ARTIFICIAL INTELLIGENCE

LAB 1

Sahil Satyam
RA1911003010675.

AIM: Implementation of 8-puzzle problem.

PROBLEM FORMULATION:

8-puzzle: It is a 3x3 matrix with 8 square blocks from 1 to 8 and a blank space. The idea is to reorder these squares into numerical order of 1 to 8 and the last square as blank.

Each of the squares adjacent to the blank block can move up, down, left, right depending on the edges of the matrix.

eg:

initial state

	1	3
4	2	5
7	8	6

→

1		3
4	2	5
7	8	6

→

1	2	3
4	5	
7	8	6

→

Goal State.

1	2	3
4	5	6
7	8	

PROBLEM SOLVING:

A* is a recursive algorithm that calls itself until a sol^n is found. In this algorithm we consider two heuristic functions, misplaced tile heuristic and manhattan distance heuristic. The misplaced tiles heuristic calculates the misplaced number of tiles of the current state as compared to the goal state. Manhattan distance heuristic function measures the least steps needed for each of the tiles in the 8-puzzle initial or current state to arrive at the goal state position. We have implemented the state space generation using both heuristics.

we are also calculating $g(n)$ which is a measure of step cost for each move made from the current state to the next state, initially it is set to 0. For each of the heuristic we have implemented $f(n) = g(n) + h(n)$ where $g(n)$ is step cost and $h(n)$ is the heuristic function used. Each of the states is explored using a priority queue is sorted and the next node to be explored is selected based on the least $f(n)$ value.

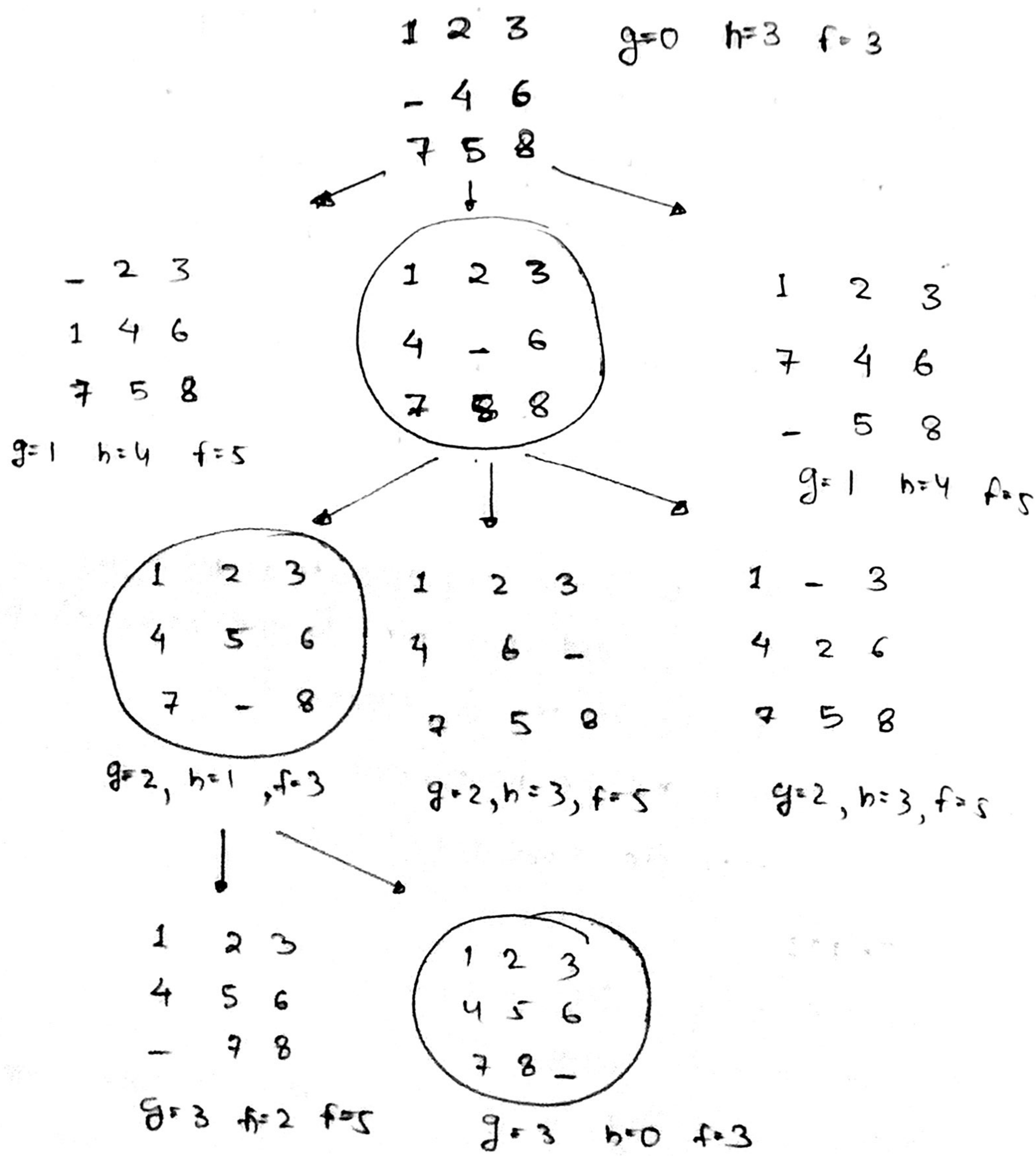
ALGORITHM:

