Search: Solving a Maze Using a Goal-based Agent

Student Name: Vishakha Kishor Satpute

I have used the following AI tools: [list tools]

I understand that my submission needs to be my own work: VS

Instructions

Total Points: Undergrads 100 / Graduate students 110

Complete this notebook. Use the provided notebook cells and insert additional code and markdown cells as needed. Submit the completely rendered notebook as a PDF file.

Introduction

The agent has a map of the maze it is in and the environment is assumed to be **deterministic, discrete, and known.** The agent must use the map to plan a path through the maze from the starting location S to the goal location G. This is a planing exercise for a goal-based agent, so you do not need to implement an environment, just use the map to search for a path. Once the plan is made, the agent in a deterministic environment (i.e., the transition function is deterministic with the outcome of each state/action pair fixed and no randomness) can just follow the path and does not need to care about the percepts. This is also called an **open-loop system.** The execution phase is trivial and we do not implement it in this exercise.

Tree search algorithm implementations that you find online and used in general algorithms courses have often a different aim. These algorithms assume that you already have a tree in memory. We are interested in dynamically creating a search tree with the aim of finding a good/the best path from the root noteto the goal state. Follow the pseudo code presented in the text book (and replicated in the slides) closely. Ideally, we would like to search only a small part of the maze, i.e., create a search tree with as few nodes as possible.

Several mazes for this exercise are stored as text files. Here is the small example maze:

In [3]:
```
with open("small_maze.txt", "r") as f:
    maze_str = f.read()
print(maze_str)
```

```
XXXXXXXXXXXXXXXXXXXXX
X XX        X X      X
X    XXXXXX X XXXXXX X
XXXXXX     S  X      X
X    X XXXXXX XX XXXXX
X XXXX X         X   X
X        XXX XXX   X X
XXXXXXXXXX    XXXXXX X
```

```
XG         XX         X
XXXXXXXXXXXXXXXXXXXXX
```

Note: The mazes above contains cycles and therefore the state space may not form proper trees unless cycles are prevented. Therfore, you will need to deal with cycle detection in your code.

Parsing and pretty printing the maze

The maze can also be displayed in color using code in the module maze_helper.py. The code parses the string representing the maze and converts it into a `numpy` 2d array which you can use in your implementation. Position are represented as a 2-tuple of the form `(row, col)`.

In [3]:

```
"""Code for the Maze Assignment by Michael Hahsler
Usage:
    import maze_helper as mh
    mh.show_some_mazes()
"""

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import colors

def parse_maze(maze_str):
    """Convert a maze as a string into a 2d numpy array"""
    maze = maze_str.split('\n')
    maze = np.array([[tile for tile in row] for row in maze if len(row) > 0])

    return maze

# show_maze: Code by Nicholas Crothers modified and expanded by M. Hahsler
# This is modified code I found on StackOverflow, at this link
# https://stackoverflow.com/questions/43971138/python-plotting-colored-grid-based-on
def show_maze(maze, fontsize = 10):
    """Display a (parsed) maze as an image."""

    cmap = colors.ListedColormap(['white', 'black', 'blue', 'green', 'red', 'gray',

    # make a deep copy first so the original maze is not changed. Python passes obje
    maze = np.copy(maze)

    goal = find_pos(maze, 'G')
    start = find_pos(maze, 'S')

    # Converts all tile types to integers
    maze[maze == ' '] = 0
    maze[maze == 'X'] = 1 # wall
    maze[maze == 'S'] = 2 # start
    maze[maze == 'G'] = 3 # goal
    maze[maze == 'P'] = 4 # position/final path
    maze[maze == '.'] = 5 # explored squares
    maze[maze == 'F'] = 6 # frontier
    maze = maze.astype(int)

    fig, ax = plt.subplots()
    ax.imshow(maze, cmap = cmap, norm = colors.BoundaryNorm(list(range(cmap.N + 1)),

    plt.text(start[1], start[0], "S", fontsize = fontsize, color = "white",
                horizontalalignment = 'center',
                verticalalignment = 'center')

    plt.text(goal[1], goal[0], "G", fontsize = fontsize, color = "white",
```

```
                horizontalalignment = 'center',
                verticalalignment = 'center')

    plt.show()

def find_pos(maze, what = "S"):
    """
    Find start/goal in a maze and returns the first one.
    Caution: there is no error checking!

    Parameters:
    maze: a array with characters prodced by parse_maze()
    what: the letter to be found ('S' for start and 'G' for goal)

    Returns:
    a tupple (x, y) for the found position.
    """

    # where returns two arrays with all found positions. We are only interested in t
    pos = np.where(maze == what)
    return(tuple([pos[0][0], pos[1][0]]))

def look(maze, pos):
    """Look at the label of a square with the position as an array of the form (x, y

    x, y = pos
    return(maze[x, y])



def welcome():

    with open(f"small_maze.txt", "r") as f:
        maze_str = f.read()
        print(maze_str)

        maze = parse_maze(maze_str)
        goal = find_pos(maze, what = "G")
        print(f"The goal is at {goal}.")

if __name__ == "__main__":
    welcome()
```

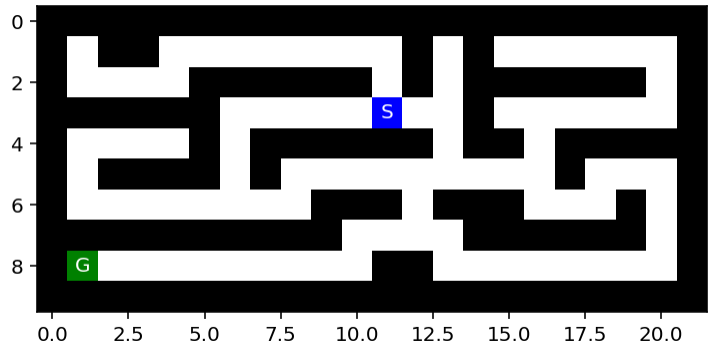
```
XXXXXXXXXXXXXXXXXXXXX
X XX        X X      X
X    XXXXXX X XXXXXX X
XXXXXX     S  X      X
X    X XXXXXX XX XXXXX
X XXXX X         X   X
X        XXX XXX   X X
XXXXXXXXXX    XXXXXX X
XG         XX        X
XXXXXXXXXXXXXXXXXXXXX
The goal is at (8, 1).
```

A helper function to visualize the maze is also available.

In [4]:

```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
# use higher resolution images in notebook
maze = parse_maze(maze_str)
print("Position(0,0):", maze[0, 0])
show_maze(maze)
```

```
Position(0,0): X
```



Find the position of the start and the goal using the helper function `find_pos()`

In [5]:
```
print("Start location:", find_pos(maze, what = "S"))
print("Goal location:", find_pos(maze, what = "G"))
```

```
Start location: (3, 11)
Goal location: (8, 1)
```

Helper function documentation.

Tree structure

Here is an implementation of the basic node structure for the search algorithms (see Fig 3.7 on page 73). I have added a method that extracts the path from the root node to the current node. It can be used to get the path when the search is completed.

In [181…

```
class Node:
    def __init__(self, pos, parent, action, cost,goal=None):
        self.pos = tuple(pos)     # the state; positions are (row,col)
        self.parent = parent      # reference to parent node. None means root node.
        self.action = action      # action used in the transition function (root node
        self.cost = cost          # for uniform cost this is the depth. It is also g(
        self.goal = goal

        if goal:
            def heuristic(nodePosition, goalPosition):
                return abs(nodePosition[0]-goalPosition[0]) + abs(nodePosition[1]-go
            self.distance = heuristic(pos,goal)

    def __lt__(self, other):
        return self.distance < other.distance

    def __str__(self):
        return f"Node - pos = {self.pos}; action = {self.action}; cost = {self.cost}

    def get_path_from_root(self):
        """returns nodes on the path from the root to the current node."""
        node = self
        path = [node]

        while not node.parent is None:
            node = node.parent
            path.append(node)

        path.reverse()

        return(path)
```

If needed, then you can add more fields to the class like the heuristic value $h(n)$ or $f(n)$.

Examples for how to create and use a tree and information on memory management can be found here.

Tasks

The goal is to:

1. Implement the following search algorithms for solving different mazes:
    - Breadth-first search (BFS)
    - Depth-first search (DFS)
    - Greedy best-first search (GBFS)
    - A* search
2. Run each of the above algorithms on the
    - small maze,
    - medium maze,
    - large maze,
    - open maze,
    - wall maze,
    - loops maze,

- empty maze, and
- empty 2_maze.

3. For each problem instance and each search algorithm, report the following in a table:
    - The solution and its path cost
    - Total number of nodes expanded
    - Maximum tree depth
    - Maximum size of the frontier
4. Display each solution by marking every maze square (or state) visited and the squares on the final path.

General [10 Points]

1. Make sure that you use the latest version of this notebook. Sync your forked repository and pull the latest revision.
2. Your implementation can use libraries like math, numpy, scipy, but not libraries that implement inteligent agents or complete search algorithms. Try to keep the code simple! In this course, we want to learn about the algorithms and we often do not need to use object-oriented design.
3. You notebook needs to be formated professionally.
    - Add additional markdown blocks for your description, comments in the code, add tables and use mathplotlib to produce charts where appropriate
    - Do not show debugging output or include an excessive amount of output.
    - Check that your PDF file is readable. For example, long lines are cut off in the PDF file. You don't have control over page breaks, so do not worry about these.
4. Document your code. Add a short discussion of how your implementation works and your design choices.

Task 1: Defining the search problem and determining the problem size [10 Points]

Define the components of the search problem:

- Initial state
- Actions
- Transition model
- Goal state
- Path cost

Use verbal descriptions, variables and equations as appropriate.

Note: You can swich the next block from code to Markdown and use formating.

Initial State: A state from which the search algorithm starts the search. For example: in the given small maze the 'S' is at position (3,11) which is the intial state.

Actions: Actions are the set of moves that an algorithm can perform. Example: north, south, east, west

Transition model: Transition model is the output of performing the given actions on the state

Goal State: Goal state is the target state that a search algorithm is trying to reach.

Path Cost: The path cost is the cost required for transition from one state to another

Give some estimates for the problem size:

- n: state space size
- d: depth of the optimal solution
- m: maximum depth of tree
- b: maximum branching factor

Describe how you would determin these values for a given maze.

State space size: If the maze is of size (10,20), the state space size for that problem will be 10 x 20 = 200

Depth of the optimal solution: The depth of the optimal solution for the given small maze is 13.

Maximum depth of tree: In worst case, the maximum depth of tree could be 200.

Maximum branching factor:The maximum branching factor for the given maze will be 4 as there are 4 actions that could be performed(East, West, North, South).

Task 2: Uninformed search: Breadth-first and depth-first [40 Points]

Implement these search strategies. Follow the pseudocode in the textbook/slides. You can use the tree structure shown above to extract the final path from your solution.

Notes:

- You can find maze solving implementations online that use the map to store information. While this is an effective idea for this two-dimensional navigation problem, it typically cannot be used for other search problems. Therefore, follow the textbook and only store information in the tree created during search, and use the `reached` and `frontier` data structures.
- DSF can be implemented using the BFS tree search algorithm and simply changing the order in which the frontier is expanded (this is equivalent to best-first search with path length as the criterion to expand the next node). However, to take advantage of the significantly smaller memory footprint of DFS, you need to implement DFS in a different way without a `reached` data structure and by releasing the memory for nodes that are not needed anymore.
- If DFS does not use a `reached` data structure, then its cycle checking abilities are limited. Remember, that DSF is incomplete if cycles cannot be prevented. You will see in your experiments that open spaces are a problem.

BFS

In [184…

```
from queue import Queue
```

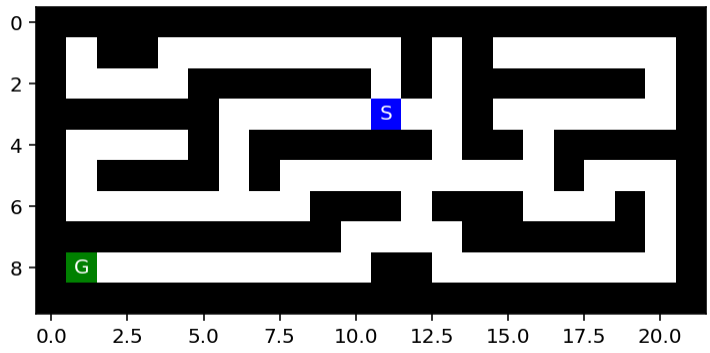
```
d = []
currentMaze =""
mazeName = ["small_maze.txt","medium_maze.txt","large_maze.txt","loops_maze.txt","wa

def bfs(currMazr,maze,startPos,endPos,action,root):
    explored = set() # Set to keep track of explored positions
    frontier = Queue() # Queue to store nodes to be explored
    frontier.put(root)

    count,size,depth,bf = 0,1,0,0

    # Loop until the frontier is empty
    while not frontier.empty():
        curr = frontier.get()  # Get the current node from the frontier
        currPos = curr.pos

        # Update depth if the current node's cost is greater
        if curr.cost > depth :
            depth = curr.cost

        # Check if the current position is the goal position
        if currPos == endPos :
            print('Depth of the tree: ', depth)
            print('Cost for the Path: ', curr.cost)
            print('Tree size: ', size)
            print('Branching Factor: ', count)
            d.append([currMazr,'BFS',frontier.qsize(),depth, curr.cost, size, count]
            return curr

        # Add the current position to the set of explored positions
        explored.add(currPos)

        # Iterate over possible actions
        for a in action.values():
            branchingFactor=0
            new = (currPos[0]+a[0],currPos[1]+a[1])
            if look(maze,new) != "X" and new not in explored:
                child = Node(new, curr, a, curr.cost + 1)
                if(currPos !=startPos and currPos!=endPos and new != endPos):
                    maze[new[0]][new[1]] = "F"
                    maze[currPos[0]][currPos[1]] = "."
                size += 1
                branchingFactor+=1
                frontier.put(child)
            if bf < branchingFactor:
                bf =branchingFactor
        count = count + 1

    return None
```

