Min Kim (msk723)

<u>Instructions:</u>

1) Ensure python 3 is installed so you can run file in command prompt
2) Open your computer's command prompt and go to the directory in which the python file is stored using "cd"
3) Input files must also be in the same directory in order for this to work (inputs will contain initial and goal states from txt file that contains seven lines. Lines 1 to 3 contain the tile pattern for the initial state and lines 5 to 7 contain the tile pattern for the goal state. Line 4 is a blank line. Each tile is separated by a space. The algo shows how to get to the goal state from the initial state using a set of moves: left, up, right down. These moves move the respective tiles into the empty tile square (represented by zero))
4) Run the command: "python search.py" without quotes
5) This is assuming the file is still named "search.py", please input the correct file name
6) By doing this the program will run and you will be prompted for the file name, enter the name of the file ex: "input1.txt" …Note, for correct formatting of output file name please keep input as "input" or "Input" without random capital letters like "inPUT"
7) You will then choose the heuristic model for your search algorithm, please choose "1" or "2" for either the first or second h(n) model
8) Here is an example of a correct input:

Command Prompt

```
C:\Users\Min Kim\Documents\Spring 2019\Aritificial Intelligence\Project>python search.py
Please enter the file name to open: input1.txt
Please enter preferred heuristic function:
 1: Sum of Manhattan Distances
 2: Sum of Manhattan Distances + 2 x # Linear Conflicts
 Choice: 2

C:\Users\Min Kim\Documents\Spring 2019\Aritificial Intelligence\Project>
```

9) This will produce a new file in the same directory replacing "input" with "Output" in the file name and will add "_A" if sum of Manhattan distances was used and "_B" if sum plus 2 * linear conflicts was used, creating an example output file name such as "Output1_A.txt"
10) The output file should show the initial and goal states in line 1-7, the depth level of the goal node (found from the search algo) in line 9, the total number of nodes

generated in line 10, and the solution/ method to solve the puzzle in line 11. Line 12 also contains the f(n) values used in the solution path of the algorithm.