- use two lists 'open list' and 'closed list'.
- open list contains all the nodes that are being generated and are not existing in the closed list & each node explored after this neighbouring nodes are discovered is put in the closed list.
- the next node chosen from the open list is based on it's f-score the node with the least f-score is picked up & explored.

f-score = h score + g score

h score : how far the goal node is

g score : number of nodes traversed from start to current.



- after expanding the current state, it is pushed into the closed list and the newly generated states are pushed into the open list.
- A state with the least f-score is selected and expanded again.
- This process continues until the goal state occurs as the current state.
-

Program :
```
from copy import deepcopy
import numpy as np
import time

# takes the input of current states and evaluvates the best path to goal state
def bestsolution(state):
    bestsol = np.array([], int).reshape(-1, 9)
    count = len(state) - 1
    while count != -1:
        bestsol = np.insert(bestsol, 0, state[count]['puzzle'], 0)
        count = (state[count]['parent'])
    return bestsol.reshape(-1, 3, 3)


# this function checks for the uniqueness of the iteration(it) state, weather it has been previously traversed or not.
def all(checkarray):
    set=[]
    for it in set:
        for checkarray in it:
            return 1
        else:
            return 0


# calculate Manhattan distance cost between each digit of puzzle(start state) and the goal state
def manhattan(puzzle, goal):
    a = abs(puzzle // 3 - goal // 3)
    b = abs(puzzle % 3 - goal % 3)
    mhcost = a + b
    return sum(mhcost[1:])



# will calcuates the number of misplaced tiles in the current state as compared to the goal state
def misplaced_tiles(puzzle,goal):
    mscost = np.sum(puzzle != goal) - 1
    return mscost if mscost > 0 else 0



#3[on_true] if [expression] else [on_false]
```

```
# will indentify the coordinates of each of goal or initial state values
def coordinates(puzzle):
    pos = np.array(range(9))
    for p, q in enumerate(puzzle):
        pos[q] = p
    return pos



# start of 8 puzzle evaluvation, using Manhattan heuristics
def evaluvate(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],
                dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

     # initializing the parent, gn and hn, where hn is manhattan distance function call
    costg = coordinates(goal)
    parent = -1
    gn = 0
    hn = manhattan(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

# We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]
    priority = np.array( [(0, hn)], dtpriority)



    while 1:
        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
        position, fn = priority[0]
        priority = np.delete(priority, 0, 0)
        # sort priority queue using merge sort,the first element is picked for exploring remove from
queue what we are exploring
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
        # Identify the blank square in input
        blank = int(np.where(puzzle == 0)[0])
        gn = gn + 1
        c = 1
        start_time = time.time()
```

```
    for s in steps:
        c = c + 1
        if blank not in s['position']:
            # generate new state as copy of current
            openstates = deepcopy(puzzle)
            openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],
openstates[blank]
            # The all function is called, if the node has been previously explored or not
            if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                end_time = time.time()
                if (( end_time - start_time ) > 2):
                    print(" The 8 puzzle is unsolvable ! \n")
                    exit
                # calls the manhattan function to calcuate the cost
                hn = manhattan(coordinates(openstates), costg)
                 # generate and add new state in the list
                q = np.array([(openstates, position, gn, hn)], dtstate)
                state = np.append(state, q, 0)
                # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the
goal state
                fn = gn + hn

                q = np.array([(len(state) - 1, fn)], dtpriority)
                priority = np.append(priority, q, 0)
