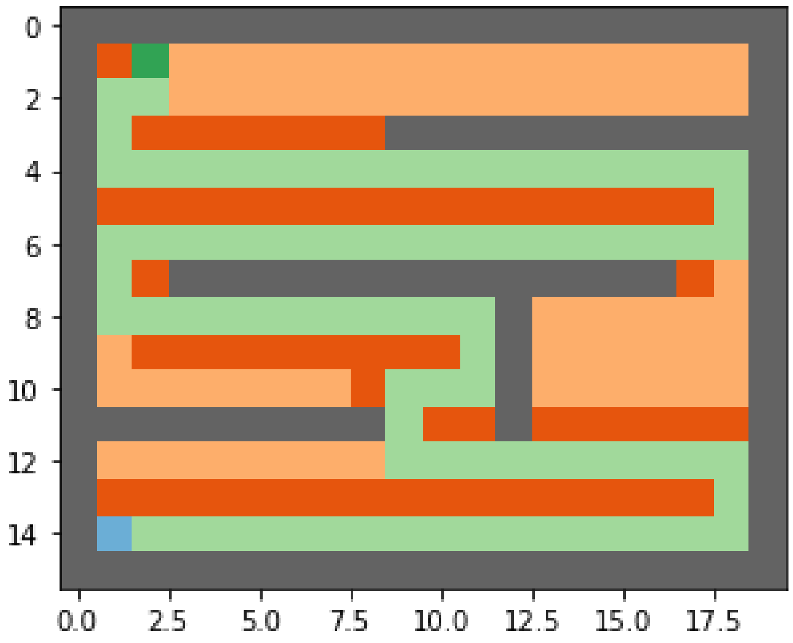
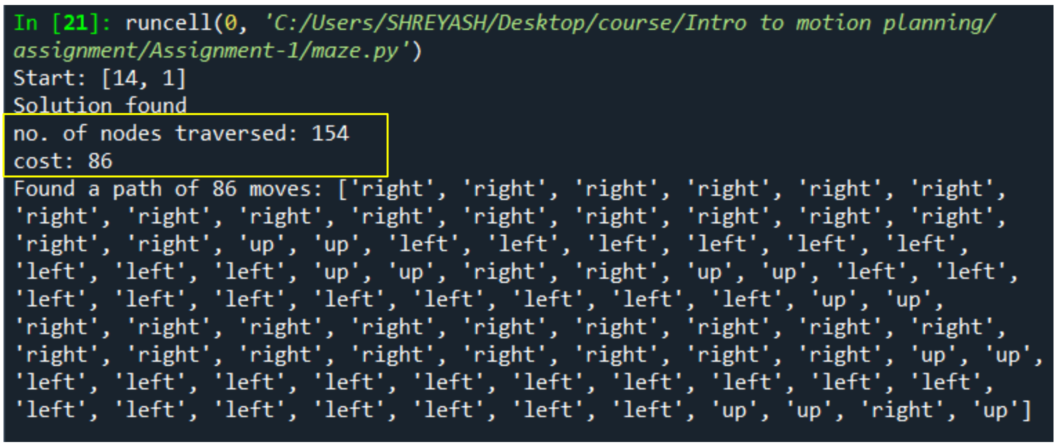
ED22S016
Shreyash Gadgil

