Project 1: 15-Puzzle Game

CS-UY 4563

By: Tomas David and Mateo xxx

Due: April 9th 2021

Instructions

1. Install python on your console with “Brew” or “pip” on your root directory as such:

   “Brew install python”

2. Download the entire folder as it is. Then navigate to the directory “15PuzzleAIProject” as such :

   “cd 15PuzzleAIProject”.

3. Then run “python 15PuzzleProblem.py”.

4. You will be prompted to enter an input file name with, until a match is found. Press enter once you have written it down in the console. Note: only files in the root of the project directory will be found, otherwise you must include the relative path to this directory.

5. The generated output will be written down in “sampleoutput.txt”. Note: this file gets re-written every time the algorithm is run with a new input. If you want to keep a record of the output, make a copy of “sampleoutput.txt” before re-running the code.

6. The output files of “InputFile1”, “InputFile2”, “InputFile3”, have been included in the directory for your convenience.

Source Code

```python
from queue import PriorityQueue
import copy
import sys
```

```
# state: 2D array (str)
# parent: Node
# action: int representing action (1-8)
# path_cost: Cost from root to node n (int)
class Node:
    #Basic attributes of every node in the tree
    def __init__(self, state, parent=None, action=None,
path_cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.path_cost = path_cost

    #Shortcut for less than

    def __lt__(self, other):
        self_mag = sumOfChessboardDistances(self.state)
+ self.path_cost
        other_mag =
sumOfChessboardDistances(other.state) + other.path_cost
        return self_mag < other_mag


class Problem:
    def __init__(self, initial, goal, actions):
        self.initial = initial
        self.goal = goal
        self.actions = actions

    def isGoal(self, state):
        return state == goal

    # Returns new_state resulting from doing action on
state
    # Could probably cut down
    def result(self, state, action):
        new_state = copy.deepcopy(state)

        blank_row = -1
        blank_col = -1

        for i in range(0, 4, 1):
            for j in range(0, 4, 1):
                if(new_state[i][j] == '0'):
                    blank_row = i
                    blank_col = j

        # Define each action depending on where the
blank space is gonna move to

        if(action == 1):
            if(blank_col == 0):
                return new_state
            new_state[blank_row][blank_col] =
new_state[blank_row][blank_col - 1]
            new_state[blank_row][blank_col - 1] = '0'
        elif(action == 2):
            if(blank_row == 0 or blank_col == 0):
                return new_state
            new_state[blank_row][blank_col] =
new_state[blank_row - 1][blank_col - 1]
            new_state[blank_row - 1][blank_col - 1] =
'0'
        elif(action == 3):
            if (blank_row == 0):
                return new_state
            new_state[blank_row][blank_col] =
new_state[blank_row - 1][blank_col]
            new_state[blank_row - 1][blank_col] = '0'
        elif(action == 4):
            if (blank_row == 0 or blank_col == 3):
                return new_state
            new_state[blank_row][blank_col] =
new_state[blank_row - 1][blank_col + 1]
            new_state[blank_row - 1][blank_col + 1] =
'0'
        elif(action == 5):
            if (blank_col == 3):
                return new_state
            new_state[blank_row][blank_col] =
new_state[blank_row][blank_col + 1]
            new_state[blank_row][blank_col + 1] = '0'
        elif (action == 6):
            if (blank_row == 3 or blank_col == 3):
                return new_state
            new_state[blank_row][blank_col] =
new_state[blank_row + 1][blank_col + 1]
            new_state[blank_row + 1][blank_col + 1] =
'0'
        elif (action == 7):
            if (blank_row == 3):
                return new_state
            new_state[blank_row][blank_col] =
new_state[blank_row + 1][blank_col]
            new_state[blank_row + 1][blank_col] = '0'
        elif (action == 8):
            if (blank_row == 3 or blank_col == 0):
                return new_state
            new_state[blank_row][blank_col] =
new_state[blank_row + 1][blank_col - 1]
            new_state[blank_row + 1][blank_col - 1] =
'0'

        return new_state

    # ActionCost will always be 1 (just for
comprehension in Expand function - can delete when
done)
    def actionCost(self, state, action, new_state):
        return 1


# A* Algorithm
def bestFirstSearch(prob):
    node = Node(prob.initial)

    # Priority Queue
    # f(n) = h(n) + g(n)
    frontier = PriorityQueue()
    frontier.put(node)

    # Lookup table
    # Using str of list as key
    reached = {}
    reached[str(prob.initial)] = node

    num_nodes = 0

    while not frontier.empty():
        node = frontier.get()
```

```python
        if(prob.isGoal(node.state)):
            return (node, num_nodes)

        for child in expand(prob, node):
            curr_state = child.state


