Assignment-3
Arsh kantiwal
102016057
2CS10

Q1. If the initial and final states are as below, find the value of the Heuristic function, by taking
(i) Euclidean Distance
(ii) Manhattan Distance
(iii) Minkowski Distance

A1.

```
import  sys

import  copy

import  math

import numpy as

np



def find_pos(s, elem):

    for i in range(len(s)):

        for j in range(len(s[0])):

            if s[i][j] == elem:

                return [i, j]
```

```
def eucledian(s, g):
    res_mat = np.zeros(len(s) * len(s[0]), dtype=float)
    res_mat = res_mat.reshape(len(s), len(s))

    for x1 in range(len(s)):
        for y1 in range(len(s[0])):
            elem = s[x1][y1]
            pos = find_pos(g, elem)

            x2 = pos[0]
            y2 = pos[1]

            res_mat[x1][y1] = math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

    summ = 0

    for i in range(len(res_mat)):
        summ += sum(res_mat[i])
    return summ


def manhattan(s, g):
```

```
    res_mat = np.zeros(len(s) * len(s[0]), dtype=float)

    res_mat = res_mat.reshape(len(s), len(s))



    for x1 in range(len(s)):

        for y1 in range(len(s[0])):

            elem = s[x1][y1]

            pos = find_pos(g, elem)



            x2 = pos[0]

            y2 = pos[1]



            res_mat[x1][y1] = abs(x2 - x1) + abs(y2 - y1)



    summ = 0



    for i in range(len(res_mat)):

        summ += sum(res_mat[i])

    return summ





def minkowiski(s, g, p):

    res_mat = np.zeros(len(s) * len(s[0]), dtype=float)

    res_mat = res_mat.reshape(len(s), len(s))
```

```
    for x1 in range(len(s)):
        for y1 in range(len(s[0])):
            elem = s[x1][y1]
            pos = find_pos(g, elem)

            x2 = pos[0]
            y2 = pos[1]

            res_mat[x1][y1] = ((abs(x2 - x1) ** p) + (abs(y2 - y1) ** p)) **
(1. / p)

    summ = 0

    for i in range(len(res_mat)):
        summ += sum(res_mat[i])
    return summ


def main():
    p_val = 3
    s0 = [[2, 0, 3], [1, 8, 4], [7, 6, 5]]
    g = [[1, 2, 3], [8, 4, 0], [7, 6, 5]]
    euc = eucledian(s0, g)
    man = manhattan(s0, g)
    mink = minkowiski(s0, g, p_val)
```

```
    print(euc, man, mink)



if __name__ == "__main__":
    main()
```

```
C:\Python\python.exe "D:/College Work/AI/Labs/Pycharm/lab3/q1.py"
5.414213562373095 6.0 5.259921049894873

Process finished with exit code 0
```

Q2. If the initial and final states are as below and H(n): number of misplaced tiles in the current state n as compared to the goal node need to be considered as the heuristic function. You need to use best first Search algorithm.

