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

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.

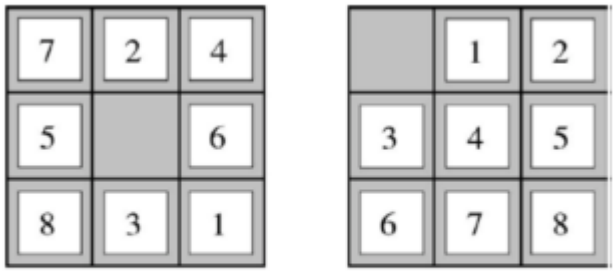

Fig.1 Ejemplo del problema 8 Puzzle

```
import copy
import time

def printNode(node):
    print(node[0],node[1],node[2])
    print(node[3],node[4],node[5])
    print(node[6],node[7],node[8])
    global nodeNumber
```

```
    print('Node:', nodeNumber)
    print('Depth:', len(node[9:]))
    print('Moves:', node[9:])
    print('------')
    nodeNumber += 1

def checkFinal(node):
    if node[:9]==finalNode:
        printNode(node)
        return True
    if node[:9] not in visitedList:
        printNode(node)
        nodeList.append(node)
        visitedList.append(node[:9])
    return False

def calculateHeuristic(node):
    distance = 0
    for current, target in enumerate(node):
        currentRow = int(current/3)
        currentColumn = current%3
        targetRow = int(target/3)
        targetColumn = target%3
        distance += abs(currentRow-targetRow) + abs(currentColumn-targetColumn)
    return distance

if __name__ == '__main__':
    startNode = [1,5,4, 3,7,2, 6,8,0]
    finalNode = [0,1,2, 3,4,5, 6,7,8]

    found = False
    nodeNumber = 0
    visitedList = []
    nodeList = []
    nodeList.append(startNode)
    visitedList.append(startNode)
    printNode(startNode)
    t0 = time.time()

    while (not found and not len(nodeList)==0):
        fList = []
        for node in nodeList:
            h = calculateHeuristic(node[:9])
            g = len(node[9:])
            f = g+h
```

```
            fList.append(f)
        currentNode = nodeList.pop(fList.index(min(fList)))
        blankIndex = currentNode.index(0)
        if blankIndex!=0 and blankIndex!=1 and blankIndex!=2:
            upNode = copy.deepcopy(currentNode)
            upNode[blankIndex] = upNode[blankIndex-3]
            upNode[blankIndex-3] = 0
            upNode.append('up')
            found = checkFinal(upNode)
        if blankIndex!=0 and blankIndex!=3 and blankIndex!=6 and found==False:
            leftNode = copy.deepcopy(currentNode)
            leftNode[blankIndex] = leftNode[blankIndex-1]
            leftNode[blankIndex-1] = 0
            leftNode.append('left')
            found = checkFinal(leftNode)
        if blankIndex!=6 and blankIndex!=7 and blankIndex!=8 and found==False:
            downNode = copy.deepcopy(currentNode)
            downNode[blankIndex] = downNode[blankIndex+3]
            downNode[blankIndex+3] = 0
            downNode.append('down')
            found = checkFinal(downNode)
        if blankIndex!=2 and blankIndex!=5 and blankIndex!=8 and found==False:
            rightNode = copy.deepcopy(currentNode)
            rightNode[blankIndex] = rightNode[blankIndex+1]
            rightNode[blankIndex+1] = 0
            rightNode.append('right')
            found = checkFinal(rightNode)

    t1 = time.time()
    print('Time:', t1-t0)
    print('------')
```

Resultado:

```
1 5 4
3 7 2
6 8 0
Node: 0
Depth: 0
Moves: []
------
1 5 4
3 7 0
6 8 2
Node: 1
Depth: 1
Moves: ['up']
------
1 5 4
3 7 2
6 0 8
Node: 2
Depth: 1
Moves: ['left']
------
1 5 4
3 0 2
```

…

…

```
Node: 4
Depth: 2
Moves: ['left', 'left']
------
1 0 4
3 5 2
6 7 8
Node: 5
Depth: 3
Moves: ['left', 'up', 'up']
------
1 5 4
0 3 2
6 7 8
Node: 6
Depth: 3
Moves: ['left', 'up', 'left']
------
1 5 4
3 2 0
6 7 8
Node: 7
```

…

…

```
------
1 4 2
3 7 5
6 0 8
Node: 20
Depth: 7
Moves: ['left', 'up', 'up', 'right', 'down', 'left', 'down']
------
0 1 2
3 4 5
6 7 8
Node: 21
Depth: 8
Moves: ['left', 'up', 'up', 'right', 'down', 'left', 'up', 'left']
```

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.

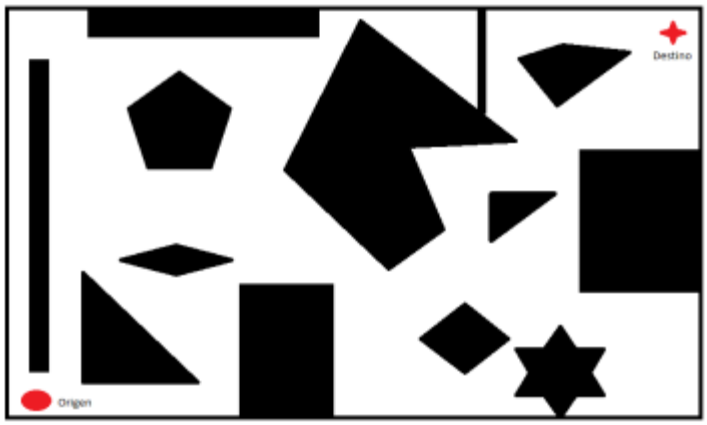

Fig.3 Plano con obstáculos poligonales.

```
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.")
```

Resultado:

```
Breath First Search(BFS):
Cost of the solution: 23
The solution path (12 nodes): [2, 1] [2, 0] [1, 0] [0, 0] [0, 1] [1, 1] [1, 2] [1, 3] [2, 3] [2, 4] [
2, 5] [2, 6]
Expanded nodes (42 nodes): [2, 1] [3, 1] [2, 0] [3, 2] [4, 1] [3, 0] [1, 0] [4, 2] [2, 2] [4, 0] [0,
0] [5, 2] [5, 0] [0, 1] [5, 3] [5, 1] [6, 0] [1, 1] [4, 3] [6, 1] [7, 0] [1, 2] [3, 3] [6, 2] [7, 1]
[1, 3] [0, 2] [3, 4] [6, 3] [7, 2] [2, 3] [0, 3] [4, 4] [7, 3] [2, 4] [0, 4] [5, 4] [2, 5] [1, 4] [0,
 5] [6, 4] [2, 6]
```

Conclusión:

Este algoritmo me gusta mucho porque siempre al generar los nodos siguientes, busca el que tenga una relación mas cercana a la solución o el nodo que se vea más prometedor, pudiendo llegar a la solución final de una manera más rápida y metafóricamente pudiendo llamarla como de una manera un poco mas lógica, ya que es la manera en que usualmente nosotros resolvemos a veces los problemas, tomando el camino mas parecido o el que nos cueste menos trabajo para llegar a una solución.