CAGATAY DUYGU

48369962

Search: Solving a Maze Using a Goal-based Agent

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 [1]:
```
with open("small_maze.txt", "r") as f:
    maze_ID_txt = f.read()
print(maze_ID_txt)
```

```
XXXXXXXXXXXXXXXXXXXXXX
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
XXXXXXXXXXXXXXXXXXXXXX
```

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

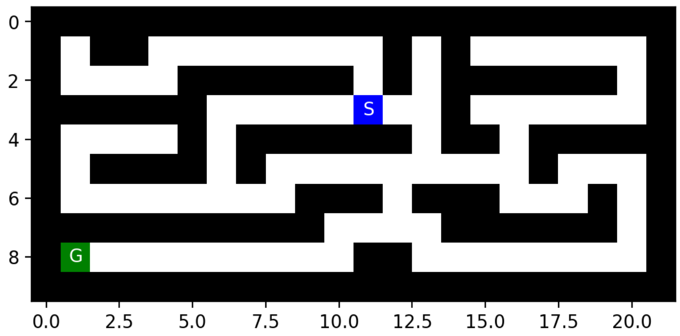
The maze can also be displayed in color using code in the module maze_helper.py. The code parses the string representing the maze and converts it into a `numpy` 2d array which you can use in your implementation. Position are represented as a 2-tuple of the form `(row, col)`.

In [2]:
```
import maze_helper as mh

maze = mh.parse_maze(maze_ID_txt)

# look at a position in the maze by subsetting the 2d array
print("Position(0,0):", maze[0, 0])

# there is also a helper function called `look(maze, pos)` available
# which uses a 2-tuple for the position.
print("Position(8,1):", mh.look(maze, (8, 1)))
```

```
Position(0,0): X
Position(8,1): G
```

A helper function to visualize the maze is also available.

In [3]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
# use higher resolution images in notebook

mh.show_maze(maze)
```



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

In [4]:
```python
print("Start location:", mh.find_pos(maze, what = "S"))
print("Goal location:", mh.find_pos(maze, what = "G"))
```

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

Helper function documentation.

In [5]:
```python
help(mh)
```

```
Help on module maze_helper:

NAME
    maze_helper

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

FUNCTIONS
    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.

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

    parse_maze(maze_str)
        Convert a maze as a string into a 2d numpy array

    show_maze(maze, fontsize=10)
        Display a (parsed) maze as an image.

    welcome()
        Welcome message.

FILE
    c:\users\yasin\cs7320-ai\search\hw\maze_helper.py
```

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 [6]:
```python
class Node:
    def __init__(self, pos, parent, action, cost):
        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 has None)
        self.cost = cost         # for uniform cost this is the depth. It is also g(n) for A* search

    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 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
- Cost

Use verbal descriptions, variables and equations as appropriate.

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

Definitions

- Initial state: The initial state is the state the agent begins the action. For instance, Arad was the initial state in the in-class example. For this homework problem, the initial state is the state the location "S" is. By running the command " *mh.find_pos(maze, what = "S")*."

- Actions: Actions(s) are the set of choices that is available in state "s".(Russel 64). In this assignment, actions are the set of positions that are available to move for next step in the current position. The actions for the initial state can begiven as follows:

  - *Actions(3,11)={(4,11), (5,11) (3,12)}* (Moving west, east, and north)

Going south is not in the action list because of the maze wall.

- Transition model: A transition model, which describes what each action does. For example, *RESULT(a,b)*, where the result is returned for performing action, *b*, in the state, *a*.For this assignment, a transition model is:
  - *Result((x,y), north)) --->(x-1,y)*
- Goal state: The goal state(s) defined as the desired state(s). There can be more than one goal state. For this search problem, the goal state is the position where the state "G" is. For graduate part of the homework, more than one G position will be given. Position of the goal state can be found with the code follows: " *mh.find_pos(maze, what* = *"G")*.
- Cost: An action cost function, denoted by ACTION-COST(s,a, s') when we are programming Action cost function or c(s,a, s') when we are doing math, that gives the numeric cost of applying action a in state s to reach state s'. A problem-solving agent should use a cost function that reflects its own performance measure; for example, for route-finding agents, the cost of an action might be the length, or it might be the time it takes to complete the action (Russel p. 65). For this application, every action has unit cost 1. Thus, shortest path (requires least action) is the best path if we decide using cost as the performance measure.

In [7]:
```python
print("Start location:", mh.find_pos(maze, what = "S"))
print("Goal location:", mh.find_pos(maze, what = "G"))
```

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

Give some estimates for the problem size:

Problem Sizes

- n State Space Size: State space size can be roughly estimated by the sizes of the maze. If we consider shape of the maze is a rectangular with A and B, the state space size of the problem is approximately AxB. There are unavailable states in this size (maze walls). However, this is a useful apporach to get this size value.
- b maximum branching factor: For this problem it is four. It can take following 4 actions: N,W,E,S.
- m: maximum depth of tree: For NxN maze I assumed it as 2*(N-1)
- d: depth of the optimal solution: It is equal to the Manhattan Distance heuristic.

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.