In [189…

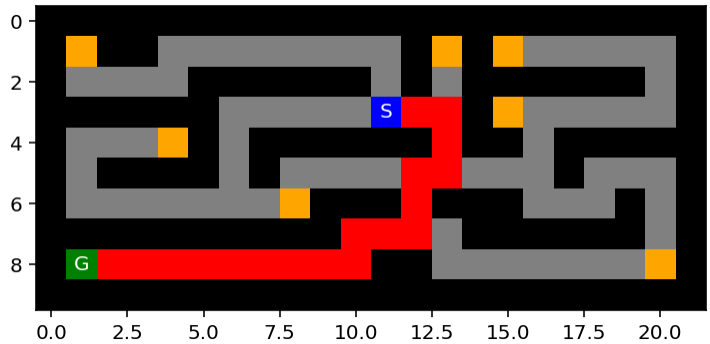
```
# Loop through each maze file
for f in mazeName:
    print(f)
    with open(f, "r") as f:
        maze_str = f.read()
    maze = parse_maze(maze_str)
    show_maze(maze)
    start = []
    goal = []
    start = find_pos(maze)
    goal = find_pos(maze, what = "G")
    actions = {"north":(-1,0),"south":(1,0),"east":(0,1),"west":(0,-1)}
    rootNode = Node(start,None, None, 0)
```

```
    finalNode = bfs(f,maze,start,goal,actions,rootNode)
    if finalNode:
        path = finalNode.get_path_from_root() # Extract and display the path from st
        print("Path:")
        for node in path:
            print(node.pos)
            if(node.pos !=start and node.pos!=goal): # Update the maze to mark the p
                maze[node.pos[0]][node.pos[1]] = "P"
        print("Cost:",finalNode.cost)
    else:
        print("Goal not found")
    show_maze(maze)
```

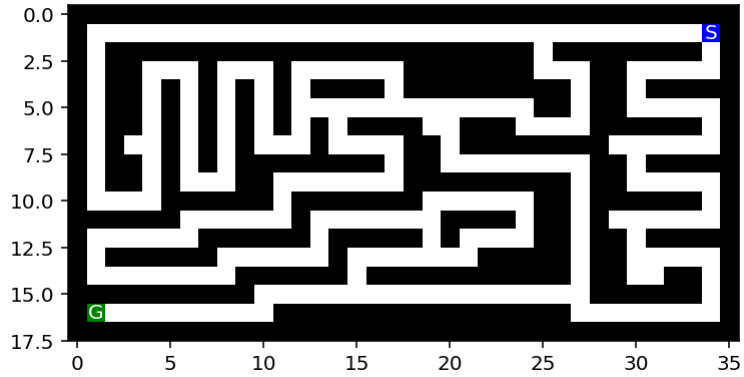
small_maze.txt




```
Depth of the tree:  19
Cost for the Path:  19
Tree size:  96
Branching Factor:  94
Path:
(3, 11)
(3, 12)
(3, 13)
(4, 13)
(5, 13)
(5, 12)
(6, 12)
(7, 12)
(7, 11)
(7, 10)
(8, 10)
(8, 9)
(8, 8)
(8, 7)
(8, 6)
(8, 5)
(8, 4)
(8, 3)
(8, 2)
(8, 1)
Cost: 19
```

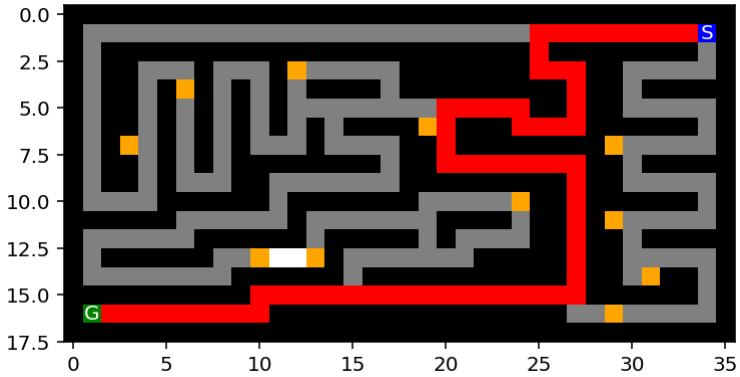


medium_maze.txt

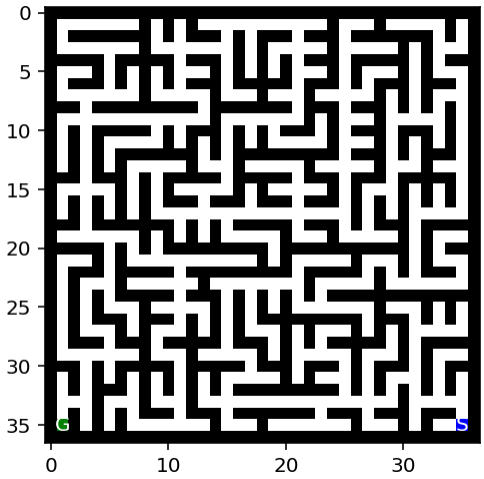



```
Depth of the tree:  68
Cost for the Path:  68
Tree size:  278
Branching Factor:  275
Path:
(1, 34)
(1, 33)
(1, 32)
(1, 31)
(1, 30)
(1, 29)
(1, 28)
(1, 27)
(1, 26)
(1, 25)
(2, 25)
(3, 25)
(3, 26)
(3, 27)
(4, 27)
(5, 27)
(6, 27)
(6, 26)
(6, 25)
(6, 24)
(5, 24)
(5, 23)
(5, 22)
(5, 21)
(5, 20)
(6, 20)
(7, 20)
(8, 20)
(8, 21)
(8, 22)
(8, 23)
(8, 24)
(8, 25)
(8, 26)
(8, 27)
(9, 27)
(10, 27)
(11, 27)
(12, 27)
(13, 27)
(14, 27)
(15, 27)
(15, 26)
(15, 25)
(15, 24)
(15, 23)
(15, 22)
```

```
(15, 21)
(15, 20)
(15, 19)
(15, 18)
(15, 17)
(15, 16)
(15, 15)
(15, 14)
(15, 13)
(15, 12)
(15, 11)
(15, 10)
(16, 10)
(16, 9)
(16, 8)
(16, 7)
(16, 6)
(16, 5)
(16, 4)
(16, 3)
(16, 2)
(16, 1)
Cost: 68
```




```
large_maze.txt
```



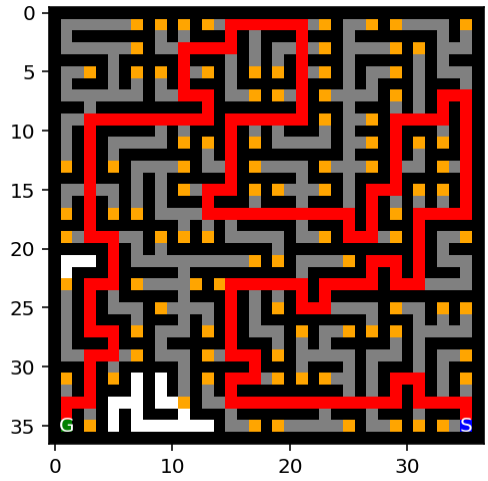

```
Depth of the tree:  210
Cost for the Path:  210
Tree size:  623
Branching Factor:  620
Path:
(35, 35)
(34, 35)
(33, 35)
(33, 34)
(33, 33)
(33, 32)
(33, 31)
```

(32, 31)
(31, 31)
(31, 30)
(31, 29)
(32, 29)
(33, 29)
(33, 28)
(33, 27)
(33, 26)
(33, 25)
(33, 24)
(33, 23)
(33, 22)
(33, 21)
(33, 20)
(33, 19)
(33, 18)
(33, 17)
(33, 16)
(33, 15)
(32, 15)
(31, 15)
(31, 16)
(31, 17)
(30, 17)
(29, 17)
(29, 16)
(29, 15)
(28, 15)
(27, 15)
(26, 15)
(25, 15)
(24, 15)
(23, 15)
(23, 16)
(23, 17)
(23, 18)
(23, 19)
(23, 20)
(23, 21)
(24, 21)
(25, 21)
(25, 22)
(25, 23)
(24, 23)
(23, 23)
(23, 24)
(23, 25)
(23, 26)
(23, 27)
(22, 27)
(21, 27)
(21, 28)
(21, 29)
(22, 29)
(23, 29)
(23, 30)
(23, 31)
(22, 31)
(21, 31)
(20, 31)
(19, 31)
(18, 31)
(17, 31)

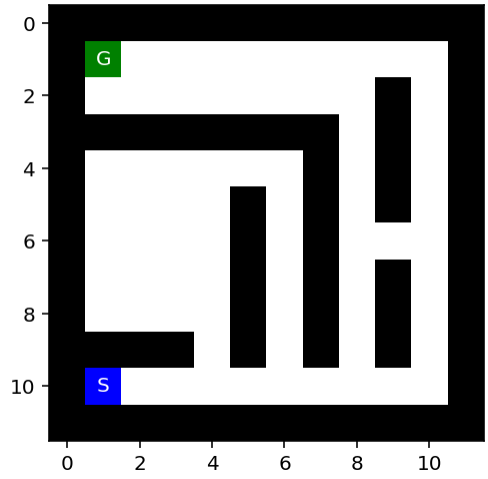
(17, 32)
(17, 33)
(17, 34)
(17, 35)
(16, 35)
(15, 35)
(14, 35)
(13, 35)
(12, 35)
(11, 35)
(10, 35)
(9, 35)
(8, 35)
(7, 35)
(7, 34)
(7, 33)
(8, 33)
(9, 33)
(9, 32)
(9, 31)
(9, 30)
(9, 29)
(10, 29)
(11, 29)
(12, 29)
(13, 29)
(14, 29)
(15, 29)
(15, 28)
(15, 27)
(16, 27)
(17, 27)
(18, 27)
(19, 27)
(19, 26)
(19, 25)
(18, 25)
(17, 25)
(17, 24)
(17, 23)
(17, 22)
(17, 21)
(17, 20)
(17, 19)
(17, 18)
(17, 17)
(17, 16)
(17, 15)
(17, 14)
(17, 13)
(16, 13)
(15, 13)
(15, 14)
(15, 15)
(14, 15)
(13, 15)
(12, 15)
(11, 15)
(10, 15)
(9, 15)
(9, 16)
(9, 17)
(9, 18)
(9, 19)

(9, 20)
(9, 21)
(8, 21)
(7, 21)
(6, 21)
(5, 21)
(4, 21)
(3, 21)
(2, 21)
(1, 21)
(1, 20)
(1, 19)
(1, 18)
(1, 17)
(1, 16)
(1, 15)
(2, 15)
(3, 15)
(3, 14)
(3, 13)
(3, 12)
(3, 11)
(4, 11)
(5, 11)
(6, 11)
(7, 11)
(7, 12)
(7, 13)
(8, 13)
(9, 13)
(9, 12)
(9, 11)
(9, 10)
(9, 9)
(9, 8)
(9, 7)
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(10, 3)
(11, 3)
(12, 3)
(13, 3)
(14, 3)
(15, 3)
(16, 3)
(17, 3)
(18, 3)
(19, 3)
(19, 4)
(19, 5)
(20, 5)
(21, 5)
(22, 5)
(23, 5)
(23, 4)
(23, 3)
(24, 3)
(25, 3)
(26, 3)
(27, 3)
(27, 4)
(27, 5)

```
(28, 5)
(29, 5)
(29, 4)
(29, 3)
(30, 3)
(31, 3)
(32, 3)
(33, 3)
(33, 2)
(33, 1)
(34, 1)
(35, 1)
Cost: 210
```

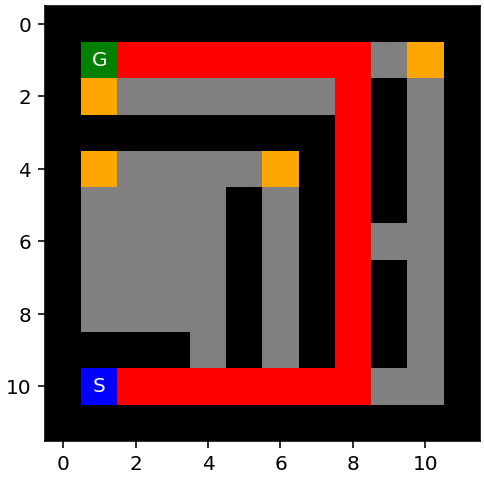


loops_maze.txt

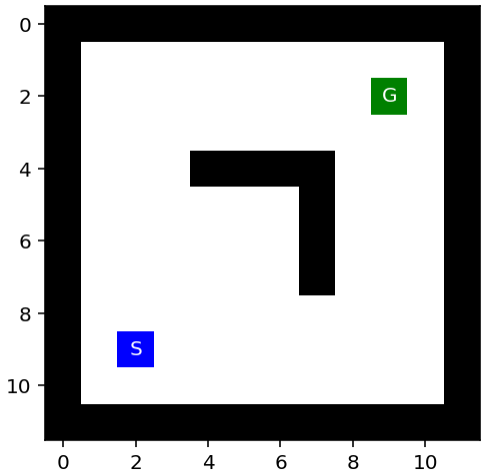



```
Depth of the tree:  23
Cost for the Path:  23
Tree size:  213
Branching Factor:  205
Path:
(10, 1)
(10, 2)
(10, 3)
(10, 4)
(10, 5)
(10, 6)
(10, 7)
(10, 8)
(9, 8)
(8, 8)
(7, 8)
(6, 8)
(5, 8)
(4, 8)
```

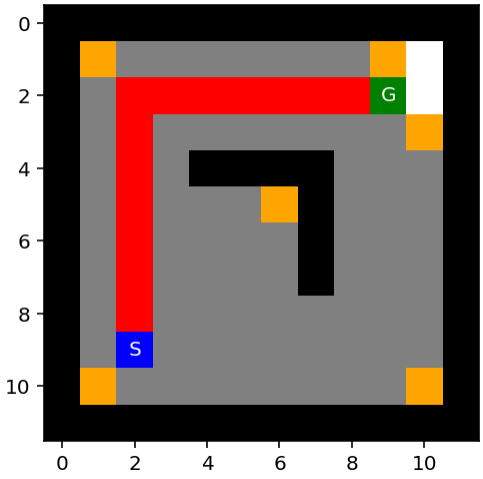
```
(3, 8)
(2, 8)
(1, 8)
(1, 7)
(1, 6)
(1, 5)
(1, 4)
(1, 3)
(1, 2)
(1, 1)
Cost: 23
```