Output1_A.txt

7 1 6

8 3 5

2 0 4

8 7 6

1 0 5

2 3 4

5

12

U U L D R

5 5 5 5 5 5

Output1_B.txt

7 1 6

8 3 5

2 0 4

8 7 6

1 0 5

2 3 4

5

12

U U L D R

5 5 5 5 5 5

Output2_A.txt

2 6 0

1 3 4

7 5 8

1 2 3

4 5 6

7 8 0

10

27

L D R U L L D R D R

10 10 10 10 10 10 10 10 10 10 10

Output2_B.txt

2 6 0

1 3 4

7 5 8

1 2 3

4 5 6

7 8 0

10

26

L D R U L L D R D R

10 12 12 10 10 10 10 10 10 10 10

Output3_A.txt

5 4 3

2 6 7

1 8 0

1 2 3

4 5 6

7 8 0

22

1179

U L U L D D R U U L D D R R U L L D R U R D

12 12 12 12 12 12 12 14 16 16 16 16 18 18 18 20 22 22 22 22 22 22 22

Output3_B.txt

5 4 3

2 6 7

1 8 0

1 2 3

4 5 6

7 8 0

22

612

U L D R U U L L D D R U U R D L U L D R R D

12 14 14 14 16 18 20 22 22 22 22 20 20 20 20 20 22 22 22 22 22 22 22

Output4_A.txt

8 7 3

0 4 5

6 2 1

1 2 3

4 5 6

7 8 0

23

1141

U R D D R U L D L U U R D R D L L U U R D R D

17 17 17 19 19 19 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23

Output4_B.txt

8 7 3

0 4 5

6 2 1

1 2 3

4 5 6

7 8 0

23

539

U R D D R U L D L U U R D R D L L U U R D R D

17 17 17 19 21 21 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23 23

Source Code:

```
from queue import PriorityQueue

from copy import deepcopy


#function opens user_inputted file and reads goal and initial states into lists, return lists and file name

def readFile():

    initial = []
```

```
    goal = []
    #input for file
    fileName = input("Please enter the file name to open: ")
    #open file with inputed text
    with open(fileName) as f:
        line = f.readline()
        count = 1   #line number
        #for each line (getting rid of white space and new line) create lists for the initial and goal
states
        while line:
            for num in line:
                if(num != ' ' and num != '\n'):
                    #goal on lines 5-7 out of 1-7
                    if (count > 4):
                        goal.append(num)
                    #goal on lines 1-3 out of 1-7
                    elif count <= 3:
                        initial.append(num)
            line = f.readline()
            count += 1
    return (initial,goal), fileName


#calculates manhattan distance between current state and goal using 2 passed states, returns the
distance sum
def manhattanDistance(currState, goalState):
    total = 0
    index = 0
    #check distance for every position/value in the state
    for num in currState:
        #do not include blank space (0)
```

```
        if num != '0':
            #can determine row and column numbers mathmatically based on index
            initRow, initCol = index % 3, int(index / 3)
            goalIndex = goalState.index(num)
            goalRow, goalCol = goalIndex % 3, int(goalIndex / 3)
            #manhattan distance formula
            total += (abs(goalRow - initRow) + abs(goalCol - initCol))
        index += 1
    return total

#detects # of linear conflicts in a state with goal using 2 passed states, returns answer
def linearConflicts(currState, goalState):
    total = 0
    currInd1 = 0
    #check for conflicts at every position in current state
    for currNum1 in currState:
        if currNum1 != '0':
            #calculate the row and columns based on index
            initRow1, initCol1 = int(currInd1 / 3), currInd1 % 3
            currInd1 += 1
            currInd2 = 0
            #compare with self to find numbers on the same row or column
            for currNum2 in currState:
                if currNum2 != '0':
                    initRow2, initCol2 = int(currInd2 / 3), currInd2 % 3
                    currInd2 += 1
                    #check for matching row #'s as well as if one is farther down than the other
                    if initRow1 == initRow2 and initCol1 < initCol2:
                        proceed = True
                    # check for matching column #'s as well as if one is farther right than the other
```

```
                elif initCol1 == initCol2 and initRow1 < initRow2:
                    proceed = True
                else:
                    proceed = False
                if proceed:
                    #calculate goal state row and column numbers based on index
                    goalIndex = goalState.index(currNum1)
                    goalRow1, goalCol1 = int(goalIndex / 3), goalIndex % 3
                    goalIndex = goalState.index(currNum2)
                    goalRow2, goalCol2 = int(goalIndex / 3), goalIndex % 3
                    #use these to detect linear conflict by seeing if positions are swapped from the
original
                    #checks if the rows or columns match then if it is the opposite direction (further
down or left)
                    if (goalRow1 == goalRow2 and goalCol1 > goalCol2) and goalRow2 ==
initRow2:
                        proceed = True
                    elif goalCol1 == goalCol2 and goalRow1 > goalRow2 and goalCol2 == initCol2:
                        proceed = True
                    else:
                        proceed = False
                    if proceed:
                        total += 1
            else:
                currInd2 += 1
        else:
            currInd1 += 1
    return total

#stores state information as well as all information about the node in the tree
class Node:
```

```
    def __init__(self, state, goalState, parentNode = None, move = None, choice = 1):
        self.state = state
        self.parent = parentNode
        if parentNode is None:
            self.depth = 0
        else:
            self.depth = parentNode.depth + 1
        self.fValue = self.costFunction(state, goalState, choice)
        self.choice = choice
        self.move = move

    #calculates function cost f(n) using g(n) and h(n) chosen using users preference and passed
states, returns the sum
    def costFunction(self, state, goalState, choice):
        # adds h(n) to g(n) / the depth, results in f(n)
        total = manhattanDistance(state, goalState) + self.depth
        #if user chooses different heuristic function, add extra value