Uninformed Search Strategies

1. BFS Algorithm:

In [8]:
```python
# Your code goes here

from collections import deque

frontier_size_max = 0
tree_depth_max = 0
number_of_nodes_expand = 0
visited = []

class Node:
    def __init__(self, pos, parent, action, cost):
        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 has None)
        self.cost = cost         # for uniform cost this is the depth. It is also g(n) for A* search

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

#Potential Child Node Extraction function:
def ChildInfo(NodeInfo, MazeInfo, frontier):
    childrenList = []


    #Describe Position Change
    PosE = NodeInfo.pos[0] + 1, NodeInfo.pos[1]
    PosW = NodeInfo.pos[0] - 1, NodeInfo.pos[1]
```

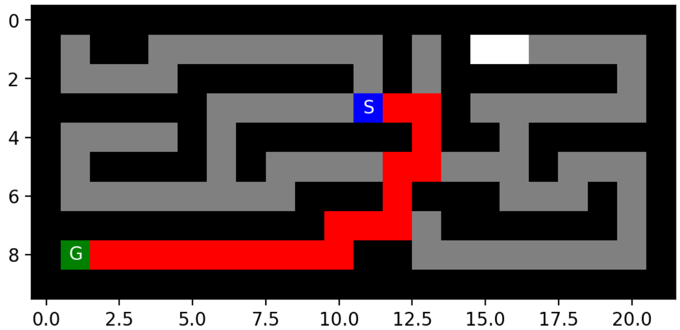
```python
    PosN = NodeInfo.pos[0], NodeInfo.pos[1] - 1
    PosS = NodeInfo.pos[0], NodeInfo.pos[1] + 1


    #extract potential nodes
    if mh.look(MazeInfo, PosE) != 'X' and not PosE in frontier:
        newChild = Node(pos = PosE, parent = NodeInfo, action = "E", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosW) != 'X' and not PosW in frontier:
        newChild = Node(pos = PosW, parent = NodeInfo, action = "W", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosN) != 'X' and not PosN in frontier:
        newChild = Node(pos = PosN, parent = NodeInfo, action = "N", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosS) != 'X' and not PosS in frontier:
        newChild = Node(pos = PosS, parent = NodeInfo, action = "S", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    return childrenList

#BFS Initialization:
def BFS_implemented(maze, root):
    global frontier_size_max
    global tree_depth_max
    global number_of_nodes_expand
    frontier_size_max = 0
    tree_depth_max = 0
    number_of_nodes_expand = 0
    que = deque() # frontier as a queue
    global visited
    visited = []
    que.append(root) #queue intilization


    while ( len(que) > 0 ): #while frontier/stack ISNAN empty
        currNode = que.popleft() #node to pop(frontier)
        visited.append(currNode.pos) #updated
        childrenList = ChildInfo(currNode, maze, visited)
        number_of_nodes_expand = number_of_nodes_expand + 1
        if currNode.cost > tree_depth_max:
            tree_depth_max = currNode.cost
        if len(que) > frontier_size_max:
            frontier_size_max = len(que)
        for i in childrenList:
            currentChild = i
            if mh.look(maze, currentChild.pos) == 'G':
                return currentChild
            if not (currentChild.pos in visited):
                que.append(currentChild)
                visited.append(currentChild.pos)
    print('failed')
    return root
```

In [9]:

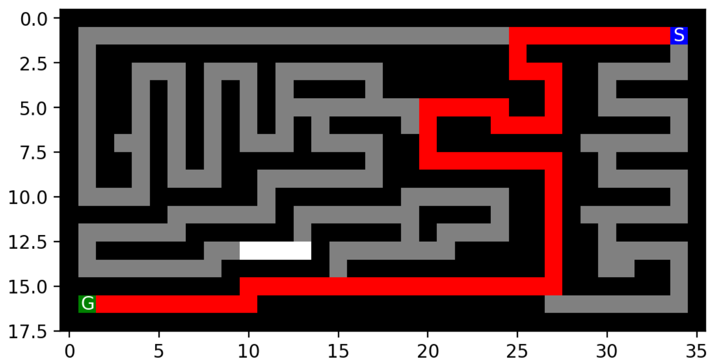
```python
import pandas as pd
#Table Making
column_display=['MazeID', 'Cost', 'Max Frontier Size', 'Visited Nodes Number', 'Max Tree depth']
data_out = []
df = pd.DataFrame(data_out, columns=column_display)
MazefileNames = ["small_maze.txt", "medium_maze.txt", "large_maze.txt", "open_maze.txt", "wall_maze.txt", "loops_maze.txt", "empty_maze.txt", "empty_2_maze.tx
verbose = True

#BFS initialize
for i in MazefileNames:
    #parse maze:
    f = open(i, "r")
    maze_ID_txt = f.read()
    maze = mh.parse_maze(maze_ID_txt)


    #DFS
    statrting_position = mh.find_pos(maze, what="S")
    root = Node(pos=statrting_position, parent=None, action=None, cost=0)  # make root node
    goalNode_curr = BFS_implemented(maze, root)
    max_frontier_curr = frontier_size_max
    num_nodes_curr = number_of_nodes_expand
    tree_depth_max_curr = tree_depth_max


    #Path
    List = []
    temp = goalNode_curr
    while temp != root:
        List.append(temp.pos)
        temp = temp.parent
    #update maze for displaying
    for x in List:
        maze[x[0]][x[1]] = "P"
    for x in visited:
        if(x not in List):
            maze[x[0]][x[1]] = "."
    maze[goalNode_curr.pos[0]][goalNode_curr.pos[1]] = "G"
    maze[statrting_position[0]][statrting_position[1]] = "S"
```

```
    fileName = i
    if verbose:
        print(fileName, ':')
        print('Solution Node: ', goalNode_curr)
        mh.show_maze(maze)

    #append to df
    data = [fileName, goalNode_curr.cost, max_frontier_curr, num_nodes_curr, tree_depth_max_curr]
    df.loc[len(df.index)] = data

BFS_data = df
```