Q1

A. Depth first search





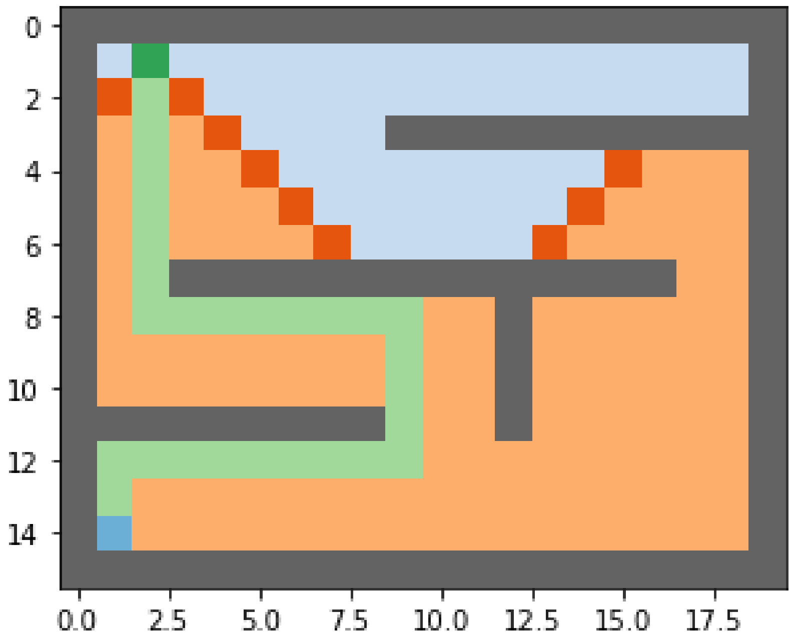
```
In [21]: runcell(0, 'C:/Users/SHREYASH/Desktop/course/Intro to motion planning/
assignment/Assignment-1/maze.py')
Start: [14, 1]
Solution found
no. of nodes traversed: 154
cost: 86
Found a path of 86 moves: ['right', 'right', 'right', 'right', 'right', 'right',
'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right',
'right', 'right', 'up', 'up', 'left', 'left', 'left', 'left', 'left', 'left',
'left', 'left', 'left', 'up', 'up', 'right', 'right', 'up', 'up', 'left', 'left',
'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up',
'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right',
'right', 'right', 'right', 'right', 'right', 'right', 'right', 'right', 'up', 'up',
'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left', 'left',
'left', 'left', 'left', 'left', 'left', 'left', 'left', 'up', 'up', 'right', 'up']
```

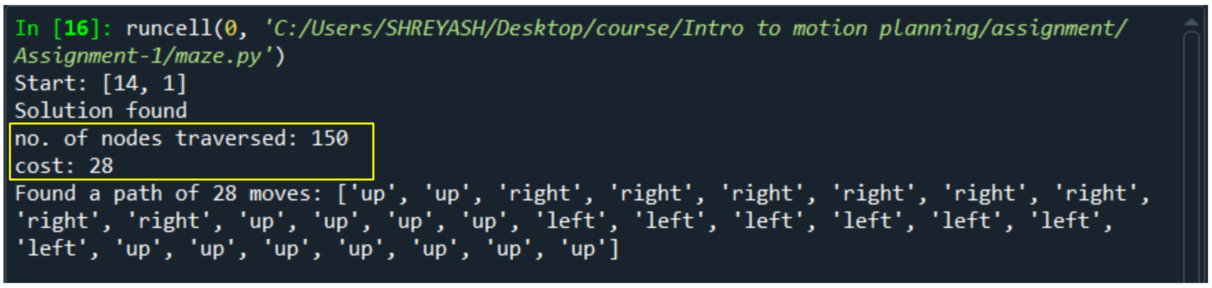
For DFS implementation, algorithm searches for a goal state by exploring the deepest nodes in the search tree first. The code maintains a stack (fringe) of nodes to be explored, with each node containing the state, a list of actions taken to reach that state (path), and the cost of those actions (act).

Initially, the start state is pushed onto the fringe. While the fringe is not empty, the topmost node is popped from the **stack**, and its state is marked as visited. If the popped node is the goal state, the list of actions taken to reach that state is returned. Otherwise, the algorithm generates the successors of the current node, and for each unvisited successor, a new node is created and pushed onto the fringe with updated path and cost values.

The algorithm keeps track of the number of nodes traversed (x), and if no solution is found, it returns an empty list. The code also uses a 2D list (visited) to keep track of the visited nodes, and sets its maximum dimensions as 16 and 20 i.e. the size of maze.

