CS 6613 AI 1: Project

Vishnu Beji

November 15, 2023

1 How to Run the program

The source code for 26 puzzle problem is written in Python language. I have provided two different methods for the convenience of the grader.

1. Jupyter Notebook .ipynb file
2. Python Script .py files

You may run any one of the above (preferably .ipynb notebook file)

1.1 Jupyter Notebook: 26puzzle.ipynb

The entire code is provided in **26puzzle.ipynb** containing 3 cells, with main function in the 3rd cell. You may simply run each cell sequentially to get the output. You may use Jupyter Notebook or VSCode to run the notebook. Make sure that input files are in the same directory as .ipynb file. Otherwise, change the directory to the desired location at variable `input` in main.

1.2 Python Script: Command Line or IDE-based execution

If you wish to run the .py files in an IDE (like VSCode) or via the command line you may check the directory vb2409_AI_project/Python_Files, containing 3 .py files. Simply run main.py using

```
python main.py
```

Make sure the input files are present in the same directory as main.py. The folder also contains puzzle.py and puzzleOperations.py. In the submission, I am also attaching the input files for ease of use. Output .txt files will be generated in the same directory.

2 Source Code (ipynb file)

2.1 Class Puzzle

Class Puzzle encapsulates reading of input file, storing the values of start and goal grids in NumPy arrays and defining the set of actions possible

[31]:
```
import numpy as np
import heapq
```

```
#Class puzzle will contain the functions required to initialise our puzzle grid, goal grid and actions.
#We are currently treating the size to be variable and it will work for a larger puzzle as well, the value \
#of 3 is given in main
class Puzzle:
    def __init__(self, input_file, size):
        self.size = size
        self.startgrid = np.zeros((self.size, self.size, self.size), dtype=int)
        self.goalgrid = np.zeros((self.size, self.size, self.size), dtype=int)
        self.actions = ['N','E','W','S','U','D']
        self.load_puzzle(input_file)

    #Loads the startgrid and goalgrid arrays by reading the input file
    def load_puzzle(self, input_file):
        with open(input_file, 'r') as file:
            # Read the entire file content into a string
            self.file_content = file.read().strip()

        #We split by double EOL to get each 3x3 layer of the puzzle
        layers = self.file_content.split('\n\n')

        #We get line by line and thereafter space separated numbers to fill into the arrays
        for i in range(self.size):
            lines = layers[:self.size][i].split('\n')
            for j in range(self.size):
                nums = lines[j].split(' ')
                for k in range(self.size):
                    self.startgrid[i][j][k] = nums[k]

        for i in range(self.size):
            lines = layers[self.size:][i].split('\n')
            for j in range(self.size):
                nums = lines[j].split(' ')
                for k in range(self.size):
                    self.goalgrid[i][j][k] = nums[k]

    #Prints the input file as such, as defined in the output format
    def print_start_and_goal_grid(self):
        print(self.file_content)

    #Prints the input file as such, to a defined output file f
    def print_to_file(self, f):
        print(self.file_content, file=f)
```