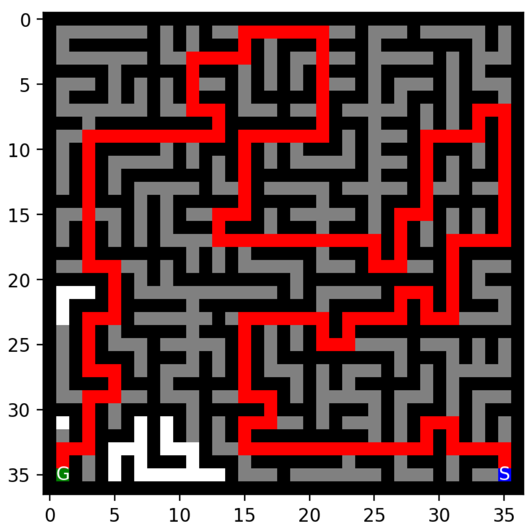
```
small_maze.txt :
Solution Node:  Node - pos = (8, 1); parent = <__main__.Node object at 0x000001E75A758BB0>; action = N; cost = 19
```



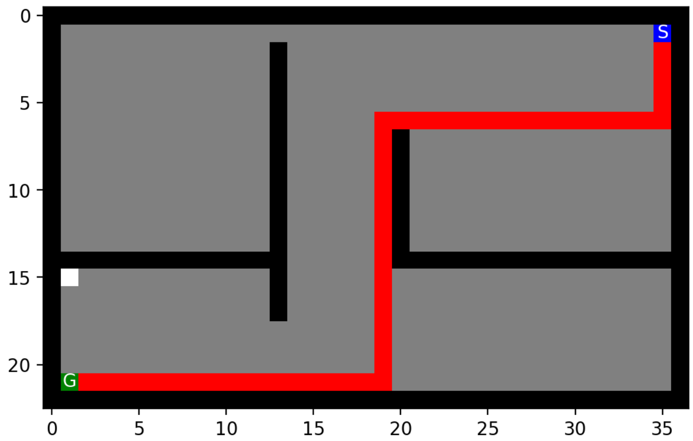
```
medium_maze.txt :
Solution Node:  Node - pos = (16, 1); parent = <__main__.Node object at 0x000001E75A782DD0>; action = N; cost = 68
```



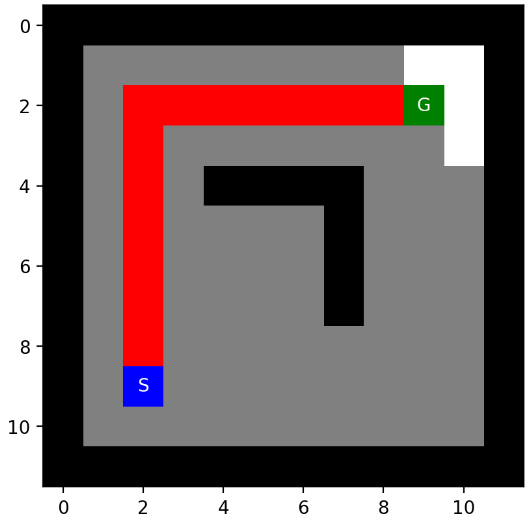
```
large_maze.txt :
Solution Node:  Node - pos = (35, 1); parent = <__main__.Node object at 0x000001E75AF8B580>; action = E; cost = 210
```



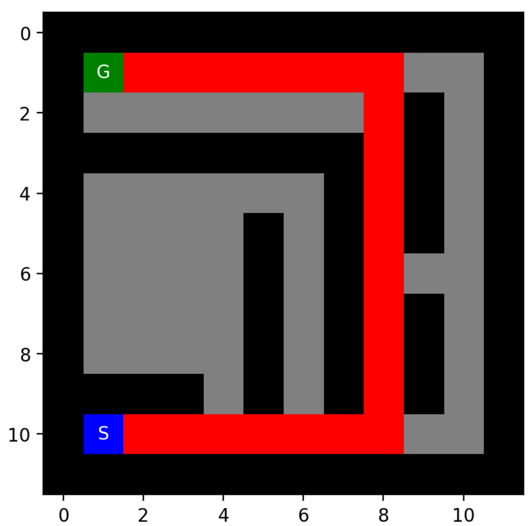
```
open_maze.txt :
Solution Node:  Node - pos = (21, 1); parent = <__main__.Node object at 0x000001E75AFCF760>; action = N; cost = 54
```



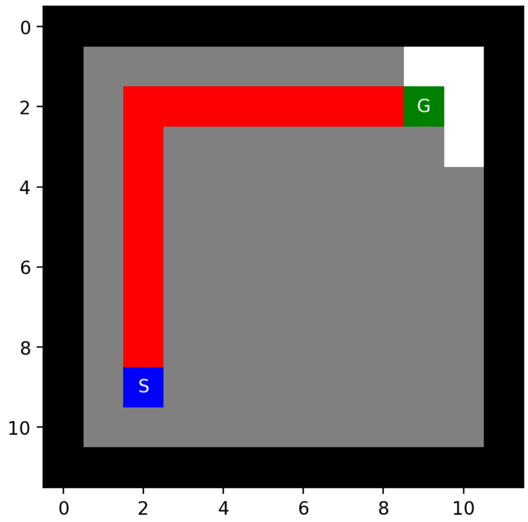
wall_maze.txt :
Solution Node: Node - pos = (2, 9); parent = <__main__.Node object at 0x000001E75A7E1390>; action = S; cost = 14