            if(str(curr_state) not in reached or
child.path_cost < reached[str(curr_state)].path_cost):
                # num_nodes updated here
                num_nodes += 1
                reached[str(curr_state)] = child
                frontier.put(child)

    return None


# Expands a node
def expand(problem, node):
    curr_state = node.state
    ret = []
    for action in problem.actions:
        new_state = problem.result(curr_state, action)
        cost = node.path_cost +
problem.actionCost(curr_state, action, new_state)
        ret.append(Node(new_state, node, action,cost))
        yield Node(new_state, node, action, cost)


# Could cut down (or one line)
# Uses dictionary, could use a goal state parameter
instead
def chessboardDistance(tile, row, col):
    goal_row = goal_dict[tile][0]
    goal_col = goal_dict[tile][1]

    row_dist = abs(goal_row - row)
    col_dist = abs(goal_col - col)

    return max(row_dist, col_dist)


# Heuristic Function
```

```
def sumOfChessboardDistances(curr_state):
    sum = 0

    for i in range(0, 4, 1):
        for j in range(0, 4, 1):
            if(curr_state[i][j] != "0"):
                curr_dist =
chessboardDistance(curr_state[i][j], i, j)
                sum += curr_dist

    return sum

# ----------------------------------------------- Main
-------------------------------------------------------
-----


initial = []
goal = []


#Ask to initialize the file until found

while True:
    print("\nEnter the name of the input, ensuring that
you type .txt afterwards.\n\nOutput will be written in
sampleoutput.txt. Please check your directory. ")
    file_name = input();
    try:
        input_file = open(file_name)
        if input_file:
            break
    except IOError:
        print ("There is no such a file, please try
again")

input_str = input_file.read()

# Splits with each new line
split_input = input_str.split('\n')

#Clean the output file

file = open("sampleoutput.txt","r+")
```

```python
file.truncate(0)
file.close()

#Open it for writing

sys.stdout = open("sampleoutput.txt", "w")


switch = False

# Splits with each space
# Appends to initial if swtich = False
# else appends to goal
for arr in split_input:
    if arr == '':
        switch = True
        continue

    if not switch:
        initial.append(arr.split(" "))
    else:
        goal.append(arr.split(" "))


# Dictionary for chessboard distance
goal_dict = {}
for row in range(0, 4, 1):
    for col in range(0, 4, 1):
        if goal[row][col] != "0":
            goal_dict[goal[row][col]] = (row, col)

# Solving the problem
p = Problem(initial, goal, [1, 2, 3, 4, 5, 6, 7, 8])
sol = bestFirstSearch(p)
node = sol[0]
num_of_nodes = sol[1]

# Actions and f values
# d number of actions
# d + 1 number of f values
acts = []
fs = []
while True:
    if(node.parent == None):
        fs.insert(0,
sumOfChessboardDistances(node.state))
        break
    acts.insert(0, node.action)
    fs.insert(0, sumOfChessboardDistances(node.state))
    node = node.parent

depth = len(acts)

print("---------------OUTPUT-----------------")
# Prints the initial and goal states
for row in range(0, 4, 1):
    for col in range(0, 4, 1):
        print(initial[row][col], end = " ") # n
    print()

print()

for row in range(0, 4, 1):
    for col in range(0, 4, 1):
        print(goal[row][col], end = " ") # m
    print()

print()

# Can just print this instead of initial and goal
separately
# print(input_str)

# d
print(depth)

# N
print(num_of_nodes)

# A
for a in acts:
    print(a, end=" ")

print()

# F
for f in fs:
    print(f, end=" ")
print()



input_file.close()
sys.stdout.close()

# Checks if the solution node has the goal state
#for row in range(0, 4, 1):
#    for col in range(0, 4, 1):
#        print(sol[0].state[row][col], end=" ")






#--------------------------------------------TO DO-----
----------------------------------
# Is it possible to get depth inside of bestFirstSearch
unsure
# Clean up the file (Change some names for clarity, cut
down unnecessary code, add comments, etc.)
# Create PDF w/ instructions
```

Input Files and Output Files

Input File 1

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

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

Input File 2

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

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

Input File 3

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

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

Output File 1

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

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

6
31
```

```
6 5 8 1 2 3
6 5 4 3 2 1 0
```

Output File 2

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

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

11
106
7 8 2 3 4 7 5 4 7 7 1
10 10 9 8 7 6 5 4 3 2 1 0
```

Output File 3

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

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

16
203
7 5 4 5 3 2 1 8 6 7 4 6 3 2 1 2
15 14 13 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```