_node = previous_node
        self.g = g
        self.h = h
        self.dir = dir

    def f(self):
        return self.g + self.h

#esta funcion obtiene la posicion actual del elemento revisado y regresa la fila

def get_pos(current_state, element):
    for row in range(len(current_state)):
        if element in current_state[row]:
            return row, current_state[row].index(element)

#obtiene el costo total que se mueve desde el nodo raiz hasta el nodo meta

def total_cost(current_state):
    cost = 0
    for row in range(len(current_state)):
        for col in range(len(current_state[0])):
            pos = get_pos(END, current_state[row][col])
            cost += abs(row - pos[0]) + abs(col - pos[1])
    return cost
#la funcion que mueve el espacio vacio dentro del puzzle y comienza a hacer los
movimientos con las direcciones declaradas
#y agrega un estado completamente nuevo en la lista de los estados cambiantes
# deepcopy lo hace recursivo, crea un nuevo contenedor para que todo pueda
reorganizarse con el espacio vacío moviéndose
def expand(node):
    listNode = []
    emptyPos = get_pos(node.current_node, 0)

    for dir in DIRECTIONS.keys():
        newPos = (emptyPos[0] + DIRECTIONS[dir][0], emptyPos[1] +
DIRECTIONS[dir][1])
        if 0 <= newPos[0] < len(node.current_node) and 0 <= newPos[1] <
len(node.current_node[0]):
            newState = deepcopy(node.current_node)
            newState[emptyPos[0]][emptyPos[1]] =
node.current_node[newPos[0]][newPos[1]]
            newState[newPos[0]][newPos[1]] = 0
```

```
            # listNode += [Node(newState, node.current_node, node.g + 1,
total_cost(newState), dir)]
            listNode.append(Node(newState, node.current_node, node.g + 1,
total_cost(newState), dir))

    return listNode

#esto considera cual es el mejor nodo para mover el espacio vacio y hacer las
iteraciones

def getBestNode(openSet):
    #global mejor F
    firstIter = True

    for node in openSet.values():
        if firstIter or node.f() < bestF:
            firstIter = False
            bestNode = node
            bestF = bestNode.f()
    return bestNode

#esto crea nuevas maneras de llegar al nodo meta regresando nuevas listas

def buildPath(closedSet):
    node = closedSet[str(END)]
    branch = list()

    while node.dir:
        branch.append({
            'dir': node.dir,
            'node': node.current_node
        })
        node = closedSet[str(node.previous_node)]
    branch.append({
        'dir': '',
        'node': node.current_node
    })
    branch.reverse()

    return branch

#llama a las funciones, pasa los parametros y setea la nueva lista en un array de
valores
#compara la nueva lista que ha sido creada con los movimientos del espacio vacio y
el final es el nodo meta
```

```
#compara en cual hay menos movimientos para obtener la mejor opcion

def main(puzzle):
    open_set = {str(puzzle): Node(puzzle, puzzle, 0, total_cost(puzzle), "")}
    closed_set = {}

    while True:
        test_node = getBestNode(open_set)
        closed_set[str(test_node.current_node)] = test_node

        if test_node.current_node == END:
            return buildPath(closed_set)

        adj_node = expand(test_node)
        for node in adj_node:
            if str(node.current_node) in closed_set.keys() or
str(node.current_node) in open_set.keys() and open_set[
                str(node.current_node)].f() < node.f():
                continue
            open_set[str(node.current_node)] = node

        del open_set[str(test_node.current_node)]


if __name__ == '__main__':      #obtiene el input para el puzzle
    inp = main([[0, 8, 3],
                [1, 2, 4],
                [7, 6, 5]])

    print('total steps : ', len(inp) -   1)    #imprime el total de pasos que hizo
el algoritmo para completar el puzzle
    print()
    print(dash + dash + right_junction, "INPUT", left_junction + dash + dash)
    for i in inp:
        if i['dir'] != '':
            letter = ''
            if i['dir'] == 'U':
                letter = 'UP'
            elif i['dir'] == 'R':
                letter = "RIGHT"
            elif i['dir'] == 'L':
                letter = 'LEFT'
            elif i['dir'] == 'D':
                letter = 'DOWN'
```

```
            print(dash + dash + right_junction, letter, left_junction + dash +
dash)
        print_puzzle(i['node'])
        print()