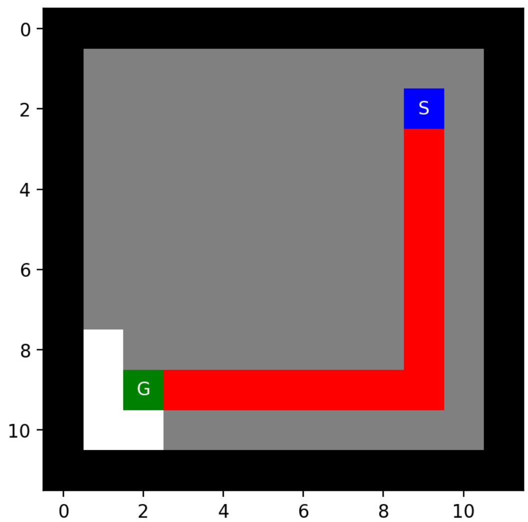
loops_maze.txt :
Solution Node: Node - pos = (1, 1); parent = <__main__.Node object at 0x000001E75C5E3E80>; action = N; cost = 23



empty_maze.txt :
Solution Node: Node - pos = (2, 9); parent = <__main__.Node object at 0x000001E75C61F2E0>; action = S; cost = 14




```
empty_2_maze.txt :
Solution Node:  Node - pos = (9, 2); parent = <__main__.Node object at 0x000001E75C68DA80>; action = N; cost = 14
```



2. DFS Algorithm

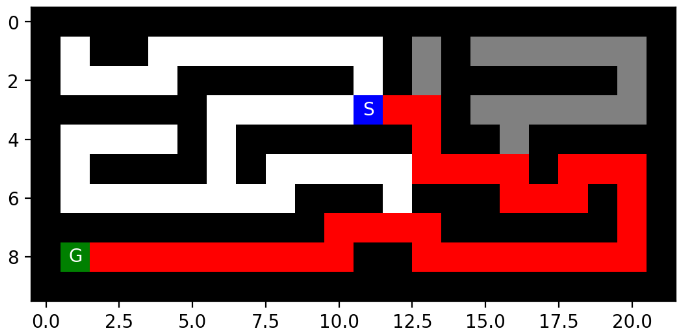
In [10]:

```
frontier_size_max = 0
tree_depth_max = 0
number_of_nodes_expand = 0
visited = []

class Node:
    def __init__(self, pos, parent, action, cost):
        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 has None)
        self.cost = cost         # for uniform cost this is the depth. It is also g(n) for A* search

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

#helper function returning list of valid potential child nodes:
def ChildInfo(NodeInfo, MazeInfo, frontier):
    childrenList = []
    #pass in a node, and return a list of a valid child arrays
    PosE = NodeInfo.pos[0] + 1, NodeInfo.pos[1]
    PosW = NodeInfo.pos[0] - 1, NodeInfo.pos[1]
    PosN = NodeInfo.pos[0], NodeInfo.pos[1] - 1
    PosS = NodeInfo.pos[0], NodeInfo.pos[1] + 1
    #look at all potential spaces
    if mh.look(MazeInfo, PosE) != 'X' and not PosE in frontier:
        newChild = Node(pos = PosE, parent = NodeInfo, action = "E", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosW) != 'X' and not PosW in frontier:
        newChild = Node(pos = PosW, parent = NodeInfo, action = "W", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosN) != 'X' and not PosN in frontier:
        newChild = Node(pos = PosN, parent = NodeInfo, action = "N", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosS) != 'X' and not PosS in frontier:
        newChild = Node(pos = PosS, parent = NodeInfo, action = "S", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    #return the actual child list
```

```python
    return childrenList


#DFS
def DFS_implement(maze, root):
    #variable intilization:
    global frontier_size_max
    global tree_depth_max
    global number_of_nodes_expand
    frontier_size_max = 0
    tree_depth_max = 0
    number_of_nodes_expand = 0
    stack = []
    global visited
    visited = []
    stack.append(root) #stack intilization
    currNode = root
    #DFS:
    while ( len(stack) > 0 ): #while frontier/stack is not empty
        prevNode = currNode
        currNode = stack.pop()
        number_of_nodes_expand = number_of_nodes_expand + 1
        if currNode.cost < prevNode.cost: #DFS USES LESS MEMORY BY RELEASING NODES NOT NEEDED ANYMORE
            prevNode = None
        visited.append(currNode.pos)
        if currNode.cost > tree_depth_max:
            tree_depth_max = currNode.cost
        if len(stack) > frontier_size_max:
            frontier_size_max = len(stack)
        if mh.look(maze, currNode.pos) == 'G': #GOAL STATE FOUND
            return currNode
        #potential children
        childrenList = ChildInfo(currNode, maze, visited)
        for i in childrenList:
            if not (i in stack):
                stack.append(i)
    print('failed')
    return root
```