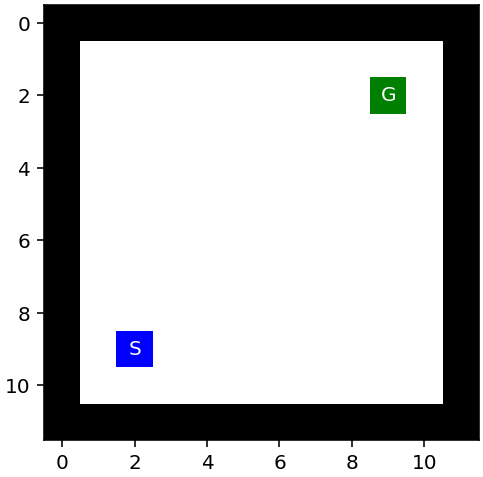
wall_maze.txt



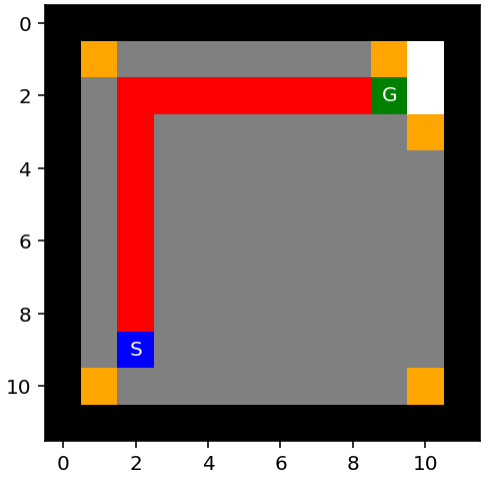

```
Depth of the tree:  14
Cost for the Path:  14
Tree size:  1726
Branching Factor:  1304
Path:
(9, 2)
(8, 2)
(7, 2)
(6, 2)
(5, 2)
(4, 2)
(3, 2)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(2, 7)
(2, 8)
(2, 9)
Cost: 14
```



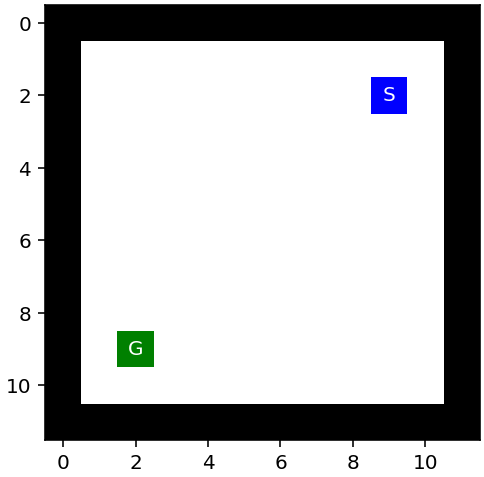
empty_maze.txt



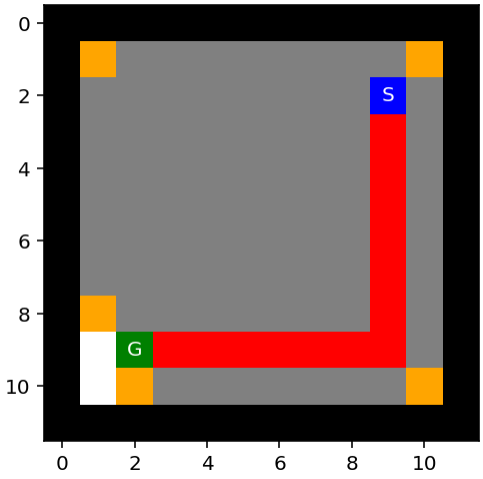

```
Depth of the tree:  14
Cost for the Path:  14
Tree size:  22978
Branching Factor:  13540
Path:
(9, 2)
(8, 2)
(7, 2)
(6, 2)
(5, 2)
(4, 2)
(3, 2)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(2, 7)
(2, 8)
(2, 9)
Cost: 14
```



empty_2_maze.txt




```
Depth of the tree:  14
Cost for the Path:  14
Tree size:  22978
Branching Factor:  13540
Path:
(2, 9)
(3, 9)
(4, 9)
(5, 9)
(6, 9)
(7, 9)
(8, 9)
(9, 9)
(9, 8)
(9, 7)
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(9, 2)
Cost: 14
```



DFS

In [186…

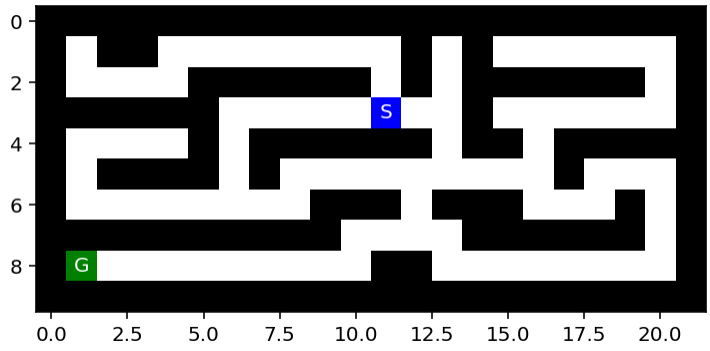
```
def dfs(currMaze,maze, start, end, actions, root):
    explored = set() # Set to keep track of explored positions
    frontier = [root] # LIFO strategy for depth-first search
    count,size,depth,bf = 0,1,0,0
    frontierSize = len(frontier)

    while frontier: # Loop until the goal is reached
        if frontierSize < len(frontier):
            frontierSize = len(frontier)

        currNode = frontier.pop()
        currPos = currNode.pos

        if currNode.cost > depth:
            depth = currNode.cost

        if currPos == end: # Goal reached
            print('Maximum Depth of the tree: ', depth)
            print('Cost for the Path: ', currNode.cost)
            print('Maximum tree size: ', size)
            print('Maximum Branching Factor: ', count)
            d.append([currMaze,'DFS',frontierSize,depth, currNode.cost, size, count])
            return currNode

        explored.add(currPos)

        for a in actions.values():
            branchingFactor = 0
            new = (currPos[0] + a[0], currPos[1] + a[1])

            if look(maze, new) != "X" and new not in explored:
                childNode = Node(new, currNode, a, currNode.cost + 1)

                if currPos != start and currPos != end and new != end:
                    maze[new[0]][new[1]] = "F"
                    maze[currPos[0]][currPos[1]] = "."
                size += 1
                branchingFactor += 1
                frontier.append(childNode)

            if bf < branchingFactor:
                bf = branchingFactor
```

```
        count += 1
```

In [188…

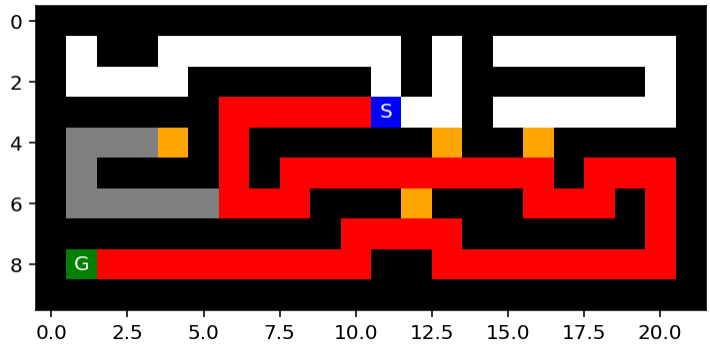
```
for f in mazeName:
    print(f)
    with open(f, "r") as f:
        maze_str = f.read()
    maze = parse_maze(maze_str)
    show_maze(maze)
    start = []
    goal = []
    start = find_pos(maze)
    goal = find_pos(maze, what = "G")
    actions = {"north":(-1,0),"south":(1,0),"east":(0,1),"west":(0,-1)}
    rootNode = Node(start,None, None, 0)
    finalNode = dfs(f,maze,start,goal,actions,rootNode)
    if finalNode:
        path = finalNode.get_path_from_root()
        print("Path:")
        for node in path:
            print(node.pos)
            if(node.pos !=start and node.pos!=goal):
                maze[node.pos[0]][node.pos[1]] = "P"
        print("Cost:",finalNode.cost)
        show_maze(maze)
    else:
        print("Goal not found")
    #show_maze(maze)
```

small_maze.txt

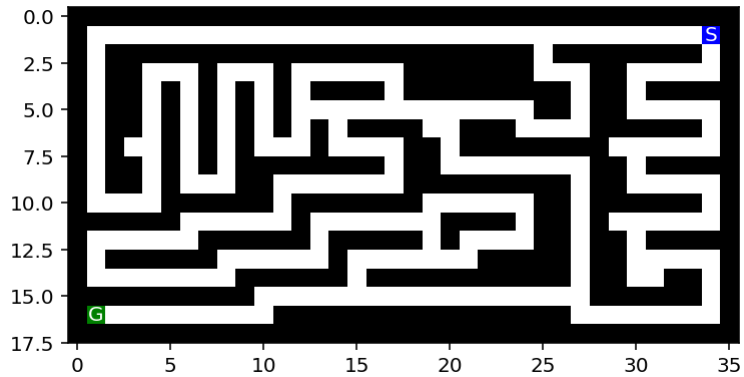



```
Maximum Depth of the tree:  49
Cost for the Path:  49
Maximum tree size:  66
Maximum Branching Factor:  59
Path:
(3, 11)
(3, 10)
(3, 9)
(3, 8)
(3, 7)
(3, 6)
(4, 6)
(5, 6)
(6, 6)
(6, 7)
(6, 8)
(5, 8)
(5, 9)
(5, 10)
(5, 11)
(5, 12)
```

```
(5, 13)
(5, 14)
(5, 15)
(5, 16)
(6, 16)
(6, 17)
(6, 18)
(5, 18)
(5, 19)
(5, 20)
(6, 20)
(7, 20)
(8, 20)
(8, 19)
(8, 18)
(8, 17)
(8, 16)
(8, 15)
(8, 14)
(8, 13)
(7, 13)
(7, 12)
(7, 11)
(7, 10)
(8, 10)
(8, 9)
(8, 8)
(8, 7)
(8, 6)
(8, 5)
(8, 4)
(8, 3)
(8, 2)
(8, 1)
Cost: 49
```




```
medium_maze.txt
```

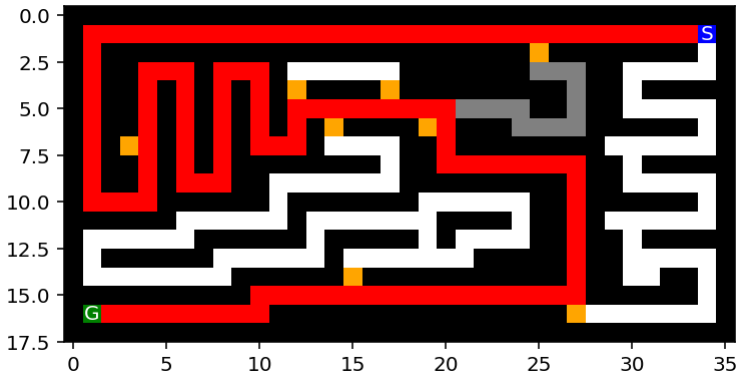



```
Maximum Depth of the tree:  130
Cost for the Path:  130
Maximum tree size:  155
Maximum Branching Factor:  146
```

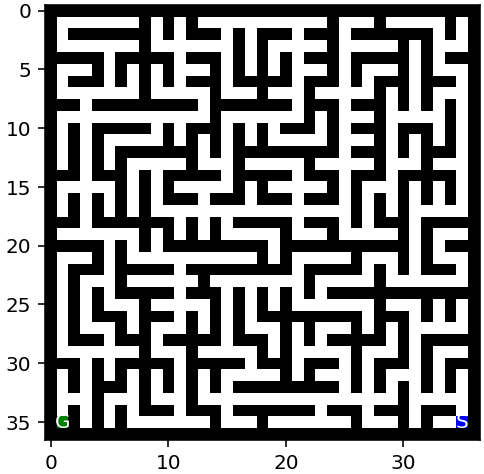
```
Path:
(1, 34)
(1, 33)
(1, 32)
(1, 31)
(1, 30)
(1, 29)
(1, 28)
(1, 27)
(1, 26)
(1, 25)
(1, 24)
(1, 23)
(1, 22)
(1, 21)
(1, 20)
(1, 19)
(1, 18)
(1, 17)
(1, 16)
(1, 15)
(1, 14)
(1, 13)
(1, 12)
(1, 11)
(1, 10)
(1, 9)
(1, 8)
(1, 7)
(1, 6)
(1, 5)
(1, 4)
(1, 3)
(1, 2)
(1, 1)
(2, 1)
(3, 1)
(4, 1)
(5, 1)
(6, 1)
(7, 1)
(8, 1)
(9, 1)
(10, 1)
(10, 2)
(10, 3)
(10, 4)
(9, 4)
(8, 4)
(7, 4)
(6, 4)
(5, 4)
(4, 4)
(3, 4)
(3, 5)
(3, 6)
(4, 6)
(5, 6)
(6, 6)
(7, 6)
(8, 6)
(9, 6)
(9, 7)
(9, 8)
```

(8, 8)
(7, 8)
(6, 8)
(5, 8)
(4, 8)
(3, 8)
(3, 9)
(3, 10)
(4, 10)
(5, 10)
(6, 10)
(7, 10)
(7, 11)
(7, 12)
(6, 12)
(5, 12)
(5, 13)
(5, 14)
(5, 15)
(5, 16)
(5, 17)
(5, 18)
(5, 19)
(5, 20)
(6, 20)
(7, 20)
(8, 20)
(8, 21)
(8, 22)
(8, 23)
(8, 24)
(8, 25)
(8, 26)
(8, 27)
(9, 27)
(10, 27)
(11, 27)
(12, 27)
(13, 27)
(14, 27)
(15, 27)
(15, 26)
(15, 25)
(15, 24)
(15, 23)
(15, 22)
(15, 21)
(15, 20)
(15, 19)
(15, 18)
(15, 17)
(15, 16)
(15, 15)
(15, 14)
(15, 13)
(15, 12)
(15, 11)
(15, 10)
(16, 10)
(16, 9)
(16, 8)
(16, 7)
(16, 6)
(16, 5)

```
(16, 4)
(16, 3)
(16, 2)
(16, 1)
Cost: 130
```



large_maze.txt

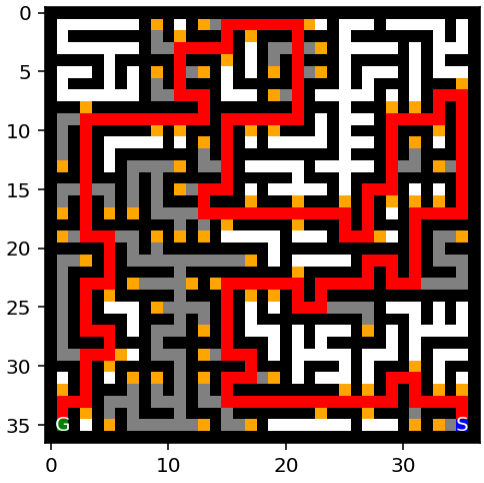



```
Maximum Depth of the tree:  222
Cost for the Path:  210
Maximum tree size:  428
Maximum Branching Factor:  390
Path:
(35, 35)
(34, 35)
(33, 35)
(33, 34)
(33, 33)
(33, 32)
(33, 31)
(32, 31)
(31, 31)
(31, 30)
(31, 29)
(32, 29)
(33, 29)
(33, 28)
(33, 27)
(33, 26)
(33, 25)
(33, 24)
(33, 23)
(33, 22)
(33, 21)
(33, 20)
(33, 19)
(33, 18)
(33, 17)
```