                  # Checking if the node in openstates are matching the goal state.
                if np.array_equal(openstates, goal):
                    print(' The 8 puzzle is solvable ! \n')
                    return state, len(priority)


    return state, len(priority)



# start of 8 puzzle evaluvation, using Misplaced tiles heuristics
def evaluvate_misplaced(puzzle, goal):
    steps = np.array([('up', [0, 1, 2], -3),('down', [6, 7, 8],  3),('left', [0, 3, 6], -1),('right', [2, 5, 8],  1)],
            dtype =  [('move',  str, 1),('position', list),('head', int)])

    dtstate = [('puzzle',  list),('parent', int),('gn',  int),('hn',  int)]

    costg = coordinates(goal)
    # initializing the parent, gn and hn, where hn is misplaced_tiles  function call
    parent = -1
    gn = 0
```

```
    hn = misplaced_tiles(coordinates(puzzle), costg)
    state = np.array([(puzzle, parent, gn, hn)], dtstate)

   # We make use of priority queues with position as keys and fn as value.
    dtpriority = [('position', int),('fn', int)]

    priority = np.array([(0, hn)], dtpriority)

    while 1:
        priority = np.sort(priority, kind='mergesort', order=['fn', 'position'])
        position, fn = priority[0]
        # sort priority queue using merge sort,the first element is picked for exploring.
        priority = np.delete(priority, 0, 0)
        puzzle, parent, gn, hn = state[position]
        puzzle = np.array(puzzle)
         # Identify the blank square in input
        blank = int(np.where(puzzle == 0)[0])
        # Increase cost g(n) by 1
        gn = gn + 1
        c = 1
        start_time = time.time()
        for s in steps:
            c = c + 1
            if blank not in s['position']:
                 # generate new state as copy of current
                openstates = deepcopy(puzzle)
                openstates[blank], openstates[blank + s['head']] = openstates[blank + s['head']],
openstates[blank]
                # The check function is called, if the node has been previously explored or not.
                if ~(np.all(list(state['puzzle']) == openstates, 1)).any():
                    end_time = time.time()
                    if (( end_time - start_time ) > 2):
                        print(" The 8 puzzle is unsolvable \n")
                        break
                    # calls the Misplaced_tiles function to calcuate the cost
                    hn = misplaced_tiles(coordinates(openstates), costg)
                    # generate and add new state in the list
                    q = np.array([(openstates, position, gn, hn)], dtstate)
                    state = np.append(state, q, 0)
                    # f(n) is the sum of cost to reach node and the cost to rech fromt he node to the
goal state
                    fn = gn + hn

                    q = np.array([(len(state) - 1, fn)], dtpriority)
```

```
                priority = np.append(priority, q, 0)
                # Checking if the node in openstates are matching the goal state.
                if np.array_equal(openstates, goal):
                    print(' The 8 puzzle is solvable \n')
                    return state, len(priority)

    return state, len(priority)
puzzle = []
print(" Input vals from 0-8 for start state ")
for i in range(0,9):
    x = int(input("enter vals :"))
    puzzle.append(x)

goal = []
print(" Input vals from 0-8 for goal state ")
for i in range(0,9):
    x = int(input("Enter vals :"))
    goal.append(x)



n = int(input("1. Manhattan distance \n2. Misplaced tiles\n"))

if(n ==1 ):
    state, visited = evaluvate(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))

if(n == 2):
    state, visited = evaluvate_misplaced(puzzle, goal)
    bestpath = bestsolution(state)
    print(str(bestpath).replace('[', ' ').replace(']', ''))
    totalmoves = len(bestpath) - 1
    print('Steps to reach goal:',totalmoves)
    visit = len(state) - visited
    print('Total nodes visited: ',visit, "\n")
    print('Total generated:', len(state))
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

(base) PS C:\Users\Admin\Desktop\Assignments\AI> python -u "c:\Users\Admin\Desktop\Assignments\AI\8puz.py"
Enter the start state matrix

1 2 3
_ 4 6
7 5 8
Enter the goal state matrix

1 2 3
4 5 6
7 8 _



  |
  |
 \'/

1 2 3
4 5 6
7 _ 8

  |
  |
 \'/

1 2 3
4 5 6
7 8 _
```