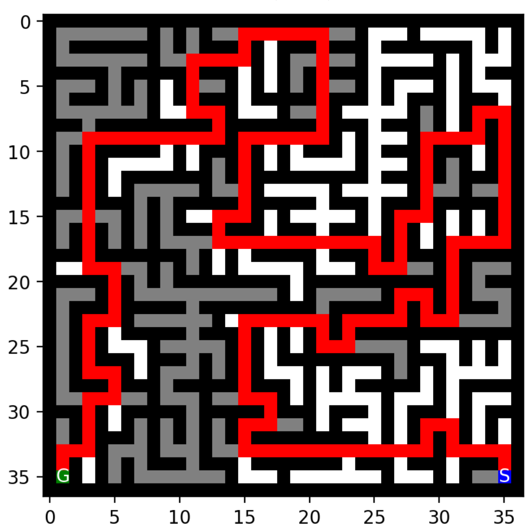
In [11]:
```python
import pandas as pd
#use pandas to dispaly as table
column_display=['MazeID', 'Cost', 'Max Frontier Size', 'Visited Nodes Number', 'Max Tree depth']
data_out = []
df = pd.DataFrame(data_out, columns=column_display)
MazefileNames = ["small_maze.txt", "medium_maze.txt", "large_maze.txt", "open_maze.txt", "wall_maze.txt", "loops_maze.txt", "empty_maze.txt", "empty_2_maze.tx
verbose = True

#do DFS
for i in MazefileNames:
    #parse maze:
    f = open(i, "r")
    maze_ID_txt = f.read()
    maze = mh.parse_maze(maze_ID_txt)

    statrting_position = mh.find_pos(maze, what="S")
    root = Node(pos=statrting_position, parent=None, action=None, cost=0)  # make root node
    goalNode_curr = DFS_implement(maze, root)
    max_frontier_curr = frontier_size_max
    num_nodes_curr = number_of_nodes_expand
    tree_depth_max_curr = tree_depth_max
    #Path
    List = []
    temp = goalNode_curr
    while temp != root:
        List.append(temp.pos)
        temp = temp.parent

    for x in List:
        maze[x[0]][x[1]] = "P"
    for x in visited:
        if(x not in List):
            maze[x[0]][x[1]] = "."
    maze[goalNode_curr.pos[0]][goalNode_curr.pos[1]] = "G"
    maze[statrting_position[0]][statrting_position[1]] = "S"

    fileName = i
    if verbose:
        print(fileName, ':')
        print('Solution Node: ', goalNode_curr)
        mh.show_maze(maze)

    #append data to the dataframe
    data = [fileName, goalNode_curr.cost, max_frontier_curr, num_nodes_curr, tree_depth_max_curr]
    df.loc[len(df.index)] = data

DFS_data = df
```

```
small_maze.txt :
Solution Node:  Node - pos = (8, 1); parent = <__main__.Node object at 0x000001E75A8CB130>; action = N; cost = 37
```

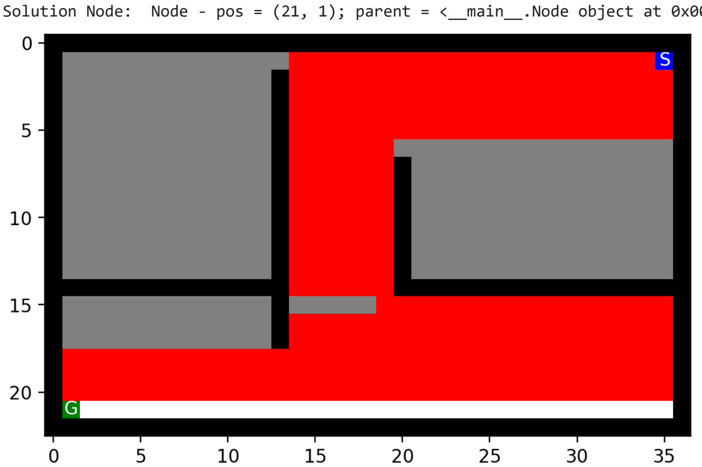


medium_maze.txt :
Solution Node: Node - pos = (16, 1); parent = <__main__.Node object at 0x000001E75AC11750>; action = N; cost = 130

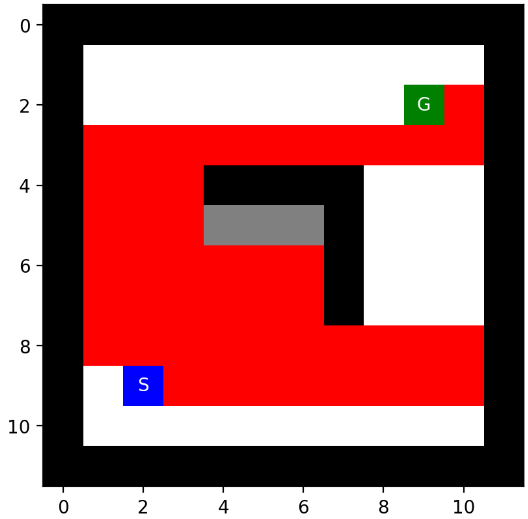
large_maze.txt :
Solution Node: Node - pos = (35, 1); parent = <__main__.Node object at 0x000001E75A86ABF0>; action = E; cost = 210



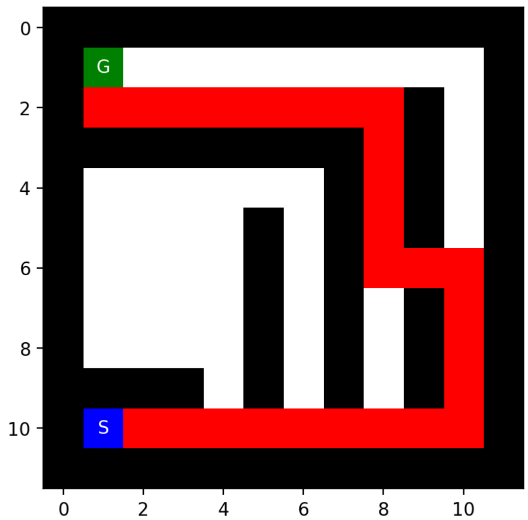
open_maze.txt :
Solution Node: Node - pos = (21, 1); parent = <__main__.Node object at 0x000001E75C5B65F0>; action = E; cost = 330