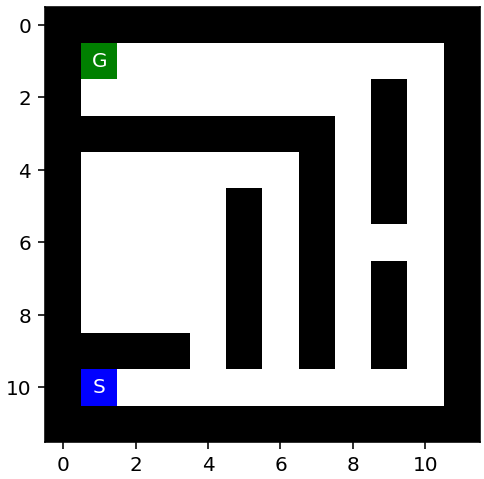
(33, 16)
(33, 15)
(32, 15)
(31, 15)
(31, 16)
(31, 17)
(30, 17)
(29, 17)
(29, 16)
(29, 15)
(28, 15)
(27, 15)
(26, 15)
(25, 15)
(24, 15)
(23, 15)
(23, 16)
(23, 17)
(23, 18)
(23, 19)
(23, 20)
(23, 21)
(24, 21)
(25, 21)
(25, 22)
(25, 23)
(24, 23)
(23, 23)
(23, 24)
(23, 25)
(23, 26)
(23, 27)
(22, 27)
(21, 27)
(21, 28)
(21, 29)
(22, 29)
(23, 29)
(23, 30)
(23, 31)
(22, 31)
(21, 31)
(20, 31)
(19, 31)
(18, 31)
(17, 31)
(17, 32)
(17, 33)
(17, 34)
(17, 35)
(16, 35)
(15, 35)
(14, 35)
(13, 35)
(12, 35)
(11, 35)
(10, 35)
(9, 35)
(8, 35)
(7, 35)
(7, 34)
(7, 33)
(8, 33)
(9, 33)

(9, 32)
(9, 31)
(9, 30)
(9, 29)
(10, 29)
(11, 29)
(12, 29)
(13, 29)
(14, 29)
(15, 29)
(15, 28)
(15, 27)
(16, 27)
(17, 27)
(18, 27)
(19, 27)
(19, 26)
(19, 25)
(18, 25)
(17, 25)
(17, 24)
(17, 23)
(17, 22)
(17, 21)
(17, 20)
(17, 19)
(17, 18)
(17, 17)
(17, 16)
(17, 15)
(17, 14)
(17, 13)
(16, 13)
(15, 13)
(15, 14)
(15, 15)
(14, 15)
(13, 15)
(12, 15)
(11, 15)
(10, 15)
(9, 15)
(9, 16)
(9, 17)
(9, 18)
(9, 19)
(9, 20)
(9, 21)
(8, 21)
(7, 21)
(6, 21)
(5, 21)
(4, 21)
(3, 21)
(2, 21)
(1, 21)
(1, 20)
(1, 19)
(1, 18)
(1, 17)
(1, 16)
(1, 15)
(2, 15)
(3, 15)

```
(3, 14)
(3, 13)
(3, 12)
(3, 11)
(4, 11)
(5, 11)
(6, 11)
(7, 11)
(7, 12)
(7, 13)
(8, 13)
(9, 13)
(9, 12)
(9, 11)
(9, 10)
(9, 9)
(9, 8)
(9, 7)
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(10, 3)
(11, 3)
(12, 3)
(13, 3)
(14, 3)
(15, 3)
(16, 3)
(17, 3)
(18, 3)
(19, 3)
(19, 4)
(19, 5)
(20, 5)
(21, 5)
(22, 5)
(23, 5)
(23, 4)
(23, 3)
(24, 3)
(25, 3)
(26, 3)
(27, 3)
(27, 4)
(27, 5)
(28, 5)
(29, 5)
(29, 4)
(29, 3)
(30, 3)
(31, 3)
(32, 3)
(33, 3)
(33, 2)
(33, 1)
(34, 1)
(35, 1)
Cost: 210
```

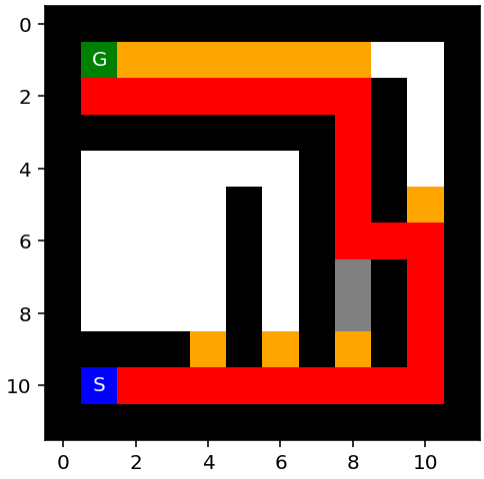


loops_maze.txt

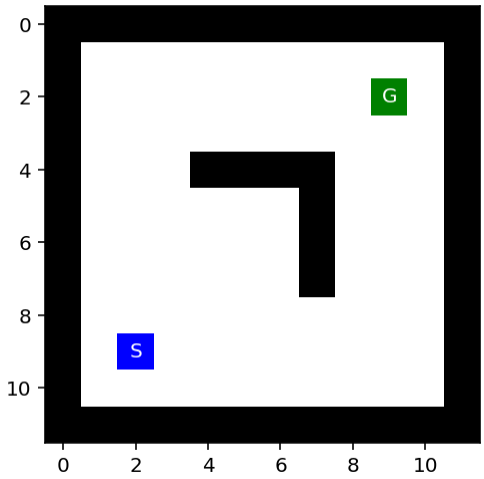


Maximum Depth of the tree: 27
Cost for the Path: 27
Maximum tree size: 42
Maximum Branching Factor: 30
Path:
(10, 1)
(10, 2)
(10, 3)
(10, 4)
(10, 5)
(10, 6)
(10, 7)
(10, 8)
(10, 9)
(10, 10)
(9, 10)
(8, 10)
(7, 10)
(6, 10)
(6, 9)
(6, 8)
(5, 8)
(4, 8)
(3, 8)
(2, 8)
(2, 7)
(2, 6)
(2, 5)
(2, 4)
(2, 3)
(2, 2)
(2, 1)

(1, 1)
Cost: 27

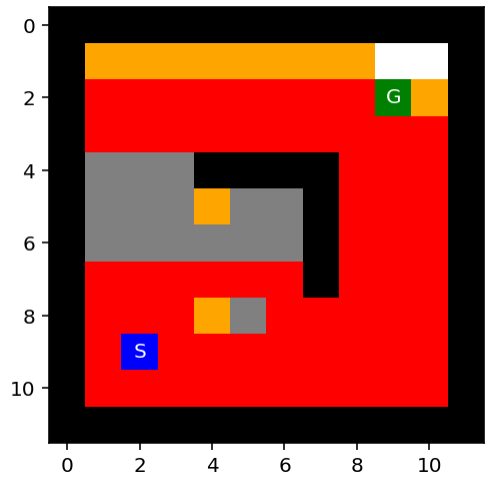


wall_maze.txt

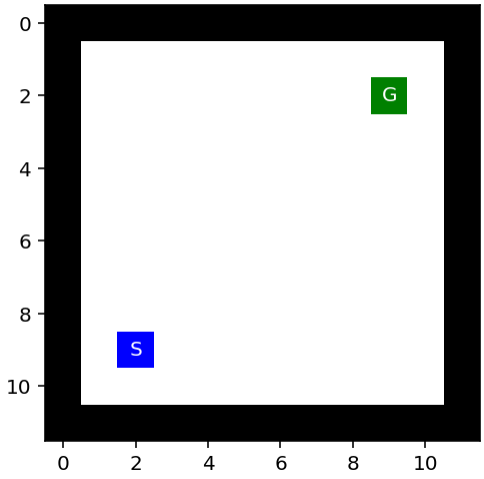


Maximum Depth of the tree: 70
Cost for the Path: 64
Maximum tree size: 147
Maximum Branching Factor: 88
Path:
(9, 2)
(9, 1)
(10, 1)
(10, 2)
(10, 3)
(10, 4)
(10, 5)
(10, 6)
(10, 7)
(10, 8)
(10, 9)
(10, 10)
(9, 10)
(9, 9)
(9, 8)
(9, 7)
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(8, 3)
(8, 2)
(8, 1)
(7, 1)
(7, 2)

```
(7, 3)
(7, 4)
(7, 5)
(7, 6)
(8, 6)
(8, 7)
(8, 8)
(8, 9)
(8, 10)
(7, 10)
(7, 9)
(7, 8)
(6, 8)
(6, 9)
(6, 10)
(5, 10)
(5, 9)
(5, 8)
(4, 8)
(4, 9)
(4, 10)
(3, 10)
(3, 9)
(3, 8)
(3, 7)
(3, 6)
(3, 5)
(3, 4)
(3, 3)
(3, 2)
(3, 1)
(2, 1)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(2, 7)
(2, 8)
(2, 9)
Cost: 64
```

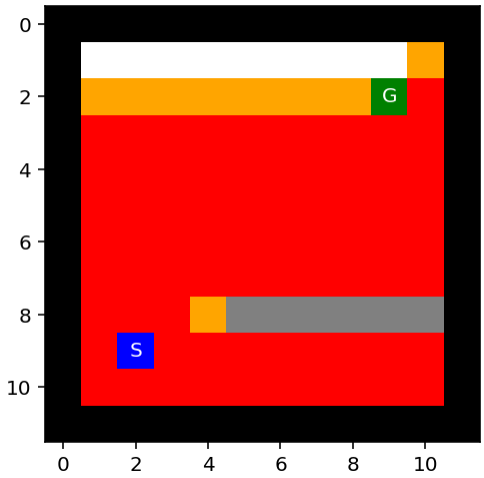


empty_maze.txt

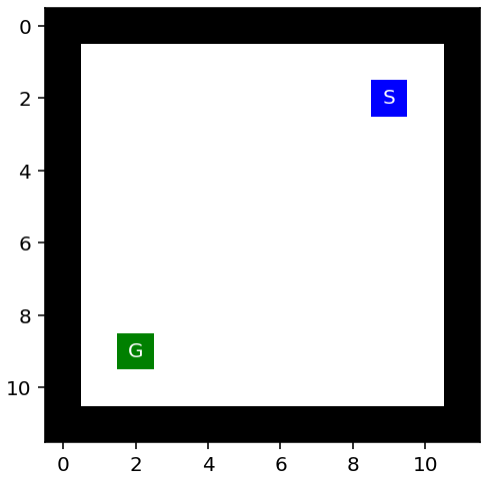



```
Maximum Depth of the tree:  74
Cost for the Path:  74
Maximum tree size:  155
Maximum Branching Factor:  81
Path:
(9, 2)
(9, 1)
(10, 1)
(10, 2)
(10, 3)
(10, 4)
(10, 5)
(10, 6)
(10, 7)
(10, 8)
(10, 9)
(10, 10)
(9, 10)
(9, 9)
(9, 8)
(9, 7)
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(8, 3)
(8, 2)
(8, 1)
(7, 1)
(7, 2)
(7, 3)
(7, 4)
(7, 5)
(7, 6)
(7, 7)
(7, 8)
(7, 9)
(7, 10)
(6, 10)
(6, 9)
(6, 8)
(6, 7)
(6, 6)
(6, 5)
(6, 4)
(6, 3)
(6, 2)
(6, 1)
(5, 1)
```

```
(5, 2)
(5, 3)
(5, 4)
(5, 5)
(5, 6)
(5, 7)
(5, 8)
(5, 9)
(5, 10)
(4, 10)
(4, 9)
(4, 8)
(4, 7)
(4, 6)
(4, 5)
(4, 4)
(4, 3)
(4, 2)
(4, 1)
(3, 1)
(3, 2)
(3, 3)
(3, 4)
(3, 5)
(3, 6)
(3, 7)
(3, 8)
(3, 9)
(3, 10)
(2, 10)
(2, 9)
Cost: 74
```

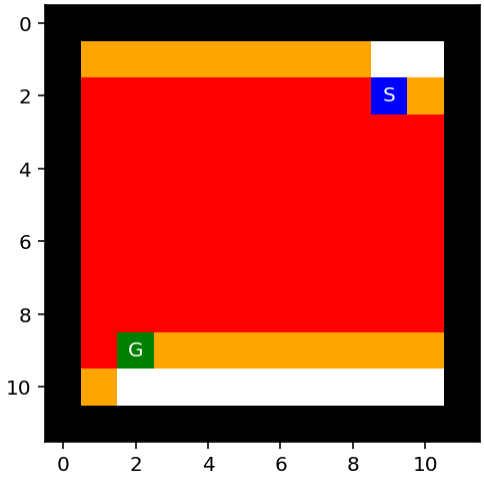



```
empty_2_maze.txt
```



```
Maximum Depth of the tree:  70
Cost for the Path:  70
Maximum tree size:  145
Maximum Branching Factor:  70
Path:
(2, 9)
(2, 8)
(2, 7)
(2, 6)
(2, 5)
(2, 4)
(2, 3)
(2, 2)
(2, 1)
(3, 1)
(3, 2)
(3, 3)
(3, 4)
(3, 5)
(3, 6)
(3, 7)
(3, 8)
(3, 9)
(3, 10)
(4, 10)
(4, 9)
(4, 8)
(4, 7)
(4, 6)
(4, 5)
(4, 4)
(4, 3)
(4, 2)
(4, 1)
(5, 1)
(5, 2)
(5, 3)
(5, 4)
(5, 5)
(5, 6)
(5, 7)
(5, 8)
(5, 9)
(5, 10)
(6, 10)
(6, 9)
(6, 8)
(6, 7)
(6, 6)
(6, 5)
(6, 4)
(6, 3)
(6, 2)
(6, 1)
(7, 1)
(7, 2)
(7, 3)
(7, 4)
(7, 5)
(7, 6)
(7, 7)
(7, 8)
(7, 9)
(7, 10)
```

```
(8, 10)
(8, 9)
(8, 8)
(8, 7)
(8, 6)
(8, 5)
(8, 4)
(8, 3)
(8, 2)
(8, 1)
(9, 1)
(9, 2)
Cost: 70
```



How does BFS and DFS deal with loops (cycles)?

