Assignment 4

Name: Krunal Rank
Roll No: U18CO081

Implement 8 Puzzle problems using the algorithms below in PYTHON.
Compare the complexity of both algorithms.
Which algorithm is best suited for implementing 8 Puzzle problems and why ?

1. Breadth First Search
2. Depth First Search
3. Uniform Cost Search
4. Greedy Best First Search

Note: For the 8 puzzle problem, Greedy Best First Search and UCS can only be implemented if the Path cost is known. Instead, I implemented Best First Search as BFS with path cost as 1 in all directions and UCS as path cost equal to Manhattan distance from goal (which may be a part of Informed search). A star algorithm is something I implemented myself for more learning. It comprises BFS + UCS.

Code:

```
import sys
from collections import deque
from copy import deepcopy
from queue import PriorityQueue
import time
from collections import Counter

class Node:
   def __init__(self,state,depth = 0,moves = None,optimizer=0):
       '''
           Parameters:
               state: State of Puzzle
               depth: Depth of State in Space Search Tree
               moves: Moves List to reach this state from initial state
               optimizer: Used for UCS Only
                   0 - Manhattan Distance
                   1 - Hamming Distance
                   2 - Combination of 0 and 1

           Returns: Node Object
       '''
       self.state = state
```

```python
        self.size = len(state)
        self.depth = depth
        self.optimizer = optimizer
        if moves is None:
            self.moves = list()
        else:
            self.moves = moves


    def getAvailableActions(self):
        '''
            Parameters: Current State
            Returns: Available Actions for Current State
            0 - Left    1 - Right   2 - Top     3 - Bottom
            Restrictions: state is self.size x self.size Array
        '''
        action = list()
        for i in range(self.size):
            for j in range(self.size):
                if self.state[i][j]==0:
                    if(i>0):
                        action.append(2)
                    if(j>0):
                        action.append(0)
                    if(i<self.size-1):
                        action.append(3)
                    if(j<self.size-1):
                        action.append(1)
                    return action
        return action

    def getResultFromAction(self,action):
        '''
            Parameters: Current State , Action
            Returns: Node with New State
            Restrictions: Action will always be valid and state is self.size x
self.size Array
        '''
        newstate = deepcopy(self.state)
        newMoves = deepcopy(self.moves)
        for i in range(self.size):
            for j in range(self.size):
```

```
            if(newstate[i][j]==0) :
                if(action==2):
                    newstate[i][j],newstate[i-1][j] =
newstate[i-1][j],newstate[i][j]
                    newMoves.append(2)
                    return Node(newstate,depth = self.depth+1,moves =
newMoves,optimizer=self.optimizer)
                if(action==3):
                    newstate[i][j],newstate[i+1][j] =
newstate[i+1][j],newstate[i][j]
                    newMoves.append(3)
                    return Node(newstate,depth = self.depth+1,moves =
newMoves,optimizer=self.optimizer)
                if(action==0):
                    newstate[i][j],newstate[i][j-1] =
newstate[i][j-1],newstate[i][j]
                    newMoves.append(0)
                    return Node(newstate,depth = self.depth+1,moves =
newMoves,optimizer=self.optimizer)
                if(action==1):
                    newstate[i][j],newstate[i][j+1] =
newstate[i][j+1],newstate[i][j]
                    newMoves.append(1)
                    return Node(newstate,depth = self.depth+1,moves =
newMoves,optimizer=self.optimizer)
        return None

    def isGoalState(self):
        '''
            Parameters: State
            Returns: True if Goal State, otherwise False
            Restrictions: State is self.size x self.size Array
        '''
        for i in range(self.size):
            for j in range(self.size):
                if(i==j and j==self.size-1):
                    continue
                if(self.state[i][j]!=(i)*self.size + (j+1)):
                    return False
        return True

    def getManhattanDistance(self):
```

```
        '''
            Parameters: State
            Returns: Manhattan Distance between Current State and Goal State
            Restrictions: State must be a self.size x self.size Array
        '''
        ans = 0
        for i in range(self.size):
            for j in range(self.size):
                if(self.state[i][j]!=0):
                    ans = ans + abs((self.state[i][j]-1)%self.size - j) +
abs((self.state[i][j]-1)//self.size - i)