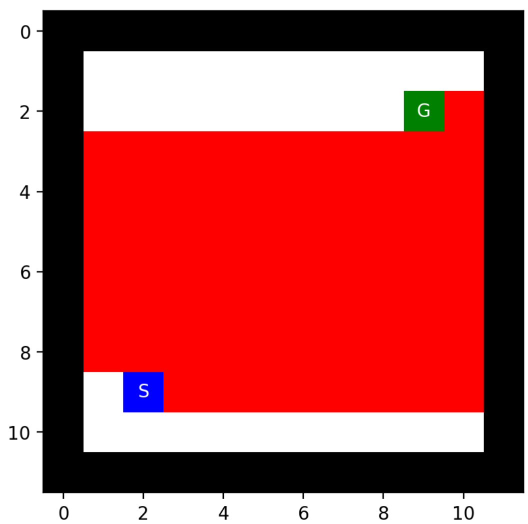
wall_maze.txt :
Solution Node: Node - pos = (2, 9); parent = <__main__.Node object at 0x000001E75C5B5930>; action = N; cost = 48



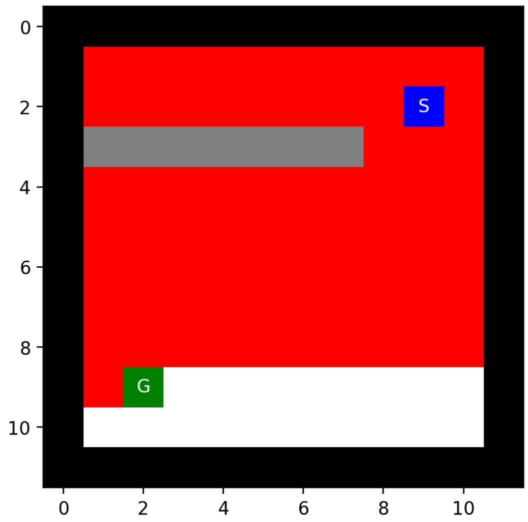
loops_maze.txt :
Solution Node: Node - pos = (1, 1); parent = <__main__.Node object at 0x000001E75AFEA1A0>; action = W; cost = 27



empty_maze.txt :
Solution Node: Node - pos = (2, 9); parent = <__main__.Node object at 0x000001E75A829E70>; action = N; cost = 70



empty_2_maze.txt :
Solution Node: Node - pos = (9, 2); parent = <__main__.Node object at 0x000001E75A78FA30>; action = S; cost = 74



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

Cycles are a big problem while performing a BFS and DFS Algorithms since they can cause infinite loops. To deal with that, a ChildInfo function is written. When looking/evaluating a new space, we check that the space is not in the frontier (if it was previously visited). Then, we do not visit duplicate spaces.

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

The implementations of BFS and DFS Algorithms are complete since they always reach the goal state if a solution exists (See the figures).

DFS is not optimal because it returns the first solution it finds and that solution is not always the best solution. The search can return a solution with the best path cost, but it just returns the first solution it finds to reach the goal state. We tried making the space complexity optimal by deleting parent nodes of incorrect paths (delete the whole tree and free the allocated memory). Complexity of our DFS implementation is O(b^m) where m = max depth and b = branches.

BFS Algorithm, is complete because the FIFO of a queue as the frontier expands every single path possible to find the optimal solution. Also, every step has unit cost, which makes the implementation optimal. For our implementation, space complexity is O(b^d*r) as well as storing the reached nodes. Time complexity for BFS is O(b^d).

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).

Informed search

1. Greedy Best-First Search Algorithm

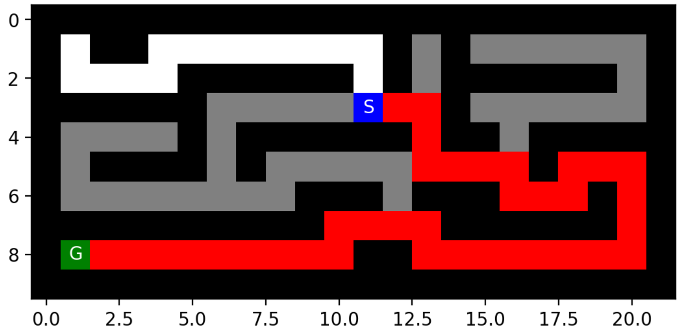
In [12]:
```
def Heuristic(node, maze): #calculate Manhattan distance and return integer value representation
    #Manhattan distance = x_distance + y_distance
    goal_pos = mh.find_pos(maze, what="G")
    x_val = abs (goal_pos[0] - node.pos[0])
    y_val = abs (goal_pos[1] - node.pos[1])
    return (x_val + y_val)
from collections import deque
frontier_size_max = 0
tree_depth_max = 0
number_of_nodes_expand = 0
visited = []

class Node:
    def __init__(self, pos, parent, action, cost):
        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 has None)
        self.cost = cost         # for uniform cost this is the depth. It is also g(n) for A* search

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


def ChildInfo(NodeInfo, MazeInfo, frontier):
    childrenList = []
    PosE = NodeInfo.pos[0] + 1, NodeInfo.pos[1]
    PosW = NodeInfo.pos[0] - 1, NodeInfo.pos[1]
    PosN = NodeInfo.pos[0], NodeInfo.pos[1] - 1
    PosS = NodeInfo.pos[0], NodeInfo.pos[1] + 1
    #look at all potential spaces
    if mh.look(MazeInfo, PosE) != 'X' and not PosE in frontier:
        newChild = Node(pos = PosE, parent = NodeInfo, action = "E", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosW) != 'X' and not PosW in frontier:
```

```python
        newChild = Node(pos = PosW, parent = NodeInfo, action = "W", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosN) != 'X' and not PosN in frontier:
        newChild = Node(pos = PosN, parent = NodeInfo, action = "N", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosS) != 'X' and not PosS in frontier:
        newChild = Node(pos = PosS, parent = NodeInfo, action = "S", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    return childrenList


#greedy Best
def greedyBestSearch(maze, root):
    global frontier_size_max
    global tree_depth_max
    global number_of_nodes_expand
    frontier_size_max = 0
    tree_depth_max = 0
    number_of_nodes_expand = 0
    que = []
    global visited
    visited = []
    que.append(root)  # stack intilization
    while (len(que) > 0):  # while frontier/stack ISNAN empty
        currNode = que.pop()
        visited.append(currNode.pos)

        number_of_nodes_expand = number_of_nodes_expand + 1
        if number_of_nodes_expand >= 20000:
            return root
        if currNode.cost > tree_depth_max:
            tree_depth_max = currNode.cost
        if len(que) > frontier_size_max:
            frontier_size_max = len(que)

        if mh.look(maze, currNode.pos) == 'G':  # GOAL STATE FOUND
            return currNode
        childrenList = ChildInfo(currNode, maze, visited)
        #  HEURISTIC
        List = []  # heuristic value, node
        for eachNode in childrenList:
            Greedy_Value = Heuristic(eachNode, maze)
            my_pair = eachNode, Greedy_Value
            List.append(my_pair)
        # sorting in ascending order
        List.sort(key=lambda x: x[1])
        for x in List:
            if not (x[0] in que):
                que.append(x[0])

    print('failed')
    return root
```