In order to prevent loops and cycles, BFS and DFS maintain track of the explored states. When they get close to a place that has already been explored, they skip it and do not add it to the tree. DFS is not optimal or complete, but the BFS implementation is both complete and optimal. The DFS algorithm starts at the end of a path or action and works its way toward the objective or a dead end, whereas the BFS algorithm visits all of its nearby nodes first.

Are your implementations complete and optimal? Explain why. What is the time and space complexity of each of **your** implementations?

The worst space and time complexities are O(M*N) and O(M*N) where M*N is the maze's size and may or may not have a goal or be in the farthest position from it.

Task 3: Informed search: Implement greedy best-first search and A* search [20 Points]

You can use the map to estimate the distance from your current position to the goal using the Manhattan distance (see https://en.wikipedia.org/wiki/Taxicab_geometry) as a heuristic function. Both algorithms are based on Best-First search which requires only a small change from the BFS algorithm you have already implemented (see textbook/slides).

In [190…

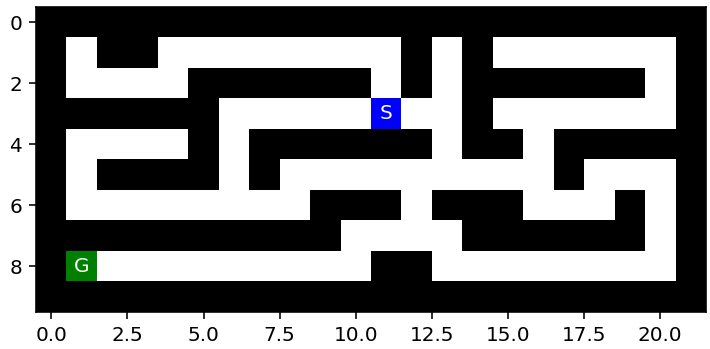
```
from queue import PriorityQueue

def find_neighbors(currMaze,node, mapBoard, goal, actions):
    currPos = node.pos
    n = []

    # Iterate over possible actions
    for action in actions.values():
```

```
        new = (currPos[0]+action[0],currPos[1]+action[1])

        if look(mapBoard,new) != "X":
            n.append(Node(new, node,action, node.cost + 1, goal=goal))

    return n

def gbfs(maze,start,goal,actions):
    explored = set()
    frontier = PriorityQueue()

    startNode = Node(start,None, None, 0,goal=goal)

    frontier.put((startNode.distance,startNode))

    count,size,depth,bf = 0,1,0,0

    while not frontier.empty():

        currNode = frontier.get()[1]
        currPos = currNode.pos

        if currNode.cost > depth :
            depth = currNode.cost

        if currPos == goal :
            print('Depth of the tree: ', depth)
            print('Cost for the Path: ', currNode.cost)
            print('Tree size is', size)
            print('Branching Factor:  ', count)
            d.append([currMaze,'gbfs',frontier.qsize(),depth, currNode.cost, size, c
            return currNode

        explored.add(currPos)
        #Getting possible actions to perform
        neighbors = find_neighbors(currNode, maze, goal,actions)

        for n in neighbors:
            branchingFactor= 0
            new = n.pos

            if new not in explored:
                frontier.put((n.distance,n))

                if currPos !=start and currPos!=goal and new != goal:
                    maze[new[0]][new[1]] = "F"
                    maze[currPos[0]][currPos[1]] = "."
                size+=1
                branchingFactor+=1
            if bf < branchingFactor:
                bf= branchingFactor
        count = count + 1

    return None
```

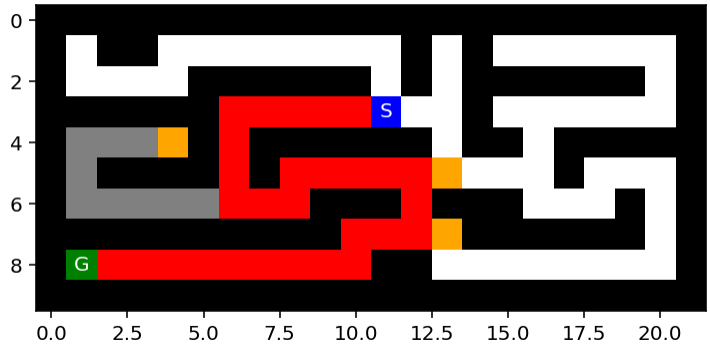
In [191…

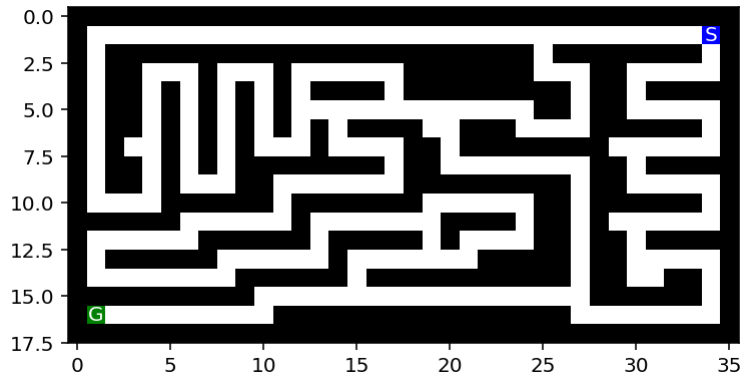
```
for f in mazeName:
    with open(f, "r") as f:
        maze_str = f.read()
    maze = parse_maze(maze_str)
    show_maze(maze)
    start = []
    goal = []
    start = find_pos(maze)
```

```
    goal = find_pos(maze, what = "G")
    actions = {"north":(-1,0),"south":(1,0),"east":(0,1),"west":(0,-1)}
    rootNode = Node(start,None, None, 0)
    finalNode = gbfs(f,maze,start,goal,actions)
    if finalNode:
        path = finalNode.get_path_from_root()
        print("Path from start to goal:")
        for node in path:
            print(node.pos)
            if(node.pos !=start and node.pos!=goal):
                maze[node.pos[0]][node.pos[1]] = "P"
        print("Cost:",finalNode.cost)
        show_maze(maze)
    else:
        print("Goal not found")
```



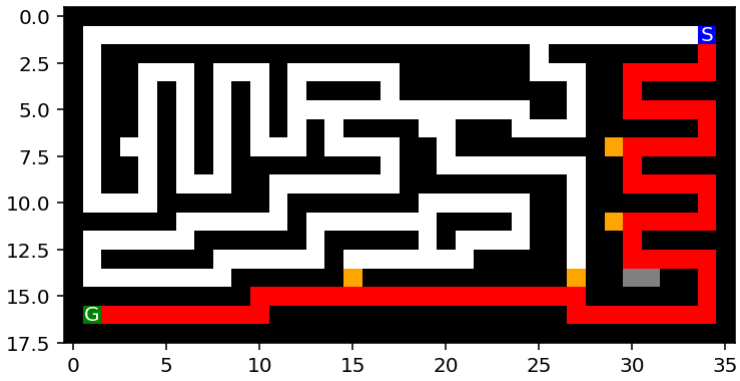

```
Depth of the tree:  29
Cost for the Path:  29
Tree size is 44
Branching Factor:   39
Path from start to goal:
(3, 11)
(3, 10)
(3, 9)
(3, 8)
(3, 7)
(3, 6)
(4, 6)
(5, 6)
(6, 6)
(6, 7)
(6, 8)
(5, 8)
(5, 9)
(5, 10)
(5, 11)
(5, 12)
(6, 12)
(7, 12)
(7, 11)
(7, 10)
(8, 10)
(8, 9)
(8, 8)
(8, 7)
(8, 6)
(8, 5)
(8, 4)
(8, 3)
(8, 2)
(8, 1)
Cost: 29
```

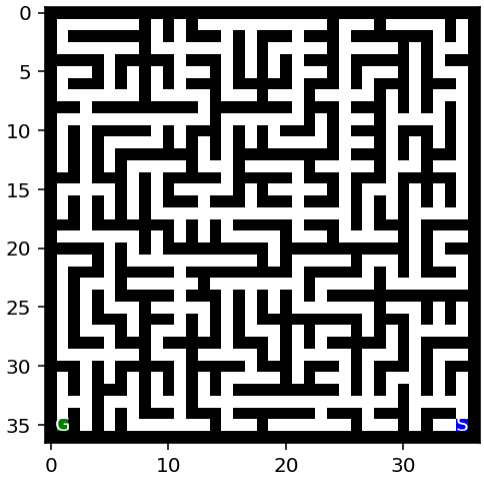





```
Depth of the tree:  74
Cost for the Path:  74
Tree size is 86
Branching Factor:   81
Path from start to goal:
(1, 34)
(2, 34)
(3, 34)
(3, 33)
(3, 32)
(3, 31)
(3, 30)
(4, 30)
(5, 30)
(5, 31)
(5, 32)
(5, 33)
(5, 34)
(6, 34)
(7, 34)
(7, 33)
(7, 32)
(7, 31)
(7, 30)
(8, 30)
(9, 30)
(9, 31)
(9, 32)
(9, 33)
(9, 34)
(10, 34)
(11, 34)
(11, 33)
(11, 32)
(11, 31)
(11, 30)
(12, 30)
(13, 30)
(13, 31)
(13, 32)
```

```
(13, 33)
(13, 34)
(14, 34)
(15, 34)
(16, 34)
(16, 33)
(16, 32)
(16, 31)
(16, 30)
(16, 29)
(16, 28)
(16, 27)
(15, 27)
(15, 26)
(15, 25)
(15, 24)
(15, 23)
(15, 22)
(15, 21)
(15, 20)
(15, 19)
(15, 18)
(15, 17)
(15, 16)
(15, 15)
(15, 14)
(15, 13)
(15, 12)
(15, 11)
(15, 10)
(16, 10)
(16, 9)
(16, 8)
(16, 7)
(16, 6)
(16, 5)
(16, 4)
(16, 3)
(16, 2)
(16, 1)
Cost: 74
```



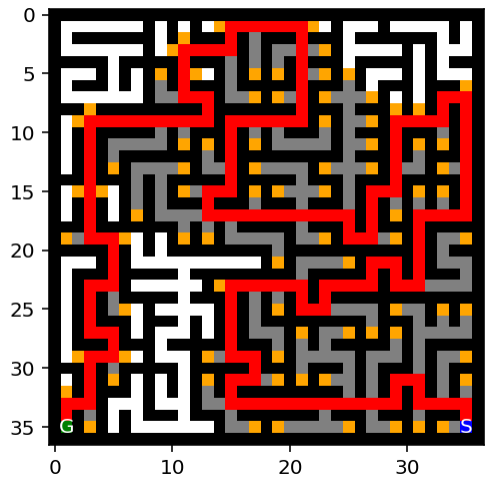


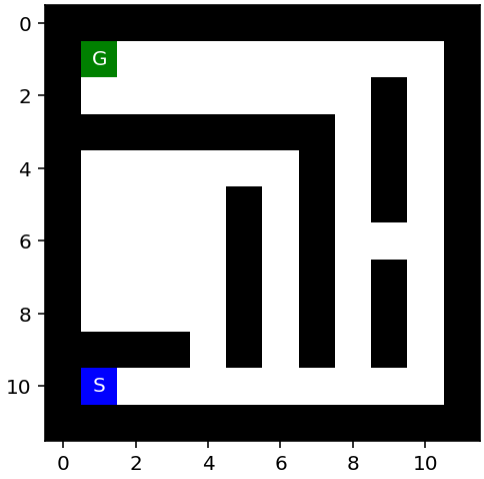

```
Depth of the tree:  210
Cost for the Path:  210
Tree size is 485
Branching Factor:   463
Path from start to goal:
(35, 35)
(34, 35)
(33, 35)
(33, 34)
(33, 33)
(33, 32)
(33, 31)
(32, 31)
(31, 31)
(31, 30)
(31, 29)
(32, 29)
(33, 29)
(33, 28)
(33, 27)
(33, 26)
(33, 25)
(33, 24)
(33, 23)
(33, 22)
(33, 21)
(33, 20)
(33, 19)
(33, 18)
(33, 17)
(33, 16)
(33, 15)
(32, 15)
(31, 15)
(31, 16)
(31, 17)
(30, 17)
(29, 17)
(29, 16)
(29, 15)
(28, 15)
(27, 15)
(26, 15)
(25, 15)
(24, 15)
(23, 15)
(23, 16)
(23, 17)
```

(23, 18)
(23, 19)
(23, 20)
(23, 21)
(24, 21)
(25, 21)
(25, 22)
(25, 23)
(24, 23)
(23, 23)
(23, 24)
(23, 25)
(23, 26)
(23, 27)
(22, 27)
(21, 27)
(21, 28)
(21, 29)
(22, 29)
(23, 29)
(23, 30)
(23, 31)
(22, 31)
(21, 31)
(20, 31)
(19, 31)
(18, 31)
(17, 31)
(17, 32)
(17, 33)
(17, 34)
(17, 35)
(16, 35)
(15, 35)
(14, 35)
(13, 35)
(12, 35)
(11, 35)
(10, 35)
(9, 35)
(8, 35)
(7, 35)
(7, 34)
(7, 33)
(8, 33)
(9, 33)
(9, 32)
(9, 31)
(9, 30)
(9, 29)
(10, 29)
(11, 29)
(12, 29)
(13, 29)
(14, 29)
(15, 29)
(15, 28)
(15, 27)
(16, 27)
(17, 27)
(18, 27)
(19, 27)
(19, 26)
(19, 25)