B. Breadth first search



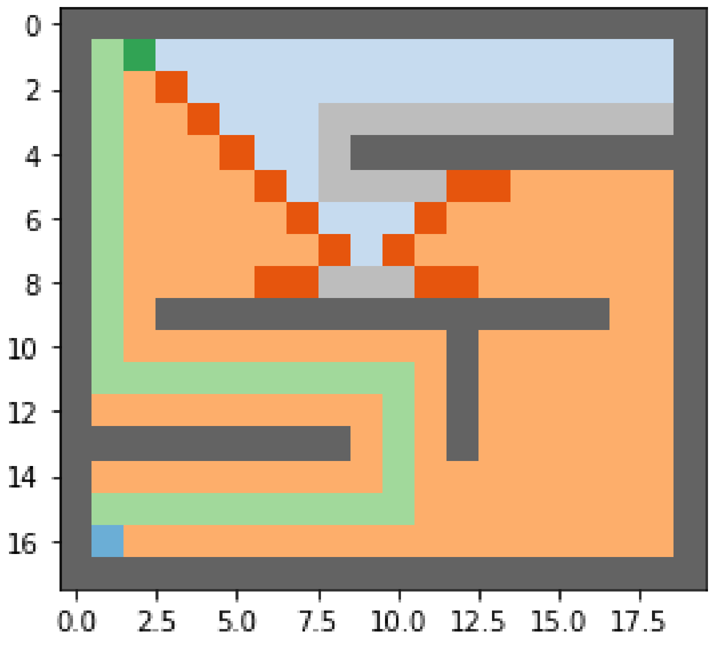


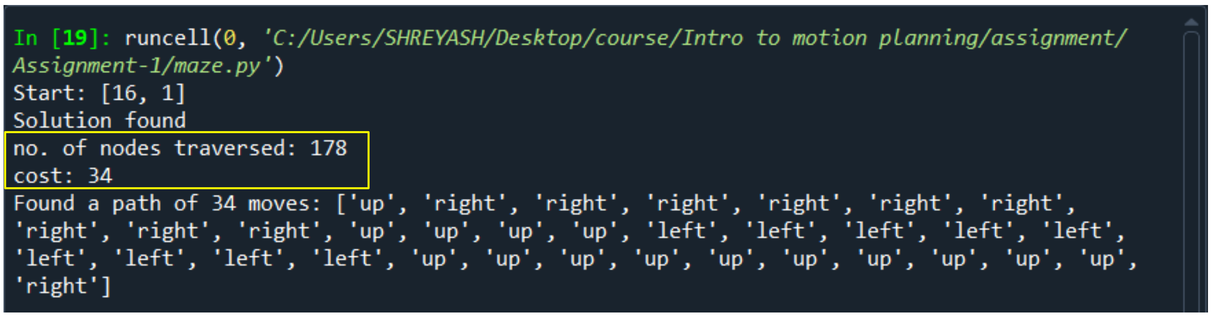
```
In [16]: runcell(0, 'C:/Users/SHREYASH/Desktop/course/Intro to motion planning/assignment/
Assignment-1/maze.py')
Start: [14, 1]
Solution found
no. of nodes traversed: 150
cost: 28
Found a path of 28 moves: ['up', 'up', 'right', 'right', 'right', 'right', 'right', 'right',
'right', 'right', 'up', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'left', 'left',
'left', 'up', 'up', 'up', 'up', 'up', 'up', 'up']
```

For BFS implementation, algorithm explores all current nodes at the same level before exploring next level. The implementation of the code is same as DFS above but with the difference that we pop initial element in fringe list, thus making fringe data structure a **queue**.