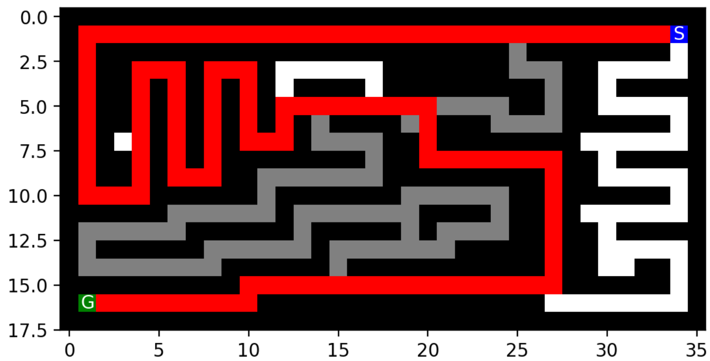
In [13]:
```python
import pandas as pd
column_display=['MazeID', 'Cost', 'Max Frontier Size', 'Visited Nodes Number', 'Max Tree depth']
data_out = []
df = pd.DataFrame(data_out, columns=column_display)
MazefileNames = ["small_maze.txt", "medium_maze.txt", "large_maze.txt", "open_maze.txt", "wall_maze.txt", "loops_maze.txt", "empty_maze.txt", "empty_2_maze.tx
verbose = True

# GBS
for i in MazefileNames:
    #parse maze:
    f = open(i, "r")
    maze_ID_txt = f.read()
    maze = mh.parse_maze(maze_ID_txt)
    #perform DFS and get stats:
    statrting_position = mh.find_pos(maze, what="S")
    root = Node(pos=statrting_position, parent=None, action=None, cost=0)  # make root node
    goalNode_curr = greedyBestSearch(maze, root)
    max_frontier_curr = frontier_size_max
    num_nodes_curr = number_of_nodes_expand
    tree_depth_max_curr = tree_depth_max
    #  path
    List = []
    temp = goalNode_curr
    while temp != root:
        List.append(temp.pos)
        temp = temp.parent
    for x in List:
        maze[x[0]][x[1]] = "P"
    for x in visited:
        if(x not in List):
            maze[x[0]][x[1]] = "."
    maze[goalNode_curr.pos[0]][goalNode_curr.pos[1]] = "G"
    maze[statrting_position[0]][statrting_position[1]] = "S"

    fileName = i
    if verbose:
        print(fileName, ':')
        print('Solution Node: ', goalNode_curr)
```

```
        mh.show_maze(maze)

    #append data to dataframe
    data = [fileName, goalNode_curr.cost, max_frontier_curr, num_nodes_curr, tree_depth_max_curr]
    df.loc[len(df.index)] = data

greedy_data = df
```

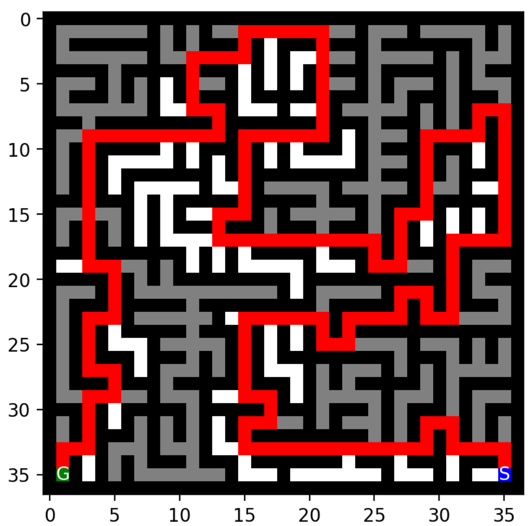
```
small_maze.txt :
Solution Node:  Node - pos = (8, 1); parent = <__main__.Node object at 0x000001E75C6806D0>; action = N; cost = 37
```



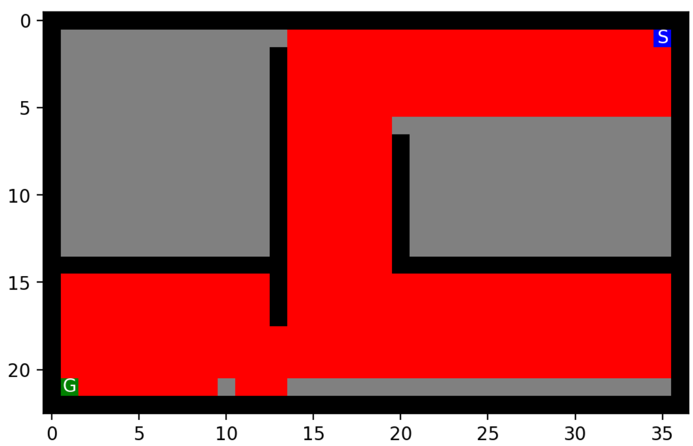

```
medium_maze.txt :
Solution Node:  Node - pos = (16, 1); parent = <__main__.Node object at 0x000001E75B04DFC0>; action = N; cost = 130
```