A2.

```
import sys
import copy


q = []
visited = []



def compare(s, g):
```

```
    if s == g:
        return (1)
    else:
        return (0)


def find_pos(s):
    for i in range(3):
        for j in range(3):
            if s[i][j] == 0:
                return ([i, j])


def up(s, pos):
    i = pos[0]
    j = pos[1]

    if i > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i - 1][j]
        temp[i - 1][j] = 0
        return (temp)
    else:
        return (s)
```

```python
def down(s, pos):
    i = pos[0]
    j = pos[1]

    if i < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i + 1][j]
        temp[i + 1][j] = 0
        return (temp)
    else:
        return (s)


def right(s, pos):
    i = pos[0]
    j = pos[1]

    if j < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j + 1]
        temp[i][j + 1] = 0
        return (temp)
```

```
    else:
        return (s)


def left(s, pos):
    i = pos[0]
    j = pos[1]

    if j > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j - 1]
        temp[i][j - 1] = 0
        return (temp)
    else:
        return (s)


def enqueue(s, val):
    global q
    q = q + [(val, s)]


def heuristic(s, g):
    d = 0
```

```
    for i in range(3):
        for j in range(3):
            if s[i][j] != g[i][j]:
                d += 1
    print(d)
    return d



def dequeue(g):
    global q
    global visited

    q.sort()
    visited = visited + [q[0][1]]

    elem = q[0][1]
    del q[0]
    return (elem)



def search(s, g):
    curr_state = copy.deepcopy(s)
    if s == g:
        return
```

```
    global visited
    while (1):

        pos = find_pos(curr_state)
        new = up(curr_state, pos)

        if new != curr_state:
            if new == g:
                print("found!! The intermediate states are:")
                print(visited + [g])
                return
            else:
                if new not in visited:
                    enqueue(new, heuristic(new, g))

        new = down(curr_state, pos)

        if new != curr_state:
            if new == g:
                print("found!! The intermediate states are:")
                print(visited + [g])
                return
            else:
```

```
                if new not in visited:
                    enqueue(new, heuristic(new, g))


        new = right(curr_state, pos)


        if new != curr_state:
            if new == g:
                print("found!! The intermediate states are:")
                print(visited + [g])
                return
            else:
                if new not in visited:
                    enqueue(new, heuristic(new, g))


        new = left(curr_state, pos)


        if new != curr_state:
            if new == g:
                print("found!! The intermediate states are:")
                print(visited + [g])
                return
            else:
                if new not in visited:
                    enqueue(new, heuristic(new, g))
```

```
        if len(q) > 0:
            curr_state = dequeue(g)
        else:
            print("not found")
            return



def main():
    s = [[2, 0, 3], [1, 8, 4], [7, 6, 5]]
    g = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
    global q
    global visited
    q = q
    visited = visited + [s]

    search(s, g)



if __name__ == "__main__":
    main()
```

```
C:\Python\python.exe "D:/College Work/AI/Labs/Pycharm/lab3/q2.py"
3
5
3
2
3
found!! The intermediate states are:
[[[2, 0, 3], [1, 8, 4], [7, 6, 5]], [[0, 2, 3], [1, 8, 4], [7, 6, 5]], [[1, 2, 3], [0, 8, 4], [7, 6, 5]], [[1, 2, 3], [8, 0, 4], [7, 6, 5]]]

Process finished with exit code 0
```

Q3.If the initial and final states are as below and H(n): number of misplaced tiles in the current state n as compared to the goal node need to be considered as the heuristic function. You need to use Hill Climbing algorithm.