        return ans

    def getHammingDistance(self):
        ans = 0
        for i in range(self.size):
            for j in range(self.size):
                if(self.state[i][j]!=0 and self.state[i][j]!= i*3 + (j+1)):
                    ans = ans + 1
        return ans

    def __hash__(self):
        flatState = [j for sub in self.state for j in sub]
        flatState = tuple(flatState)
        return hash(flatState)

    def __gt__(self, other):
        if(self.optimizer==0):
            if(self.getManhattanDistance()>other.getManhattanDistance()):
                return True
            else:
                return False
        elif(self.optimizer==1):
            if(self.getHammingDistance()>other.getHammingDistance()):
                return True
            else:
                return False
        elif(self.optimizer==2):
            if(self.getHammingDistance() + self.getManhattanDistance()
>other.getHammingDistance() + self.getManhattanDistance()):
                return True
```

```
            else:
                return False
        return True

    def __ge__(self, other):
        if(self.optimizer==0):
            if(self.getManhattanDistance() >= other.getManhattanDistance()):
                return True
            else:
                return False
        elif(self.optimizer==1):
            if(self.getHammingDistance() >= other.getHammingDistance()):
                return True
            else:
                return False
        elif(self.optimizer==2):
            if(self.getHammingDistance() + self.getManhattanDistance() >=
other.getHammingDistance() + self.getManhattanDistance()):
                return True
            else:
                return False
        return True

    def __lt__(self, other):
        if(self.optimizer==0):
            if(self.getManhattanDistance()<other.getManhattanDistance()):
                return True
            else:
                return False
        elif(self.optimizer==1):
            if(self.getHammingDistance()<other.getHammingDistance()):
                return True
            else:
                return False
        elif(self.optimizer==2):
            if(self.getHammingDistance() + self.getManhattanDistance() <
other.getHammingDistance() + self.getManhattanDistance()):
                return True
            else:
                return False
        return True
```

```
   def __le__(self, other):
       if(self.optimizer==0):
           if(self.getManhattanDistance()<=other.getManhattanDistance()):
               return True
           else:
               return False
       elif(self.optimizer==1):
           if(self.getHammingDistance()<=other.getHammingDistance()):
               return True
           else:
               return False
       elif(self.optimizer==2):
           if(self.getHammingDistance() + self.getManhattanDistance() <=
other.getHammingDistance() + self.getManhattanDistance()):
               return True
           else:
               return False
       return True

   def __eq__(self, other):
       if(self.optimizer==0):
           if(self.getManhattanDistance() == other.getManhattanDistance()):
               return True
           else:
               return False
       elif(self.optimizer==1):
           if(self.getHammingDistance() == other.getHammingDistance()):
               return True
           else:
               return False
       elif(self.optimizer==2):
           if(self.getHammingDistance() + self.getManhattanDistance() ==
other.getHammingDistance() + self.getManhattanDistance()):
               return True
           else:
               return False
       return True

class Solver:

   def __init__(self,state):
       self.state = state
```

```python
def isSolvable(self):
    '''
        Parameters: State
        Returns: True if state is solvable, otherwise False
    '''
    flatState = [j for sub in self.state for j in sub]
    inversions = 0
    for i in range(len(flatState)-1):
        for j in range(i+1,len(flatState)):
            if flatState[i]!= 0 and flatState[j]!=0 and flatState[i]>flatState[j]:
                inversions = inversions + 1
    return inversions%2==0

def breadth_first_search(self):
    '''
        Parameters: State
        Returns: List of Moves to solve the state, otherwise None if unsolvable
    '''
    if(self.isSolvable()==False):
        return (None,None)

    closed = list()
    q = deque()
    q.append(Node(state = self.state,depth = 0))
    while q:
        node = q.popleft()
        if node.isGoalState():
            return (node.moves,len(closed))
        if node.state not in closed:
            closed.append(node.state)
            for action in node.getAvailableActions():
                q.append(node.getResultFromAction(action))

    return (None,None)

def depth_first_search(self):
    '''
        Parameters: State