2.2 Helper Functions

[32]:
```
#Calculate manhattan distance by adding absolute value of difference in i, j and␣
 ↪k
def manhat_distance(cur_state, goal_state):
    manhat_dist = 0
    for i in range(size):
        for j in range(size):
            for k in range(size):
                i2, j2, k2 = np.argwhere(cur_state == goal_state[i, j, k])[0]
                if(goal_state[i, j, k]!=0):
                    manhat_dist += abs(i-i2)+abs(j-j2)+abs(k-k2)
    return manhat_dist

#Computes what the next state will be based on current state and what action is␣
 ↪performed
def compute_next_state(cur_state, action):
    i, j, k = np.argwhere(cur_state == 0)[0]
    i2, j2, k2 = i, j, k
    if(action == 'N'):
        j2 = j-1
    if(action == 'S'):
        j2 = j+1
    if(action == 'W'):
        k2 = k-1
    if(action == 'E'):
        k2 = k+1
    if(action == 'U'):
        i2 = i-1
    if(action == 'D'):
        i2 = i+1

    #Handling edge cases with if condition to avoid invalid actions
    if(i2<size and i2>=0 and j2<size and j2>=0 and k2<size and k2>=0):
        #We create a copy to avoid overwriting, as list would be passed by␣
 ↪reference
        next_state = np.copy(cur_state)
        # Swap the values
        next_state[i, j, k], next_state[i2, j2, k2] = next_state[i2, j2, k2],␣
 ↪next_state[i, j, k]
        return next_state
    else:
        return None

#This generator will generate a monotonically increasing counter everytime it␣
 ↪gets called.
#It is called everytime we generate a new node.
```

```
#This will be the second value in priority queues' sorting key, to pick the␣
 ↪older element first,
#once multiple states have same priority(f-value) and thereby break the tie.
def monotonically_increasing_generator():
    current_value = 0
    while True:
        current_value += 1
        yield current_value

#This is the function that searches for goal state using A* algorithm with the␣
 ↪heuristic
#f(node) = manhattan_distance(node,goal) + distance(start,node)
def solve(start_state, goal_state, actions):
    frontier = []
    visited = [tuple(start_state.ravel())]
    mono_gen = monotonically_increasing_generator()

    start_dist = manhat_distance(start_state, goal_state)

    start_node = (start_dist, next(mono_gen), start_state, 0, [], [start_dist])
    heapq.heappush(frontier, start_node)
    #boolean variable to check if we have reached a goal state
    found = False
    while(frontier):
        #We use this heapq to maintain out priority queue. Node contains details␣
 ↪about the state,
        #the actions taken to reach till there, the value of f (distance)
        cur_node = heapq.heappop(frontier)

        cur_state,  cur_depth, cur_actionpath, cur_dist_list = cur_node[2:6]

        #If we reach a goal we return the node and the number of nodes generated␣
 ↪and halt the iteration
        if np.array_equal(cur_state, goal_state):
            found = True
            #Here next(mono_gen)-1 will give the output as the total number of␣
 ↪nodes generated
            return cur_node, next(mono_gen)-1

        #Here we generate child nodes for every node expanded
        for action in actions:
            #First compute the set of next states based on actions
            next_state = compute_next_state(cur_state, action)

            #Check if the action would have led to a valid state
            if next_state is not None:
```

```
                #We flatten the 3D arr and convert to tuple as np array cannot␣
↪be used for direct comparison
                next_state_tuple = tuple(next_state.ravel())
                #We check if the state has been reached previously and avoid if␣
↪yes, as it is graph search
                if next_state_tuple not in visited:
                    visited.append(next_state_tuple)
                    depth = cur_depth + 1

                    #Calculating f(node) = manhattan_distance(node,goal) +␣
↪distance(start,node)
                    dist = manhat_distance(next_state, goal_state) + depth
                    dist_list = cur_dist_list[:]
                    dist_list.append(dist)

                    #Compute action path by taking parent's actionpath and␣
↪appending the current action
                    actionpath = cur_actionpath[:]
                    actionpath.append(action)

                    #Node is generated with its distance(priority key), counter,␣
↪state, depth, actionpath
                    #and distance list
                    #We use monotonic generator to handle cases with same␣
↪priority, priority will be
                    #given to older entries in the priority queue. We prioritise␣
↪by distance.
                    next_node = (dist, next(mono_gen), next_state, depth,␣
↪actionpath, dist_list)

                    #Push the child into the priority queue with given␣
↪information
                    heapq.heappush(frontier, next_node)

    if(found == False):
        print("No solution exists")
        return
```

2.3 Main Function

[33]:
```
if __name__ == "__main__":
    size = 3
    input = 'input1.txt'
    output = 'output1.txt'
    #Initialise puzzle object with contents from the input file
    puzzle = Puzzle(input, size)
```

```
    #solve function takes the start state, goal state and actions to search and␣
↪return goal node
    #containing all the meta info about the search and number of nodes generated
    goal_node, num_gen_nodes = solve(puzzle.startgrid, puzzle.goalgrid, puzzle.
↪actions)

    #Retrieving depth, action path and list of distances from goal node
    depth_goal_node = goal_node[3]
    actionpath_goal_node = ' '.join(goal_node[4])
    distlist_goal_node = ' '.join(map(str, goal_node[5]))

    #Outputs
    puzzle.print_start_and_goal_grid()
    print("", depth_goal_node, num_gen_nodes, actionpath_goal_node,␣
↪distlist_goal_node, sep='\n')

    #Write the ouputs to a file
    with open(output, 'w') as f:
        # Redirect the last two print statements to the file
        puzzle.print_to_file(f)
        print("", depth_goal_node, num_gen_nodes, actionpath_goal_node,␣
↪distlist_goal_node, sep='\n', file=f)
```

3 Outputs

3.1 Output for input1.txt

```
1 2 3
4 0 5
6 7 8

9 10 11
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

1 2 3
4 13 5
6 7 8

9 10 11
15 12 14
24 16 17
```

```
18 19 20
21 0 23
25 22 26

6
23
D W S D E N
6 6 6 6 6 6 6
```

3.2 Output for input2.txt

```
1 2 3
4 0 5
6 7 8

9 10 11
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

1 10 2
4 5 3
6 7 8

9 13 11
21 12 14
15 16 17

18 0 20
24 19 22
25 26 23

13
44
E N W D S W D S E E N W N
13 13 13 13 13 13 13 13 13 13 13 13 13 13
```

3.3 Output for input3.txt

```
1 2 3
4 0 5
6 7 8

9 10 11
```

```
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

0 2 3
1 7 14
6 8 5

12 9 10
4 13 11
21 16 17

18 19 20
22 25 23
15 24 26

16
59
S E N D N W W S D E S W U N U N
16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
```