ED22S016
Shreyash Gadgil

Q2. Uniform Cost Search



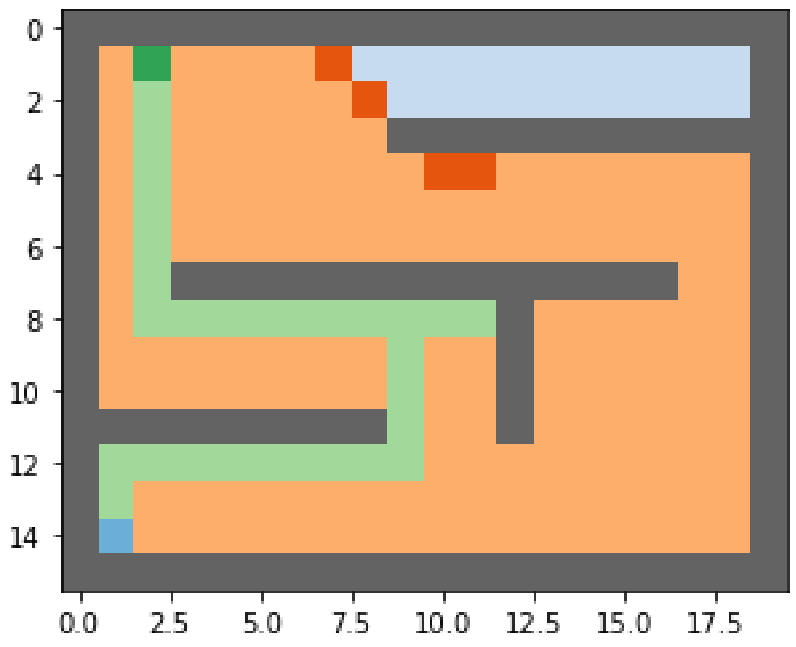


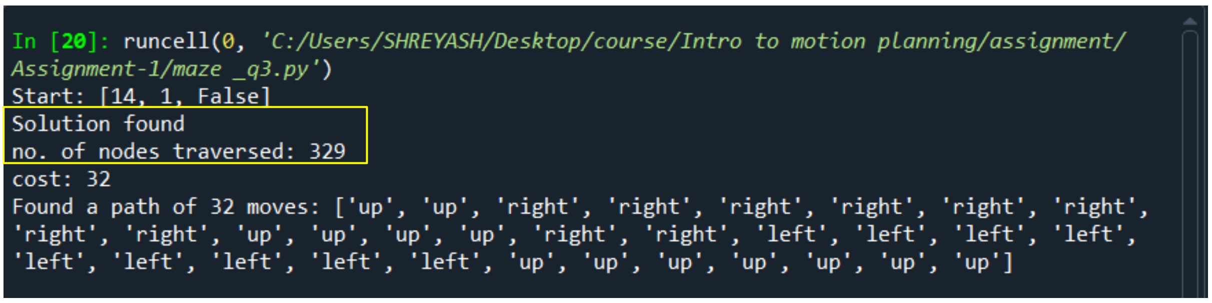
```
In [19]: runcell(0, 'C:/Users/SHREYASH/Desktop/course/Intro to motion planning/assignment/
Assignment-1/maze.py')
Start: [16, 1]
Solution found
no. of nodes traversed: 178
cost: 34
Found a path of 34 moves: ['up', 'right', 'right', 'right', 'right', 'right', 'right',
'right', 'right', 'right', 'up', 'up', 'up', 'up', 'left', 'left', 'left', 'left', 'left',
'left', 'left', 'left', 'left', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up', 'up',
'right']
```

In the Uniform Cost search algorithm the path with lowest weight is given highest priority, generally this is done by a priority queue. But in this implementation, I have added a sorting step for 'fringe element' at the start of 'for' loop, thus creating a priority queue which expands the lowest weighted nodes first.

ED22S016
Shreyash Gadgil

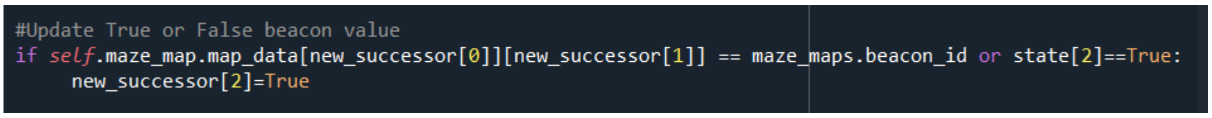
Q3. Go to goal, by passing through beacon





```
In [20]: runcell(0, 'C:/Users/SHREYASH/Desktop/course/Intro to motion planning/assignment/
Assignment-1/maze _q3.py')
Start: [14, 1, False]
Solution found
no. of nodes traversed: 329
cost: 32
Found a path of 32 moves: ['up', 'up', 'right', 'right', 'right', 'right', 'right', 'right',
'right', 'right', 'up', 'up', 'up', 'up', 'right', 'right', 'left', 'left', 'left', 'left',
'left', 'left', 'left', 'left', 'left', 'up', 'up', 'up', 'up', 'up', 'up', 'up']
```

For the maze to go through the beacon, we expand state space function to include a Boolean flag true/false(state=[x,y,False]). When the node traverses through beacon we make the flag true and the successors of the beacon element will also be true. We do this by changing following line in '.getSuccessor()' function.



```
#Update True or False beacon value
if self.maze_map.map_data[new_successor[0]][new_successor[1]] == maze_maps.beacon_id or state[2]==True:
    new_successor[2]=True
```