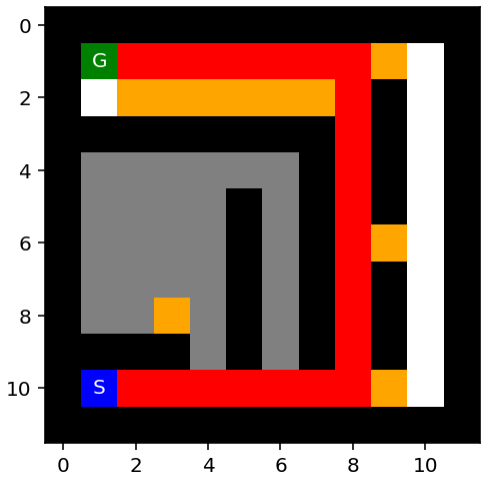
(18, 25)
(17, 25)
(17, 24)
(17, 23)
(17, 22)
(17, 21)
(17, 20)
(17, 19)
(17, 18)
(17, 17)
(17, 16)
(17, 15)
(17, 14)
(17, 13)
(16, 13)
(15, 13)
(15, 14)
(15, 15)
(14, 15)
(13, 15)
(12, 15)
(11, 15)
(10, 15)
(9, 15)
(9, 16)
(9, 17)
(9, 18)
(9, 19)
(9, 20)
(9, 21)
(8, 21)
(7, 21)
(6, 21)
(5, 21)
(4, 21)
(3, 21)
(2, 21)
(1, 21)
(1, 20)
(1, 19)
(1, 18)
(1, 17)
(1, 16)
(1, 15)
(2, 15)
(3, 15)
(3, 14)
(3, 13)
(3, 12)
(3, 11)
(4, 11)
(5, 11)
(6, 11)
(7, 11)
(7, 12)
(7, 13)
(8, 13)
(9, 13)
(9, 12)
(9, 11)
(9, 10)
(9, 9)
(9, 8)
(9, 7)

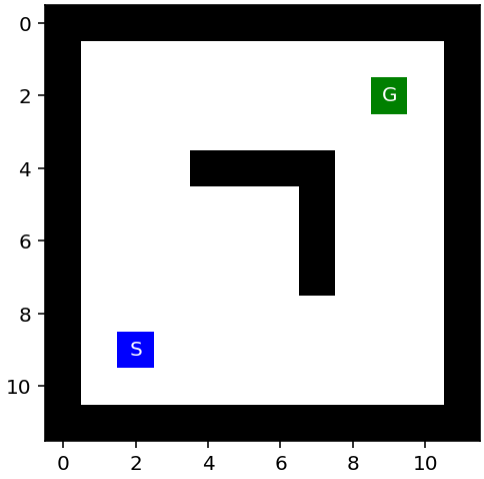
```
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(10, 3)
(11, 3)
(12, 3)
(13, 3)
(14, 3)
(15, 3)
(16, 3)
(17, 3)
(18, 3)
(19, 3)
(19, 4)
(19, 5)
(20, 5)
(21, 5)
(22, 5)
(23, 5)
(23, 4)
(23, 3)
(24, 3)
(25, 3)
(26, 3)
(27, 3)
(27, 4)
(27, 5)
(28, 5)
(29, 5)
(29, 4)
(29, 3)
(30, 3)
(31, 3)
(32, 3)
(33, 3)
(33, 2)
(33, 1)
(34, 1)
(35, 1)
Cost: 210
```



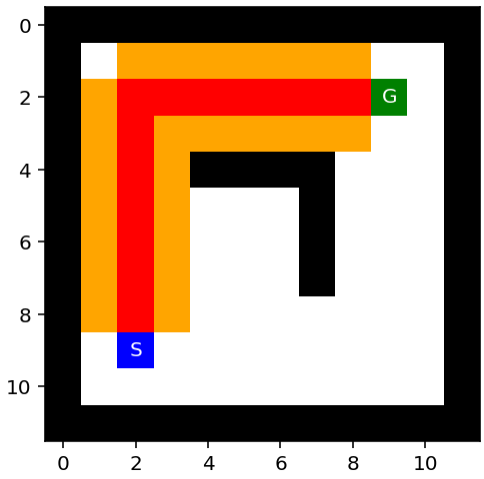


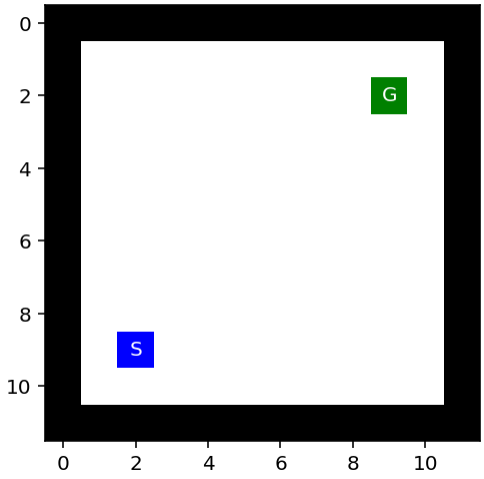

```
Depth of the tree:  23
Cost for the Path:  23
Tree size is 115
Branching Factor:   103
Path from start to goal:
(10, 1)
(10, 2)
(10, 3)
(10, 4)
(10, 5)
(10, 6)
(10, 7)
(10, 8)
(9, 8)
(8, 8)
(7, 8)
(6, 8)
(5, 8)
(4, 8)
(3, 8)
(2, 8)
(1, 8)
(1, 7)
(1, 6)
(1, 5)
(1, 4)
(1, 3)
(1, 2)
(1, 1)
Cost: 23
```



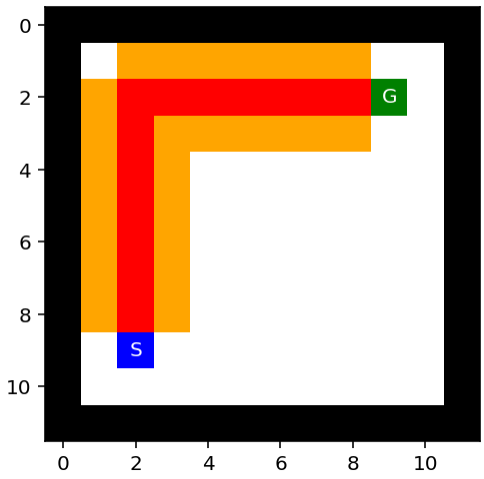


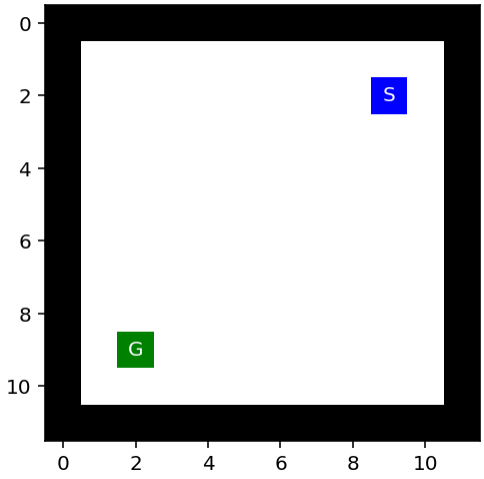

```
Depth of the tree:  14
Cost for the Path:  14
Tree size is 44
Branching Factor:   14
Path from start to goal:
(9, 2)
(8, 2)
(7, 2)
(6, 2)
(5, 2)
(4, 2)
(3, 2)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(2, 7)
(2, 8)
(2, 9)
Cost: 14
```



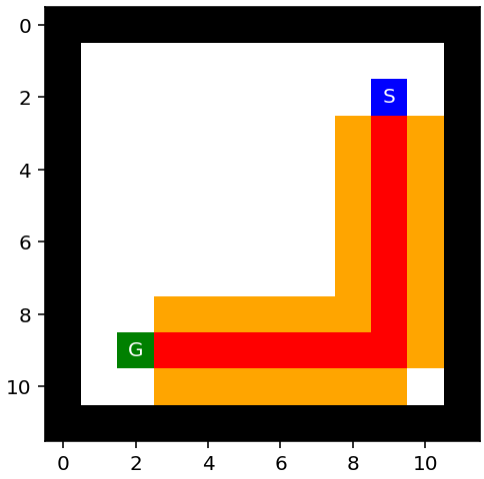



```
Depth of the tree:  14
Cost for the Path:  14
Tree size is 44
Branching Factor:   14
Path from start to goal:
(9, 2)
(8, 2)
(7, 2)
(6, 2)
(5, 2)
(4, 2)
(3, 2)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(2, 7)
(2, 8)
(2, 9)
Cost: 14
```






```
Depth of the tree:  14
Cost for the Path:  14
Tree size is 44
Branching Factor:   14
Path from start to goal:
(2, 9)
(3, 9)
(4, 9)
(5, 9)
(6, 9)
(7, 9)
(8, 9)
(9, 9)
(9, 8)
(9, 7)
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(9, 2)
Cost: 14
```



A* Algorithm

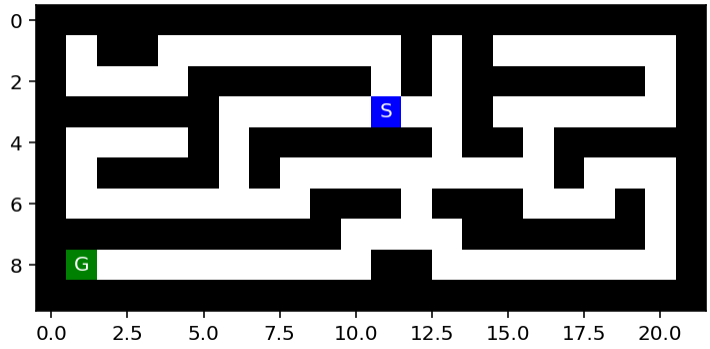
In [201…

```python
def a_star(maze,start,goal,actions):
    explored = set()
    frontier = PriorityQueue()

    startNode = Node(start, None, None, 0, goal=goal)
    frontier.put((startNode.distance + 0, startNode))
```

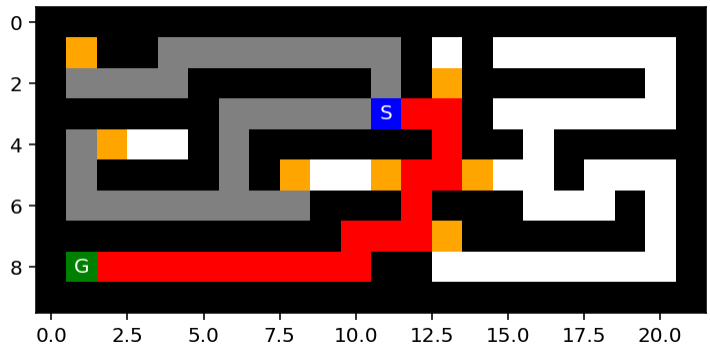
```
    count,size,depth,bf = 0,1,0,0

    while frontier:

        currNode = frontier.get()[1]
        currPos = currNode.pos

        if currNode.cost > depth :
            depth = currNode.cost

        if currPos == goal :
            print('Depth of the tree: ', depth)
            print('Cost for the Path: ', currNode.cost)
            print('Maximum tree size: ', size)
            print('Maximum Branching Factor: ', count)
            d.append([currentMaze,'a star',frontier.qsize(),maxDepth, currNode.cost,
            return currNode

        explored.add(currPos)
        neighbors = get_possible_neighbors(currNode, maze, goal,actions)

        for n in neighbors:
            branchingFactor=0
            new = n.pos

            if new not in explored:
                frontier.put((n.distance + n.cost,n))

                if currPos !=start and currPos!=goal and new != goal :
                    maze[new[0]][new[1]] = "F"
                    maze[currPos[0]][currPos[1]] = "."
                size+=1
                branchingFactor+=1
            if bf<branchingFactor:
                bf= branchingFactor
        count = count + 1

    return None
```

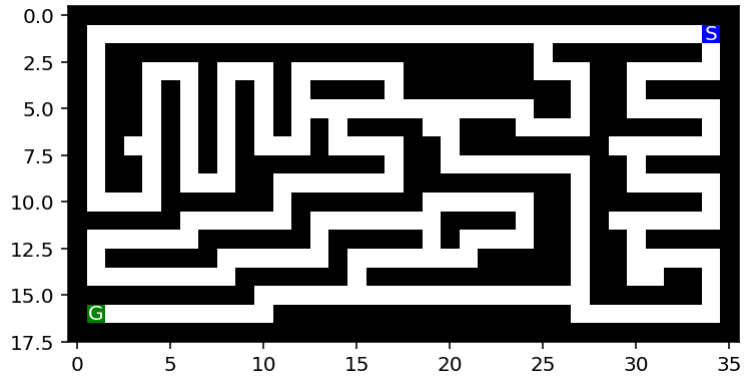
In [199…

```
for f in mazeName:
    currentMaze = f[:f.find(".")]
    with open(f, "r") as f:
        maze_str = f.read()
    maze = parse_maze(maze_str)
    show_maze(maze)
    start = []
    goal = []
    start = find_pos(maze)
    goal = find_pos(maze, what = "G")
    actions = {"north":(-1,0),"south":(1,0),"east":(0,1),"west":(0,-1)}
    rootNode = Node(start,None, None, 0)
    finalNode = a_star_search(maze,start,goal,actions)
    if finalNode:
        path = finalNode.get_path_from_root()
        print("Path:")
        for node in path:
            print(node.pos)
            if(node.pos !=start and node.pos!=goal):
                maze[node.pos[0]][node.pos[1]] = "P"
        print("Cost:",finalNode.cost)
        show_maze(maze)
    else:
        print("Goal not found")
```




```
The Maximum Depth of the tree is 19
The cost for the Path is 19
The Maximum tree size is 57
The maximum Branching Factor is  49
Path:
(3, 11)
(3, 12)
(3, 13)
(4, 13)
(5, 13)
(5, 12)
(6, 12)
(7, 12)
(7, 11)
(7, 10)
(8, 10)
(8, 9)
(8, 8)
(8, 7)
(8, 6)
(8, 5)
(8, 4)
(8, 3)
(8, 2)
(8, 1)
Cost: 19
```

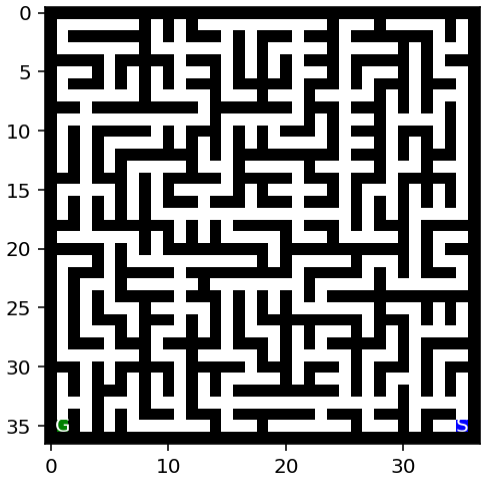





```
The Maximum Depth of the tree is 68
The cost for the Path is 68
```

```
The Maximum tree size is 232
The maximum Branching Factor is  224
Path:
(1, 34)
(1, 33)
(1, 32)
(1, 31)
(1, 30)
(1, 29)
(1, 28)
(1, 27)
(1, 26)
(1, 25)
(2, 25)
(3, 25)
(3, 26)
(3, 27)
(4, 27)
(5, 27)
(6, 27)
(6, 26)
(6, 25)
(6, 24)
(5, 24)
(5, 23)
(5, 22)
(5, 21)
(5, 20)
(6, 20)
(7, 20)
(8, 20)
(8, 21)
(8, 22)
(8, 23)
(8, 24)
(8, 25)
(8, 26)
(8, 27)
(9, 27)
(10, 27)
(11, 27)
(12, 27)
(13, 27)
(14, 27)
(15, 27)
(15, 26)
(15, 25)
(15, 24)
(15, 23)
(15, 22)
(15, 21)
(15, 20)
(15, 19)
(15, 18)
(15, 17)
(15, 16)
(15, 15)
(15, 14)
(15, 13)
(15, 12)
(15, 11)
(15, 10)
(16, 10)
(16, 9)
```

```
(16, 8)
(16, 7)
(16, 6)
(16, 5)
(16, 4)
(16, 3)
(16, 2)
(16, 1)
Cost: 68
```