A3.

```
import sys
import copy

curr_min = sys.maxsize
q = []
visited = []


def compare(s, g):
    if s == g:
        return (1)
    else:
        return (0)
```

```python
def find_pos(s):
  for i in range(len(s)):
      for j in range(len(s[0])):
          if s[i][j] == 0:
              return ([i, j])



def up(s, pos):
  i = pos[0]
  j = pos[1]

  if i > 0:
      temp = copy.deepcopy(s)
      temp[i][j] = temp[i - 1][j]
      temp[i - 1][j] = 0
      return (temp)
  else:
      return (s)



def down(s, pos):
  i = pos[0]
```

```
    j = pos[1]


    if i < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i + 1][j]
        temp[i + 1][j] = 0
        return (temp)
    else:
        return (s)



def right(s, pos):
    i = pos[0]
    j = pos[1]


    if j < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j + 1]
        temp[i][j + 1] = 0
        return (temp)
    else:
        return (s)
```

```python
def left(s, pos):
    i = pos[0]
    j = pos[1]

    if j > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j - 1]
        temp[i][j - 1] = 0
        return (temp)
    else:
        return (s)



def enqueue(s):
    global q
    q = q + [s]



def heuristic(s, g):
    d = 0
    for i in range(len(s)):
        for j in range(len(s[0])):
            if s[i][j] != g[i][j]:
                d += 1
```

```
    return d



def dequeue(g):

    h = []

    global q

    global visited

    global curr_min


    for i in range(len(q)):

        h = h + [heuristic(q[i], g)]


    if min(h) < curr_min:

        curr_min = min(h)

        index = h.index(min(h))

        visited = visited + [q[index]]

    else:

        print("optimal solution found !! The intermediate states are: ")

        print(visited)

        exit()


    elem = q[index]

    q = []

    return (elem)
```

```python
def search(s, g):
    curr_state = copy.deepcopy(s)
    if s == g:
        return

    global visited
    while (1):

        pos = find_pos(curr_state)
        new = up(curr_state, pos)

        if new != curr_state:
            if new == g:
                print("Goal State found !! The intermediate States are :")
                print(visited + [g])
                return
            else:
                if new not in visited:
                    enqueue(new)

        new = down(curr_state, pos)
```

```python
        if new != curr_state:

            if new == g:

                print("Goal State found !! The intermediate States are :")

                print(visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new)


        new = right(curr_state, pos)


        if new != curr_state:

            if new == g:

                print("Goal State found !! The intermediate States are :")

                print(visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new)


        new = left(curr_state, pos)


        if new != curr_state:

            if new == g:
```

```
                print("Goal State found !! The intermediate States are :")
                print(visited + [g])
                return
            else:
                if new not in visited:
                    enqueue(new)

        if len(q) > 0:
            curr_state = dequeue(g)
        else:
            print("not found")
            return


def main():
    s = [[2, 8, 3], [1, 5, 4], [7, 6, 0]]
    g = [[1, 2, 7], [8, 0, 5], [3, 4, 6]]
    global q
    global visited
    q = q + [s]
    visited = visited + [s]

    search(s, g)
```

```
if __name__ == "__main__":
    main()
```

```
C:\Python\python.exe "D:/College Work/AI/Labs/Pycharm/lab3/q3.py"
optimal solution found !! The intermediate states are:
[[[2, 8, 3], [1, 5, 4], [7, 6, 0]], [[2, 8, 3], [1, 5, 4], [7, 0, 6]], [[2, 8, 3], [1, 0, 4], [7, 5, 6]]]

Process finished with exit code 0
```

Q4. If the initial and final states are as below and H(n): Manhattan distance as the heuristic function. You need to use Best First Search algorithm.

A4.

```
import copy
from heuristic import *


q = []
visited = []




def compare(s, g):
    if s == g:
        return (1)
    else:
        return (0)
```

```python
def find_pos(s):
    for i in range(3):
        for j in range(3):
            if s[i][j] == 0:
                return ([i, j])


def up(s, pos):
    i = pos[0]
    j = pos[1]

    if i > 0:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i - 1][j]
        temp[i - 1][j] = 0
        return (temp)
    else:
        return (s)


def down(s, pos):
    i = pos[0]
```

```
    j = pos[1]


    if i < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i + 1][j]
        temp[i + 1][j] = 0
        return (temp)
    else:
        return (s)



def right(s, pos):
    i = pos[0]
    j = pos[1]


    if j < 2:
        temp = copy.deepcopy(s)
        temp[i][j] = temp[i][j + 1]
        temp[i][j + 1] = 0
        return (temp)
    else:
        return (s)
```

```
def left(s, pos):

    i = pos[0]

    j = pos[1]


    if j > 0:

        temp = copy.deepcopy(s)

        temp[i][j] = temp[i][j - 1]

        temp[i][j - 1] = 0

        return (temp)

    else:

        return (s)




def enqueue(s, val):

    global q

    q = q + [(val, s)]




def heuristic(s, g):

    d = (s, g)

    print

    d

    # d = 0

    # for i in range(3):
```

```
    #     for j in range(3):
    #         if s[i][j] != g[i][j]:
    #             d += 1
    return d


def dequeue(g):
    global q
    global visited

    q.sort()
    visited = visited + [q[0][1]]

    elem = q[0][1]
    del q[0]
    return (elem)


def search(s, g):
    curr_state = copy.deepcopy(s)
    if s == g:
        return

    global visited
```

```
    while (1):


        pos = find_pos(curr_state)

        new = up(curr_state, pos)


        if new != curr_state:

            if new == g:

                print("found!! The intermediate states are:")

                print(visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new, heuristic(new, g))


        new = down(curr_state, pos)


        if new != curr_state:

            if new == g:

                print("found!! The intermediate states are:")

                print(visited + [g])

                return

            else:

                if new not in visited:

                    enqueue(new, heuristic(new, g))
```

```
    new = right(curr_state, pos)

    if new != curr_state:
        if new == g:
            print("found!! The intermediate states are:")
            print(visited + [g])
            return
        else:
            if new not in visited:
                enqueue(new, heuristic(new, g))

    new = left(curr_state, pos)

    if new != curr_state:
        if new == g:
            print("found!! The intermediate states are:")
            print(visited + [g])
            return
        else:
            if new not in visited:
                enqueue(new, heuristic(new, g))

    if len(q) > 0:
```

```python
            curr_state = dequeue(g)
        else:
            print("not found")
            return


def main():
    s = [[2, 0, 3], [1, 8, 4], [7, 6, 5]]
    g = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
    global q
    global visited
    q = q
    visited = visited + [s]

    search(s, g)


if __name__ == "__main__":
    main()
```

```
C:\Python\python.exe "D:/College Work/AI/Labs/Pycharm/lab3/q5.py"
found!! The intermediate states are:
[[[2, 0, 3], [1, 8, 4], [7, 6, 5]], [[0, 2, 3], [1, 8, 4], [7, 6, 5]], [[1, 2, 3], [0, 8, 4], [7, 6, 5]], [[1, 2, 3], [8, 0, 4], [7, 6, 5]]]

Process finished with exit code 0
```

