Project 1 (15 Puzzle)
CS 4613 Artificial Intelligence
Professor Edward Wong
Gina Joerger
April 4, 2020

How To Run The Program:

To run the program, simply run the code provided in the A*.txt file in a python compiler. To change which input text document will be addressed in the program, the code in the main function will have to be rewritten, so instead of file = open('Input1.txt', 'r'), another document can be called. To have an output with a different name than Output1.txt, the line of code stating sys.stdout = open('Output1.txt','wt') will also have to be adjusted. These two lines are both in the main() function.

Source Code:

```
from queue import PriorityQueue
import sys

numNodes = 0

class Node:
    def __init__(self, state, parent, operator, depth, pathCost):
        self.state = state
        self.parent = parent
        self.children = list()
        self.operator = operator
        self.depth = depth
        self.pathCost = pathCost

    def same(self, state): #Compares two states, sees if they are equal and/or the goal state.
        if self.state == state:
            return True
        else:
            return False

    def __lt__(self, other): #Compares the path cost of two states.
        return self.pathCost < other.pathCost

    def move(self): #Creates the children af†er checking if the moves are possible.
        global numNodes
        zeroIndex = self.state.index(0)

        new = self.state[:]
        if zeroIndex not in [0, 1, 2, 3]: # Makes children after going up
            temp = new[zeroIndex - 4]
```

```
            new[zeroIndex - 4] = new[zeroIndex]
            new[zeroIndex] = temp
            child = Node(new, self, "U", self.depth + 1, self.pathCost)
            self.children.append(child)
            numNodes += 1

        new = self.state[:]
        if zeroIndex not in [3, 7, 11, 15]:  # Makes children after going right
            temp = new[zeroIndex + 1]
            new[zeroIndex + 1] = new[zeroIndex]
            new[zeroIndex] = temp
            child = Node(new, self, "R", self.depth + 1, self.pathCost)
            self.children.append(child)
            numNodes += 1

        new = self.state[:]
        if zeroIndex not in [12, 13, 14, 15]:  # Makes children after going down
            temp = new[zeroIndex + 4]
            new[zeroIndex + 4] = new[zeroIndex]
            new[zeroIndex] = temp
            child = Node(new, self, "D", self.depth + 1, self.pathCost)
            self.children.append(child)
            numNodes += 1

        new = self.state[:]
        if zeroIndex not in [0, 4, 8, 12]:  # Makes children after going left
            temp = new[zeroIndex - 1]
            new[zeroIndex - 1] = new[zeroIndex]
            new[zeroIndex] = temp
            child = Node(new, self, "L", self.depth + 1, self.pathCost)
            self.children.append(child)
            numNodes += 1

def aStar(initial, final): #A* Algorithm
    p = PriorityQueue()
    p.put(Node(initial, None, "", 0, 0))
    GoalFound = False

    while p and not GoalFound: #Until goal is found, the move function is recursed.
        node = p.get()
        node.move()
        for child in node.children: #For every child, the pathCost is updated with the proper path
cost
            if child.same(final):
                GoalFound = True
                x = child.depth + manhattanH(child.state, final)
```

```
            path(child, x)

        cost = child.depth + manhattanH(child.state, final)

        child.pathCost = cost

        p.put(child, cost)

def manhattanH(state, final): #Sums the distances of each state to its place in the final state.
    x = 0
    for i in range(0, 16):
        x += manhattanA(state.index(i), final.index(i))
    return x


def manhattanA(x, y): #Auxiliary function for the manhattan distance heuristic.
    #Matrix coordinates from the state list.
    m = {0: (1, 1), 0.25: (1, 2), 0.50: (1, 3), 0.75: (1, 4),
         1: (2, 1), 1.25: (2, 2), 1.50: (2, 3), 1.75: (2, 4),
         2: (3, 1), 2.25: (3, 2), 2.50: (3, 3), 2.75: (3, 4),
         3: (4, 1), 3.25: (4, 2), 3.50: (4, 3), 3.75: (4, 4)}

    x1, y1 = m[x/4]
    x2, y2 = m[y/4]
    return abs(x1-x2) + abs(y1-y2)


def path(Node, x): #Prints depth of node, then goes to each parent node to obtain the operator
needed to create the path to solution.
    node = Node
    path = list()
    f = list()

    path.append(node.operator) #Gets the operator of last node
    f.append(node.pathCost) #Gets the path cost of last node
    depth = node.depth

    while node.parent is not None: #Recurses up the A* graph until the root node is found
        node = node.parent
        path.insert(0, node.operator)
        f.insert(0, node.pathCost)

    del path[0]
    del f[-1]

    f.insert(len(f), x)
```

```
    print("\n%d" % depth) #Prints depth
    print(numNodes) #Prints number of nodes
    for item in range(len(path)): #Prints the directions
        if item == (len(path)-1):
            print(path[item], end="\n")
        else:
            print(path[item], end=" ")
    for item in f: #prints the f(n) value
        print(item, end=" ")

def main(): # Reads given file, takes the input and output. Puts them into the A* funtion, then
outputs results into a text file.
    insert = []
    finalinsert = []
    initial = []
    final = []

    sys.stdout = open('Output1.txt','wt') #Makes output txt file for writing
    file = open('Input1.txt', 'r') #Opens the Input File to read

    for i in range(0, 4): #Reads each line and puts it in a list
        x = file.readline()
        print(x, end='') #Prints onto output text file
        insert.append(x)

    for i in insert: #Makes the items in list consistent for A*
        x = list(map(int, i.split()))
        for i in x:
            initial.append(i)

    for i in range(7, 12): #Reads each line and puts it in a list
        x = file.readline()
        print(x, end='') #Prints onto output text file
        finalinsert.append(x)

    for i in finalinsert: #Makes the items in list consistent for A*
        x = list(map(int, i.split()))
        for i in x:
            final.append(i)

    file.close()

    aStar(initial, final)
```

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

Output 1

```
1 2 3 4
5 6 0 7
8 9 10 11
12 13 14 15

1 2 3 4
5 9 6 7
8 13 0 11
12 14 10 15

5
22
L D D R U
0 7 6 7 6 5
```

Output 2

```
1 5 3 13
8 0 6 4
15 10 7 9
11 14 2 12

1 5 3 13
8 10 6 4
0 15 2 9
11 7 14 12

6
39
D R D L U L
0 7 8 9 8 7 6
```

Output 3

9 13 7 4
12 3 0 1
2 15 5 6
14 10 11 8

13 3 7 4
9 1 0 6
12 2 5 8
14 15 10 11

12
96
R D D L L U L U U R D R
0 13 14 15 14 15 14 15 14 15 14 13 12

Output 4

13 12 2 11
10 1 8 9
0 3 15 14
6 4 7 5

10 13 12 11
8 1 2 9
3 4 15 5
6 0 14 7

16
817
R U R U L L D R D R R D L U L D
0 13 16 17 18 17 18 17 16 15 18 19 18 17 18 17 16