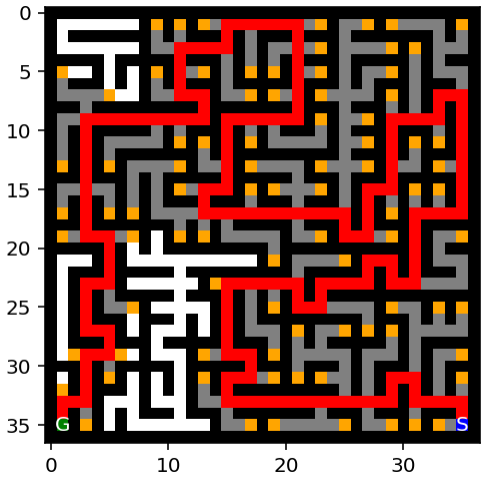


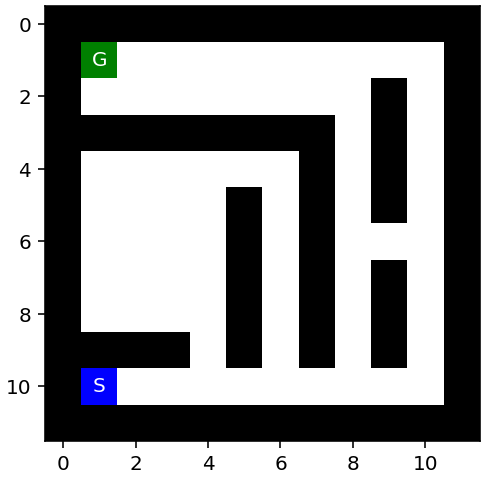

```
The Maximum Depth of the tree is 210
The cost for the Path is 210
The Maximum tree size is 547
The maximum Branching Factor is  539
Path:
(35, 35)
(34, 35)
(33, 35)
(33, 34)
(33, 33)
(33, 32)
(33, 31)
(32, 31)
(31, 31)
(31, 30)
(31, 29)
(32, 29)
(33, 29)
(33, 28)
(33, 27)
(33, 26)
(33, 25)
(33, 24)
(33, 23)
(33, 22)
(33, 21)
(33, 20)
```

(33, 19)
(33, 18)
(33, 17)
(33, 16)
(33, 15)
(32, 15)
(31, 15)
(31, 16)
(31, 17)
(30, 17)
(29, 17)
(29, 16)
(29, 15)
(28, 15)
(27, 15)
(26, 15)
(25, 15)
(24, 15)
(23, 15)
(23, 16)
(23, 17)
(23, 18)
(23, 19)
(23, 20)
(23, 21)
(24, 21)
(25, 21)
(25, 22)
(25, 23)
(24, 23)
(23, 23)
(23, 24)
(23, 25)
(23, 26)
(23, 27)
(22, 27)
(21, 27)
(21, 28)
(21, 29)
(22, 29)
(23, 29)
(23, 30)
(23, 31)
(22, 31)
(21, 31)
(20, 31)
(19, 31)
(18, 31)
(17, 31)
(17, 32)
(17, 33)
(17, 34)
(17, 35)
(16, 35)
(15, 35)
(14, 35)
(13, 35)
(12, 35)
(11, 35)
(10, 35)
(9, 35)
(8, 35)
(7, 35)
(7, 34)

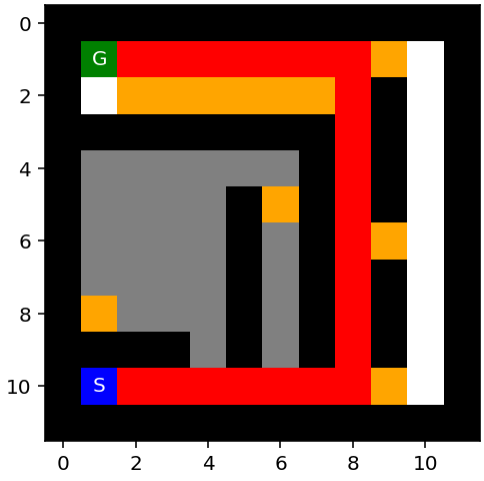
(7, 33)
(8, 33)
(9, 33)
(9, 32)
(9, 31)
(9, 30)
(9, 29)
(10, 29)
(11, 29)
(12, 29)
(13, 29)
(14, 29)
(15, 29)
(15, 28)
(15, 27)
(16, 27)
(17, 27)
(18, 27)
(19, 27)
(19, 26)
(19, 25)
(18, 25)
(17, 25)
(17, 24)
(17, 23)
(17, 22)
(17, 21)
(17, 20)
(17, 19)
(17, 18)
(17, 17)
(17, 16)
(17, 15)
(17, 14)
(17, 13)
(16, 13)
(15, 13)
(15, 14)
(15, 15)
(14, 15)
(13, 15)
(12, 15)
(11, 15)
(10, 15)
(9, 15)
(9, 16)
(9, 17)
(9, 18)
(9, 19)
(9, 20)
(9, 21)
(8, 21)
(7, 21)
(6, 21)
(5, 21)
(4, 21)
(3, 21)
(2, 21)
(1, 21)
(1, 20)
(1, 19)
(1, 18)
(1, 17)
(1, 16)

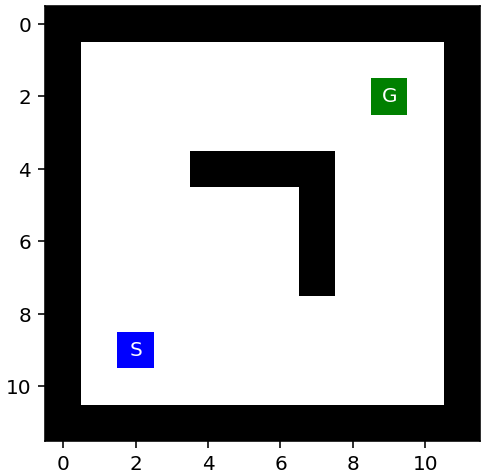
```
(1, 15)
(2, 15)
(3, 15)
(3, 14)
(3, 13)
(3, 12)
(3, 11)
(4, 11)
(5, 11)
(6, 11)
(7, 11)
(7, 12)
(7, 13)
(8, 13)
(9, 13)
(9, 12)
(9, 11)
(9, 10)
(9, 9)
(9, 8)
(9, 7)
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(10, 3)
(11, 3)
(12, 3)
(13, 3)
(14, 3)
(15, 3)
(16, 3)
(17, 3)
(18, 3)
(19, 3)
(19, 4)
(19, 5)
(20, 5)
(21, 5)
(22, 5)
(23, 5)
(23, 4)
(23, 3)
(24, 3)
(25, 3)
(26, 3)
(27, 3)
(27, 4)
(27, 5)
(28, 5)
(29, 5)
(29, 4)
(29, 3)
(30, 3)
(31, 3)
(32, 3)
(33, 3)
(33, 2)
(33, 1)
(34, 1)
(35, 1)
Cost: 210
```



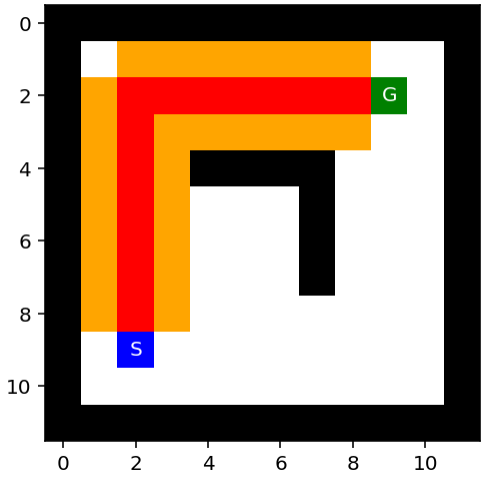


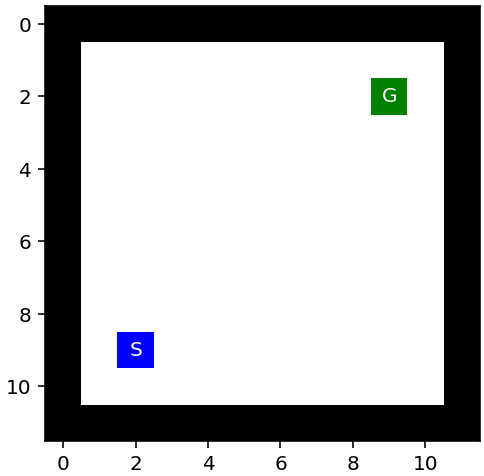
The Maximum Depth of the tree is 23
The cost for the Path is 23
The Maximum tree size is 75
The maximum Branching Factor is 64
Path:
(10, 1)
(10, 2)
(10, 3)
(10, 4)
(10, 5)
(10, 6)
(10, 7)
(10, 8)
(9, 8)
(8, 8)
(7, 8)
(6, 8)
(5, 8)
(4, 8)
(3, 8)
(2, 8)
(1, 8)
(1, 7)
(1, 6)
(1, 5)
(1, 4)
(1, 3)
(1, 2)
(1, 1)
Cost: 23



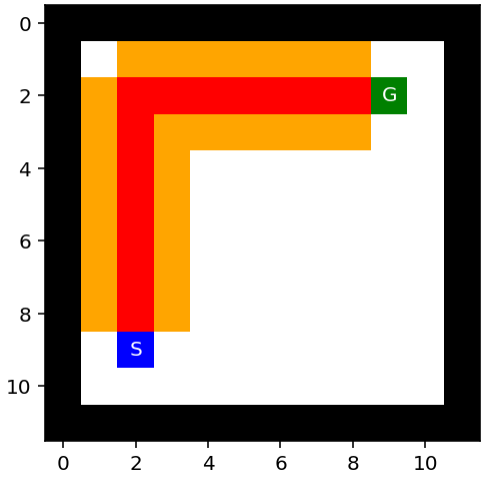


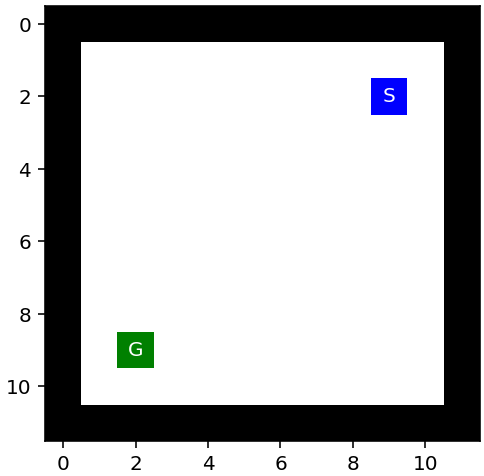

```
The Maximum Depth of the tree is 14
The cost for the Path is 14
The Maximum tree size is 44
The maximum Branching Factor is  14
Path:
(9, 2)
(8, 2)
(7, 2)
(6, 2)
(5, 2)
(4, 2)
(3, 2)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(2, 7)
(2, 8)
(2, 9)
Cost: 14
```



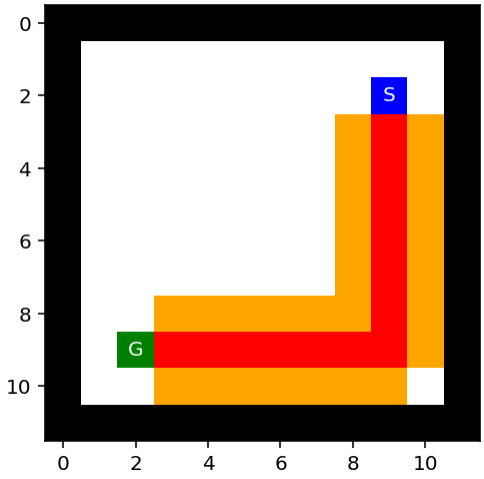



```
The Maximum Depth of the tree is 14
The cost for the Path is 14
The Maximum tree size is 44
The maximum Branching Factor is  14
Path:
(9, 2)
(8, 2)
(7, 2)
(6, 2)
(5, 2)
(4, 2)
(3, 2)
(2, 2)
(2, 3)
(2, 4)
(2, 5)
(2, 6)
(2, 7)
(2, 8)
(2, 9)
Cost: 14
```






```
The Maximum Depth of the tree is 14
The cost for the Path is 14
The Maximum tree size is 44
The maximum Branching Factor is  14
Path:
(2, 9)
(3, 9)
(4, 9)
(5, 9)
(6, 9)
(7, 9)
(8, 9)
(9, 9)
(9, 8)
(9, 7)
(9, 6)
(9, 5)
(9, 4)
(9, 3)
(9, 2)
Cost: 14
```



Greedy search is complete but not optimal, while A-star is both optimal and complete. O(N) and O(N) is the worst case scenario for space and time difficulties because it only adds feasible nodes to the tree and determines which nodes to visit.

Are your implementations complete and optimal? What is the time and space complexity?

Task 4: Comparison and discussion [20 Points]

Run experiments to compare the implemented algorithms.

How to deal with issues:

- Your implementation returns unexpected results: Try to debug and fix the code. Visualizing the maze, the current path and the frontier after every step is very helpful. If the code still does not work, then mark the result with an asterisk (*) and describe the issue below the table.
- Your implementation cannot consistently solve a specific maze and ends up in an infinite loop: Debug. If it is a shortcoming of the algorithm/implementation, then put "N/A*" in the results table and describe why this is happening.