        if choice == 2:
            total += (2*(linearConflicts(state, goalState)))
        return total

    #future comparison methods allows use of nodes in priority queue with node objects
    #check == for this class
    def __eq__(self, other):
        return self.fValue == other.fValue

    #checks >
    def __gt__(self, other):
        return self.fValue > other.fValue
```

```
    #check <
    def __lt__(self, other):
        return self.fValue < other.fValue

    #checks <=
    def __le__(self, other):
        return self == other or self < other

    # checks >=
    def __ge__(self, other):
        return self == other or self > other

    # checks !=
    def __ne__(self, other):
        return not (self == other)

#class represents the puzzle problem and organizes all nodes in search tree while also containing the search algorithm
class EightPuzzle:
    def __init__(self, initial, goalState, choice = 1):
        self.initial = Node(initial, goalState, None, None, choice)
        self.goal = Node(goalState, goalState, None, None, choice)
        #frontier is the priority queue that selects the lowest costing f(n) value
        self.frontier = PriorityQueue()
        #graph search needs closed list to avoid repeating states
        self.closed = []
        #starts total nodes at one to include root node
        self.totalNodes = 1
```

```
    #inserts node into frontier if it is not in the closed list, is passed state, parent, move, and choice to create node
    def insertNode(self, newState, parent, move, choice):
        newNode = Node(newState, self.goal.state, parent, move, choice)
        proceed = True
        #checks if state was already found
        for closedNode in self.closed:
            if newNode.state == closedNode.state:
                proceed = False
        if proceed:
            self.frontier.put(newNode)
            self.totalNodes += 1


    #expand node in tree with possible moves, take the passed node and choice to create new nodes
    def expandNodes(self, currNode, choice):
        #check moves possible based on blank space location
        index = currNode.state.index('0')
        row, col = int(index / 3), index % 3
        #deep copies the state to avoid changing it preemptively before differnt kind of moves
        newState = deepcopy(self.closed[-1]).state
        #next if statements make the move on the state and pass it to be created as a node
        #move Up
        if row > 0:
            newState[index] = newState[index - 3]
            newState[index - 3] = '0'
            self.insertNode(newState, currNode, 'U', choice)
            newState = deepcopy(self.closed[-1]).state
        # move Down
        if row < 2:
```

```
            newState[index] = newState[index + 3]
            newState[index + 3] = '0'
            self.insertNode(newState, currNode, 'D', choice)
            newState = deepcopy(self.closed[-1]).state
        # move Left
        if col > 0:
            newState[index] = newState[index - 1]
            newState[index - 1] = '0'
            self.insertNode(newState, currNode, 'L', choice)
            newState = deepcopy(self.closed[-1]).state
        # move Right
        if col < 2:
            newState[index] = newState[index + 1]
            newState[index + 1] = '0'
            self.insertNode(newState, currNode, 'R', choice)
            newState = deepcopy(self.closed[-1]).state
        #returns the next best move based on cost
        return self.frontier.get()

    #the search algorithm that keeps running based on choice of h(n) until completion
    def aStarSearchAlgo(self, choice):
        currNode = self.initial
        #while the state is not the goal state
        while self.goal.state != currNode.state:
            #add explored node to the closed list and expand
            self.closed.append(deepcopy(currNode))
            currNode = self.expandNodes(currNode, choice)
        return currNode
```

```
    #retrieve lists containing the paths taken and the costs from the goal to the root, uses a node to trace up using parents
    def getPathLists(self, finalNode):
        moves = []
        costs = []
        moves.append(finalNode.move)
        costs.append(finalNode.fValue)
        node = finalNode.parent
        #while the node has not found the parent, keeping recording the moves and costs
        while node is not None:
            moves.append(node.move)
            costs.append(node.fValue)
            node = node.parent
        return moves, costs



def main():
    states, filename = readFile()
    #get user input on correct heuristic function
    userChoice = input("Please enter preferred heuristic function:\n 1: Sum of Manhattan Distances\n 2: Sum of Manhattan Distances + 2 x # Linear Conflicts\n Choice: ")
    #creates the puzzle object
    myPuzzle = EightPuzzle(states[0], states[1], int(userChoice))
    #finds the last node in the correct path found by the algorithm
    result = myPuzzle.aStarSearchAlgo(int(userChoice))

    #creates the file name
    filename = filename.replace("input", "Output")
    filename = filename.replace("Input", "Output")
    if int(userChoice) == 1:
```

```
    filename = filename.replace(".", "_A.")
else:
    filename = filename.replace(".", "_B.")
#creates file and writes the inital and goal states
file = open(filename, "w")
count = 0
for num in states[0]:
    if count % 3 == 0 and count != 0:
        file.write("\n")
    file.write(num + " ")
    count += 1
file.write("\n")
for num in states[1]:
    if count % 3 == 0 and count != 0:
        file.write("\n")
    file.write(num + " ")
    count += 1
file.write("\n\n")

#writes the depth and total nodes, then the correct moves and costs from root to goal node
file.write(str(result.depth) + "\n" + str(myPuzzle.totalNodes) + "\n")
moves, costs = myPuzzle.getPathLists(result)
moves = moves[::-1]
costs = costs[::-1]
for myMove in moves:
    if myMove is not None:
        file.write(myMove + " ")
file.write("\n")
for num in costs:
    file.write(str(num) + " ")
```

main()