    print(dash + dash + right_junction, 'ABOVE IS THE OUTPUT', left_junction +
dash + dash)
```

resultados:

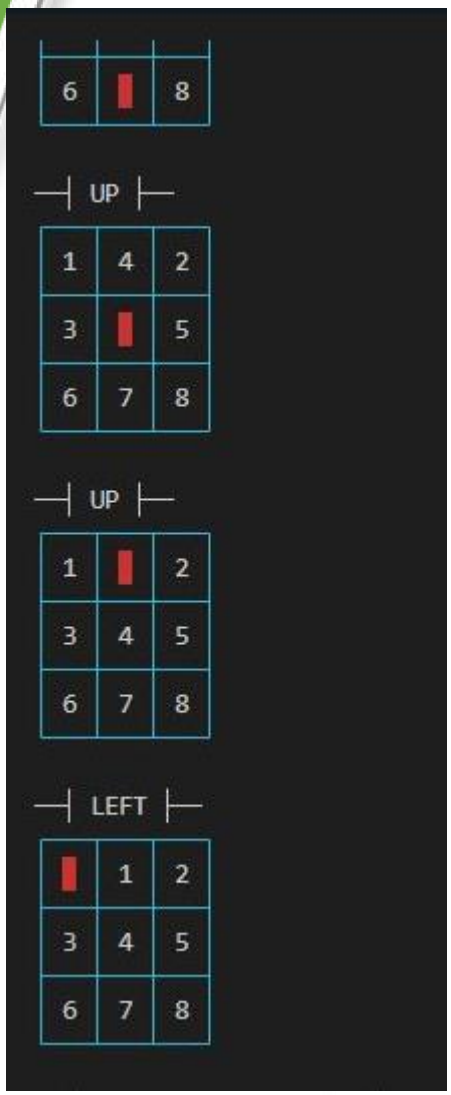



7 6 5
DOWN
8 3 4
1 2
7 6 5
LEFT
8 3 4
1 2
7 6 5
LEFT
8 3 4
1 2
7 6 5
UP

…

…

…



2.-Considere el problema de encontrar el camino más corto entre dos puntos en un plano de geométricas distintas (Fig. 2). El punto origen es un círculo, mientras que el destino es una estrella, ambos son de color rojo. En este caso el espacio de estados corresponde al conjunto de posiciones (x, y) presentes en el plano.

Se deben implementar ambos algoritmos de búsqueda no informada que permitan encontrar de forma automática la mejor trayectoria entre ambos puntos.

El plano puede ser distinto al de la Fig. 2. sin embargo, debe representar complejidad para su solución.

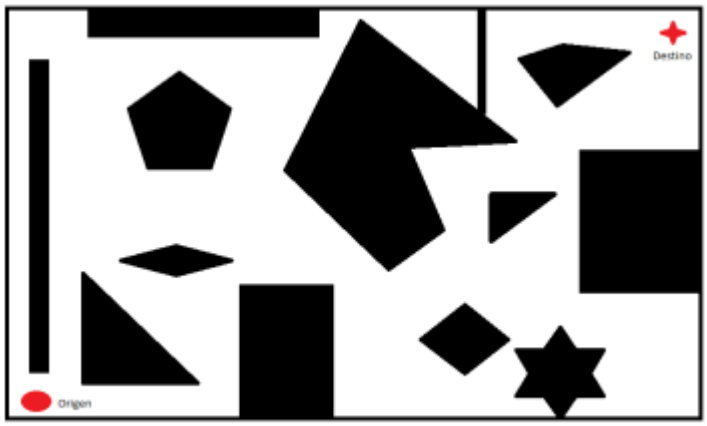


Fig.3 Plano con obstáculos poligonales.

```
def heuristic_search(algorithm, sort_by):

    # Variables
    goal_state = None
    solution_cost = 0
    solution = []

    # Limpiamos frontier y visited, y agregamos root a frontier.
    frontier.clear()
    visited.clear()
    frontier.append(graph.root)

    while len(frontier) > 0:

        # Ordemos frontier de acuerdo a la heuristica
        sort_frontier(sort_by)

        # removemos el nodo coorecto de frontier y lo agregamos a los visitados.
        current_node = frontier.pop(0)
        visited[current_node] = None

        # Para GBFS, si estamos en la meta
        if is_goal(current_node):
            goal_state = current_node
            break

        # print(current_node, current_node.parent)

        # Agregamos a la frontera como BFS.
```

```
        add_to_frontier(current_node, "BFS")

    #Revisa si GBFS fue exitoso
    if goal_state is not None:

        # Calculamos el costo de la solucion y la solucion
        current = goal_state
        while current is not None:
            solution_cost += current.cost
            solution.insert(0, current)
            # Get the parent node and continue...
            #optenemos el nodo padre y continuamos
            current = current.parent

        # Print the results...
        print_results(algorithm, solution_cost, solution, visited)
    else:
        print("No goal state found.")

def return_cost_and_heuristic(node):
    return node.heuristic + node.cost
```

Resultado:

```
A Star Search(A*):
Cost of the solution: 18
The solution path (19 nodes): [2, 1] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [0, 2] [0, 3] [0, 4] [0, 5] [0, 6] [0,
 7] [1, 7] [2, 7] [3, 7] [4, 7] [5, 7] [5, 6]
Expanded nodes (48 nodes): [2, 1] [3, 1] [3, 2] [4, 2] [5, 2] [5, 3] [4, 3] [3, 3] [3, 4] [4, 4] [5, 4] [6, 4] [5, 1]
 [4, 1] [5, 0] [6, 0] [6, 1] [6, 2] [6, 3] [7, 3] [7, 2] [7, 0] [7, 1] [2, 2] [4, 0] [3, 0] [2, 0] [1, 0] [0, 0] [0,
1] [1, 1] [1, 2] [0, 2] [0, 3] [0, 4] [0, 5] [0, 6] [1, 4] [0, 7] [1, 7] [2, 7] [1, 6] [3, 7] [3, 6] [1, 5] [4, 7] [5
, 7] [5, 6]
```

Conclusión:

Este algoritmo se me hizo muy complicado de entender ya que es un desastre total, pero ya después de batallar mucho tiempo entendí como es que este funcionaba y es dar la mejor solución considerando g(n) y h(n) que la combinación de estos no s puede ayudar a encontrar la mejor ruta tanto en distancia como las mas optima, imagino que este algoritmo es el que utilizan las aplicaciones que te indican como y por donde llegar considerando el trafico o detalles, como ellos, pienso que este algoritmo lo encontrare mucho en mi futuro como ingeniero.