        Returns: List of Moves to solve the state, otherwise None if unsolvable
    '''
    if(self.isSolvable()==False):
```

```python
            return (None,None)
        closed = list()
        q = list()
        q.append(Node(state = self.state,depth = 0))
        while q:
            node = q.pop()
            if node.isGoalState():
                return (node.moves,len(closed))
            if node.state not in closed:
                closed.append(node.state)
                for action in node.getAvailableActions():
                    q.append(node.getResultFromAction(action))

        return (None,None)

    def uniform_cost_search(self,optimizer=0):
        '''
            Parameters: State, Optimizer
            Returns: List of Moves to solve the state, otherwise None if unsolvable
        '''
        if(self.isSolvable()==False):
            return (None,None)
        closed = list()
        q = PriorityQueue()
        q.put(Node(state = self.state,depth = 0,optimizer=optimizer))
        while q:
            node = q.get()
            if node.isGoalState():
                return (node.moves,len(closed))
            if node.state not in closed:
                closed.append(node.state)
                for action in node.getAvailableActions():
                    q.put(node.getResultFromAction(action))

        return (None,None)

    def a_star(self):
        '''
            Parameters: State, Optimizer
            Returns: List of Moves to solve the state, otherwise None if unsolvable
        '''
        if(self.isSolvable()==False):
```

```
            return (None,None)
        closed = dict()
        q = PriorityQueue()
        node = Node(state = self.state,depth = 0)
        q.put((node.getManhattanDistance(),node))
        while q:
            dist,node = q.get()
            closed[node] = dist
            if node.isGoalState():
                return (node.moves,len(closed))
            for action in node.getAvailableActions():
                nextNode = node.getResultFromAction(action)
                nextDist = nextNode.getManhattanDistance()
                if nextNode not in closed or nextNode.depth + nextDist <
closed[nextNode]:
                    q.put((nextNode.depth+nextDist,nextNode))
        return (None,None)

def toWord(action):
    '''
        Parameters: List of moves
        Returns: Returns List of moves in Word
    '''
    if(action==0):
        return "Left"
    if(action==1):
        return "Right"
    if(action==2):
        return "Top"
    if(action==3):
        return "Bottom"
initialState =  [[8,2,3],[4,6,5],[7,8,0]]
# [[6,8,5],[2,3,4],[1,0,7]] #[[6,8,5],[2,3,4],[1,0,7]]
# [[13,11,10,7],[6,0,15,2],[14,1,8,12],[5,3,4,9]]
solver = Solver(initialState)
print("Initial State:- {}".format(initialState))
n = Node(state=initialState,depth=0)

print('-------------------------A Star--------------------------------')
startTime = time.time()
moves,nodesGenerated = solver.a_star()
endTime = time.time()
```

```python
if moves is None:
   print("Given State is Unsolvable!")
else:
   wordMoves = list(map(toWord,moves))
   print("Nodes Generated:- {}".format(nodesGenerated))
   print("No. of moves:- {}".format(len(moves)))
   print("Required Moves:- {}".format(wordMoves))
   print("Execution Time:- {:.2f} ms".format((endTime-startTime)*1000))



print('-------------------------UCS--------------------------------')
startTime = time.time()
moves,nodesGenerated = solver.uniform_cost_search()
endTime = time.time()
if moves is None:
   print("Given State is Unsolvable!")
else:
   wordMoves = list(map(toWord,moves))
   print("Nodes Generated:- {}".format(nodesGenerated))
   print("No. of moves:- {}".format(len(moves)))
   print("Required Moves:- {}".format(wordMoves))
   print("Execution Time:- {:.2f} ms".format((endTime-startTime)*1000))



print('-------------------------BFS--------------------------------')
startTime = time.time()
moves,nodesGenerated = (solver.breadth_first_search())
endTime = time.time()
if moves is None:
   print("Given State is Unsolvable!")
else:
   wordMoves = list(map(toWord,moves))
   print("Nodes Generated:- {}".format(nodesGenerated))
   print("No. of moves:- {}".format(len(moves)))
   print("Required Moves:- {}".format(wordMoves))
   print("Execution Time:- {:.2f} ms".format((endTime-startTime)*1000))



print('-------------------------DFS--------------------------------')
```

```
startTime = time.time()
moves,nodesGenerated = (solver.depth_first_search())
endTime = time.time()
if moves is None:
   print("Given State is Unsolvable!")
else:
   wordMoves = list(map(toWord,moves))
   print("Nodes Generated:- {}".format(nodesGenerated))
   print("No. of moves:- {}".format(len(moves)))
   print("Required Moves:- {}".format(wordMoves))
   print("Execution Time:- {:.2f} ms".format((endTime-startTime)*1000))
```