In [103…

```
import pandas as pd
def compare() :
    global d
    df= pd.DataFrame(d, columns = ['maze','search', 'frontier_size', 'tree_depth','p
    display(df.groupby(['search','maze']).mean())
    display (df)
    return df
df= compare()
print(df)
```

		frontier_size	tree_depth	path_cost	tree_size	nodes_expanded
search	**maze**					
BFS	**empty_2_maze**	9437.00	14.00	14.00	22978.0	13540.00
	empty_maze	9437.00	14.00	14.00	22978.0	13540.00
	large_maze	2.00	210.00	210.00	623.0	620.00
	loops_maze	7.00	23.00	23.00	213.0	205.00

search	maze	frontier_size	tree_depth	path_cost	tree_size	nodes_expanded
	medium_maze	2.00	68.00	68.00	278.0	275.00
	small_maze	1.00	19.00	19.00	96.0	94.00
	wall_maze	421.00	14.00	14.00	1726.0	1304.00
DFS	**empty_2_maze**	75.00	70.00	70.00	145.0	70.00
	empty_maze	55.75	55.50	55.50	116.5	60.75
	large_maze	37.00	210.25	201.25	395.5	358.50
	loops_maze	12.00	27.00	27.00	42.0	30.00
	medium_maze	9.00	130.00	130.00	155.0	146.00
	small_maze	7.00	49.00	49.00	66.0	59.00
	wall_maze	44.50	52.50	48.00	110.5	66.00
a star	**empty_2_maze**	29.00	14.00	14.00	44.0	14.00
	empty_maze	29.00	14.00	14.00	44.0	14.00
	large_maze	7.00	210.00	210.00	547.0	539.00
	loops_maze	10.00	23.00	23.00	75.0	64.00
	medium_maze	7.00	68.00	68.00	232.0	224.00
	small_maze	7.00	19.00	19.00	57.0	49.00
	wall_maze	29.00	14.00	14.00	44.0	14.00
gbfs	**empty_2_maze**	30.00	14.00	14.00	44.0	14.00
	empty_maze	30.00	14.00	14.00	44.0	14.00
	large_maze	22.00	210.00	210.00	485.0	463.00
	loops_maze	21.00	23.00	23.00	115.0	103.00
	medium_maze	5.00	74.00	74.00	86.0	81.00
	small_maze	5.00	29.00	29.00	44.0	39.00
	wall_maze	30.00	14.00	14.00	44.0	14.00

	maze	search	frontier_size	tree_depth	path_cost	tree_size	nodes_expanded
0	small_maze	BFS	1	19	19	96	94
1	medium_maze	BFS	2	68	68	278	275
2	large_maze	BFS	2	210	210	623	620
3	loops_maze	BFS	7	23	23	213	205
4	wall_maze	BFS	421	14	14	1726	1304
...	...	...	...	...	...	...	...
84	large_maze	a star	7	210	210	547	539
85	loops_maze	a star	10	23	23	75	64
86	wall_maze	a star	29	14	14	44	14
87	empty_maze	a star	29	14	14	44	14

	maze	search	frontier_size	tree_depth	path_cost	tree_size	nodes_expanded
88	empty_2_maze	a star	29	14	14	44	14

89 rows × 7 columns

```
            maze  search  frontier_size  tree_depth  path_cost  tree_size  \
0     small_maze     BFS              1          19         19         96
1    medium_maze     BFS              2          68         68        278
2     large_maze     BFS              2         210        210        623
3     loops_maze     BFS              7          23         23        213
4      wall_maze     BFS            421          14         14       1726
..           ...     ...            ...         ...        ...        ...
84    large_maze  a star              7         210        210        547
85    loops_maze  a star             10          23         23         75
86     wall_maze  a star             29          14         14         44
87    empty_maze  a star             29          14         14         44
88  empty_2_maze  a star             29          14         14         44

    nodes_expanded
0               94
1              275
2              620
3              205
4             1304
..             ...
84             539
85              64
86              14
87              14
88              14

[89 rows x 7 columns]
```

Complete the following table for each maze.

Small maze

algorithm	path cost	# of nodes expanded	max tree depth	max # of nodes in memory	max frontier size
BFS					
DFS					
GBS					
A*					

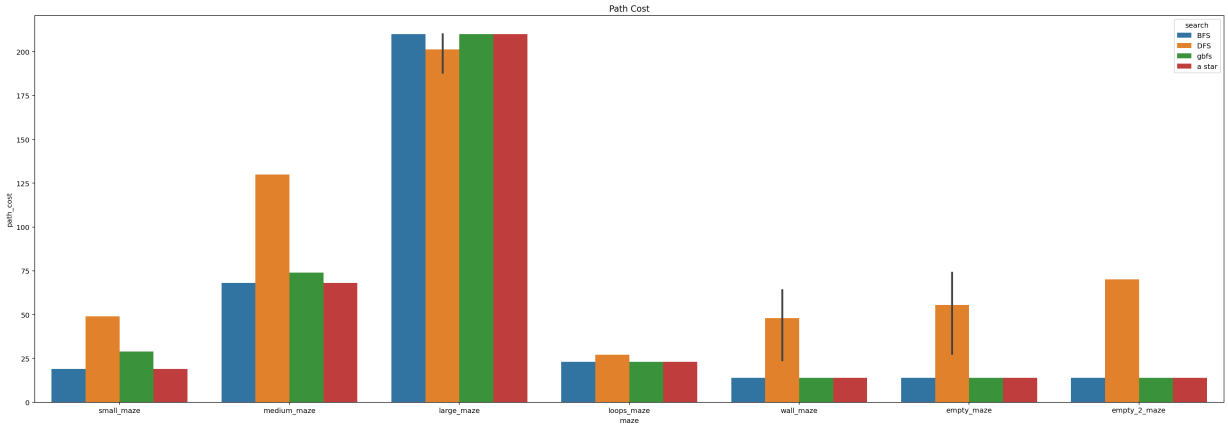
Medium Maze

...

Present the results as using charts (see Python Code Examples/charts and tables).

In [102…

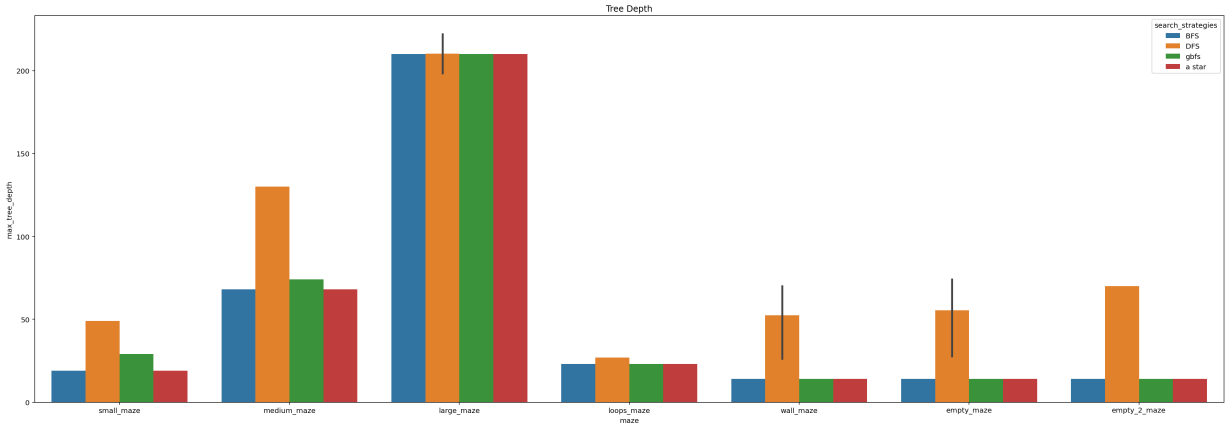
```
# Add charts
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(30, 10))
ax = sns.barplot(x="maze", y="path_cost", data=df,  hue='search')
ax.set_title('Path Cost')
plt.show()
```



Discuss the most important lessons you have learned from implementing the different search strategies.

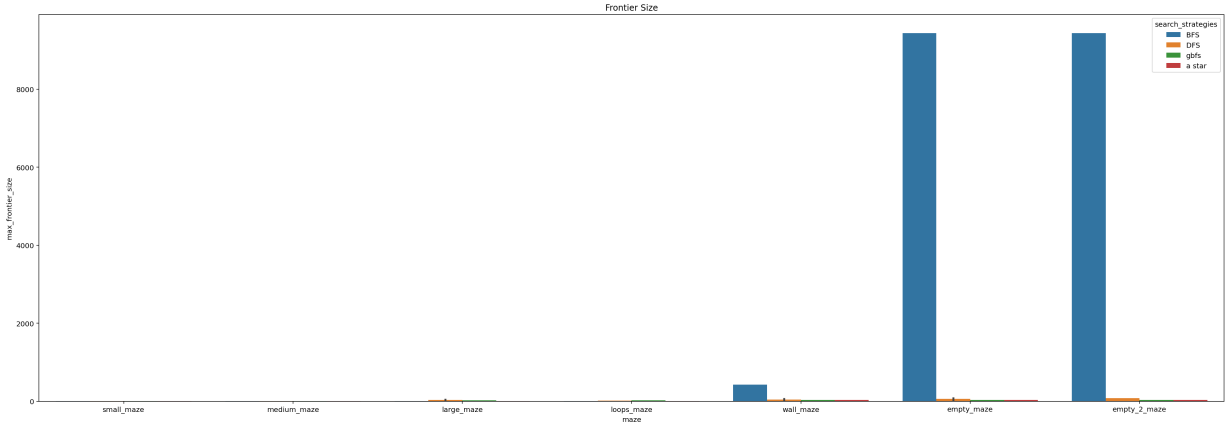
In [97]:

```
# Add discussion
import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(30, 10))
ax = sns.barplot(x="maze", y="max_tree_depth", data=df,  hue='search')
ax.set_title('Tree Depth')
plt.show()
```



In [98]:

```
import seaborn as sns
import matplotlib.pyplot as plt
plt.figure(figsize=(30, 10))
ax = sns.barplot(x="maze", y="max_frontier_size", data=df,  hue='search')
ax.set_title('Frontier Size')
plt.show()
```



Graduate student advanced task: IDS and Multiple goals [10 Points]

Undergraduate students: This is a bonus task you can attempt if you like [+5 Bonus Points].

Create a few mazes with multiple goals by adding one or two more goals to the medium size maze. Solve the maze with your implementations for DFS, BFS, and implement in addition IDS (iterative deepening search using DFS).

Run experiments to show which implementations find the optimal solution and which do not. Discuss why that is the case.

In [195…

```
class Graph:
    def __init__(self, maze):
        self.maze = maze
        self.rows = len(maze)
        self.cols = len(maze[0])

    def get_neighbors(self, node):
        row, col = node
        neighbors = []
        if row > 0 and not self.maze[row-1][col]:  # up
            neighbors.append((row-1, col))
        if row < self.rows-1 and not self.maze[row+1][col]:  # down
            neighbors.append((row+1, col))
        if col > 0 and not self.maze[row][col-1]:  # left
            neighbors.append((row, col-1))
        if col < self.cols-1 and not self.maze[row][col+1]:  # right
            neighbors.append((row, col+1))
        return neighbors

def dfs(graph, start, goals):
    stack = [(start, [start])]
    while stack:
        node, path = stack.pop()
        if node in goals:
            return path
        for neighbor in graph.get_neighbors(node):
            if neighbor not in path:
                stack.append((neighbor, path + [neighbor]))

def bfs(graph, start, goals):
    queue = [(start, [start])]
    while queue:
        node, path = queue.pop(0)
        if node in goals:
            return path
        for neighbor in graph.get_neighbors(node):
            if neighbor not in path:
                queue.append((neighbor, path + [neighbor]))

def ids(graph, start, goals, depth=10):
    for i in range(depth):
        result = dls(graph, start, goals, i)
        if result:
            return result

def dls(graph, node, goals, depth, path=[]):
    if depth == 0 and node in goals:
```

```
            return path + [node]
        if depth > 0:
            for neighbor in graph.get_neighbors(node):
                if neighbor not in path:
                    result = dls(graph, neighbor, goals, depth-1, path + [node])
                    if result:
                        return result
```

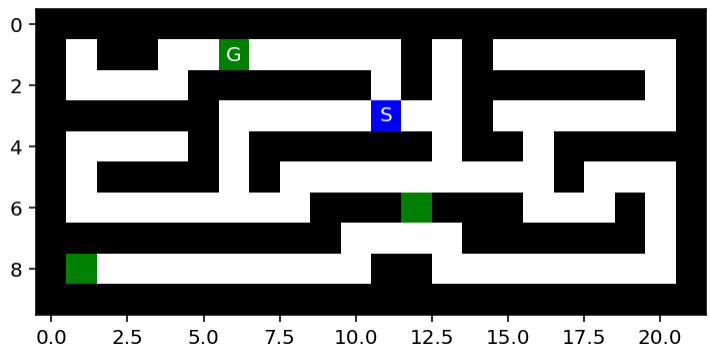
In [197…

```
maze = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 0, 0]
]

# Create a graph instance
graph = Graph(maze)

# Define the start and goal nodes
start = (1, 1)
goals = [(3, 3), (2, 2)]

# Solve the maze using DFS
dfs_path = dfs(graph, start, goals)
print("DFS path:", dfs_path)

# Solve the maze using BFS
bfs_path = bfs(graph, start, goals)
print("BFS path:", bfs_path)

# Solve the maze using IDS
ids_path = ids(graph, start, goals, depth=10)
print("IDS path:", ids_path)
```

```
DFS path: [(1, 1), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4),
(3, 4), (2, 4), (2, 3), (2, 2)]
BFS path: [(1, 1), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (2, 3), (2, 2)]
IDS path: [(1, 1), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (2, 3), (2, 2)]
```

In [110…

```
with open("multiple_goal_maze.txt", "r") as f:
    maze_str = f.read()
    muliple_goal = parse_maze(maze_str)
    show_maze(muliple_goal)
```



More advanced tasks to think about

Instead of defining each square as a state, use only intersections as states. Now the storage requirement is reduced, but the path length between two intersections can be different. If we use total path length measured as the number of squares as path cost, how can we make sure that BFS and iterative deepening search is optimal? Change the code to do so.

In [ ]:
```
# Your code/answer goes here
```

Modify your A* search to add weights (see text book) and explore how different weights influence the result.

In [ ]:
```
# Your code/answer goes here
```

What happens if the agent does not know the layout of the maze in advance (i.e., faces an unkown, only partially observable environment)? How does the environment look then (PEAS description)? How would you implement a rational agent to solve the maze? What if the agent still has a GPS device to tell the distance to the goal?

In [ ]:
```
# Your code/answer goes here
```