Q5. Solve this given problem using Uniform Cost search. A is the initial state and G is the goal state

A5.

```
def uniform_cost_search(goal, start):
    # minimum cost upto
    # goal state from starting
    global graph, cost
    answer = []

    # create a priority queue
    queue = []

    # set the answer vector to max value
    for i in range(len(goal)):
        answer.append(10 ** 8)

    # insert the starting index
    queue.append([0, start])

    # map to store visited node
```

```
    visited = {}


    # count
    count = 0


    # while the queue is not empty
    while (len(queue) > 0):


        # get the top element of the
        queue = sorted(queue)
        p = queue[-1]


        # pop the element
        del queue[-1]


        # get the original value
        p[0] *= -1


        # check if the element is part of
        # the goal list
        if (p[1] in goal):


            # get the position
            index = goal.index(p[1])
```

```
            # if a new goal is reached
            if (answer[index] == 10 ** 8):
                count += 1

            # if the cost is less
            if (answer[index] > p[0]):
                answer[index] = p[0]

            # pop the element
            del queue[-1]

            queue = sorted(queue)
            if (count == len(goal)):
                return answer

        # check for the non visited nodes
        # which are adjacent to present node
        if (p[1] not in visited):
            for i in range(len(graph[p[1]])):
                # value is multiplied by -1 so that
                # least priority is at the top
                queue.append([(p[0] + cost[(p[1], graph[p[1]][i])]) * -1,
graph[p[1]][i]])
```

```
        # mark as visited
        visited[p[1]] = 1

    return answer



# main function
if __name__ == '__main__':
    # create the graph
    graph, cost = [[] for i in range(8)], {}

    # add edge
    graph[0].append(1)
    graph[0].append(3)
    graph[3].append(1)
    graph[3].append(6)
    graph[3].append(4)
    graph[1].append(6)
    graph[4].append(2)
    graph[4].append(5)
    graph[2].append(1)
    graph[5].append(2)
    graph[5].append(6)
    graph[6].append(4)
```

```
# add the cost
cost[(0, 1)] = 2
cost[(0, 3)] = 5
cost[(1, 6)] = 1
cost[(3, 1)] = 5
cost[(3, 6)] = 6
cost[(3, 4)] = 2
cost[(2, 1)] = 4
cost[(4, 2)] = 4
cost[(4, 5)] = 3
cost[(5, 2)] = 6
cost[(5, 6)] = 3
cost[(6, 4)] = 7

# goal state
goal = []

# set the goal
# there can be m
goal.append(6)

# get the answer
answer = uniform
```

```python
# print the answer

print("Minimum cost from 0 to 6 is = ", answer[0])
```

```
C:\Python\python.exe "D:/College Work/AI/Labs/Pycharm/lab3/q5.py"
Minimum cost from 0 to 6 is =  3

Process finished with exit code 0
```