Output:

Example 1:

```
Initial State:- [[8, 2, 3], [4, 6, 5], [7, 8, 0]]
-------------------------A Star--------------------------------
Given State is Unsolvable!
-------------------------UCS--------------------------------
Given State is Unsolvable!
-------------------------BFS--------------------------------
Given State is Unsolvable!
-------------------------DFS--------------------------------
Given State is Unsolvable!
```

Example 2:

```
Initial State:- [[1, 2, 3], [4, 5, 6], [0, 7, 8]]
-------------------------A Star--------------------------------
Nodes Generated:- 3
No. of moves:- 2
Required Moves:- ['Right', 'Right']
Execution Time:- 0.48 ms
-------------------------UCS--------------------------------
Nodes Generated:- 2
No. of moves:- 2
Required Moves:- ['Right', 'Right']
Execution Time:- 0.37 ms
-------------------------BFS--------------------------------
Nodes Generated:- 6
No. of moves:- 2
Required Moves:- ['Right', 'Right']
```

```
Execution Time:- 0.55 ms
------------------------DFS-------------------------------
Nodes Generated:- 2
No. of moves:- 2
Required Moves:- ['Right', 'Right']
Execution Time:- 0.20 ms
```

Example 3:

```
Initial State:- [[6, 8, 5], [2, 3, 4], [1, 0, 7]]
------------------------A Star-------------------------------
Nodes Generated:- 431
No. of moves:- 23
Required Moves:- ['Right', 'Top', 'Top', 'Left', 'Left', 'Bottom', 'Bottom', 'Right',
'Right', 'Top', 'Top', 'Left', 'Bottom', 'Bottom', 'Right', 'Top', 'Top', 'Left',
'Left', 'Bottom', 'Right', 'Bottom', 'Right']
Execution Time:- 104.20 ms
------------------------UCS-------------------------------
Nodes Generated:- 38
No. of moves:- 27
Required Moves:- ['Right', 'Top', 'Left', 'Top', 'Left', 'Bottom', 'Bottom', 'Right',
'Top', 'Top', 'Right', 'Bottom', 'Left', 'Top', 'Left', 'Bottom', 'Bottom', 'Right',
'Right', 'Top', 'Left', 'Bottom', 'Left', 'Top', 'Right', 'Right', 'Bottom']
Execution Time:- 9.10 ms
------------------------BFS-------------------------------
```

As seen in the third example, BFS and DFS are quite slow. On the other hand, A star algorithm has obtained the shortest path and UCS has the least execution time.