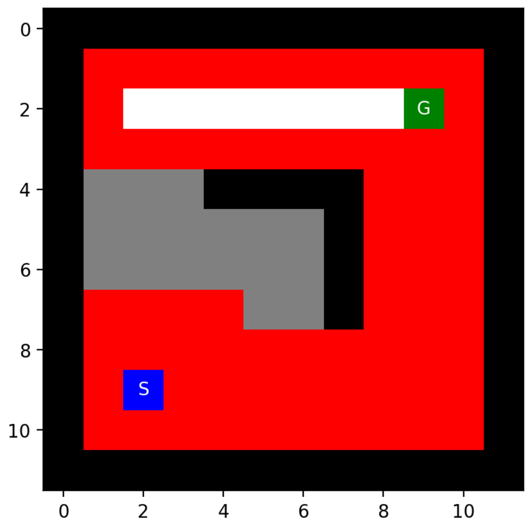

```
large_maze.txt :
Solution Node:  Node - pos = (35, 1); parent = <__main__.Node object at 0x000001E75AC335E0>; action = E; cost = 210
```



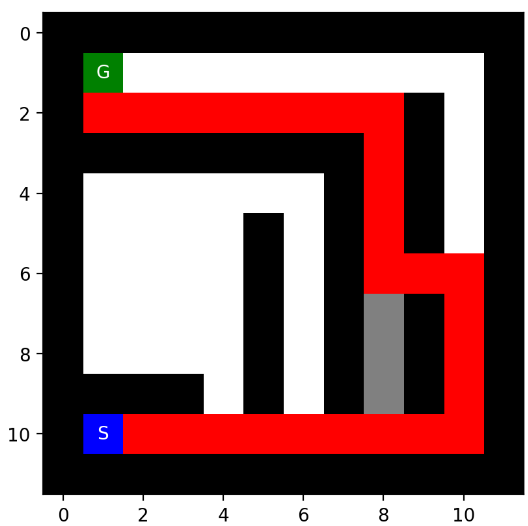

```
open_maze.txt :
Solution Node:  Node - pos = (21, 1); parent = <__main__.Node object at 0x000001E75ACC11B0>; action = N; cost = 382
```



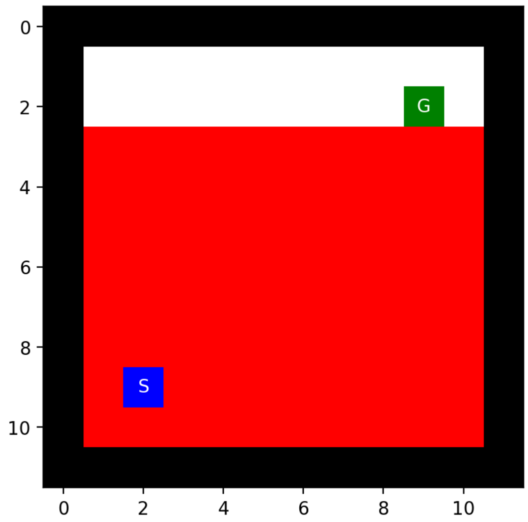

```
wall_maze.txt :
Solution Node:  Node - pos = (2, 9); parent = <__main__.Node object at 0x000001E75ACC1FC0>; action = N; cost = 68
```



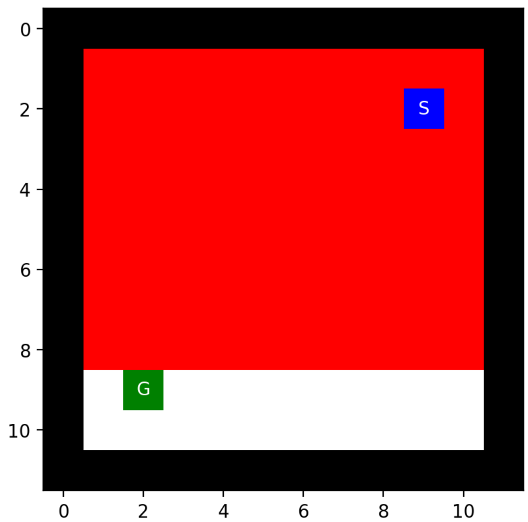

```
loops_maze.txt :
Solution Node:  Node - pos = (1, 1); parent = <__main__.Node object at 0x000001E75B05A470>; action = W; cost = 27
```




```
empty_maze.txt :
Solution Node:  Node - pos = (2, 9); parent = <__main__.Node object at 0x000001E75C6240A0>; action = W; cost = 80
```




```
empty_2_maze.txt :
Solution Node:  Node - pos = (9, 2); parent = <__main__.Node object at 0x000001E75A78C040>; action = E; cost = 80
```



2. A* Algorithm

In [14]:

```python
def AStarHeuristic(node, maze): #calculate A* heuristic

    # g(n) = Cost to current node (get cost from root)
    # h(n) = Manhattan Distance
    #g(n)+h(n)=heuristic
    goal_pos = mh.find_pos(maze, what="G")
    x_val = abs (goal_pos[0] - node.pos[0])
    y_val = abs (goal_pos[1] - node.pos[1])
    h = x_val + y_val #h(n)
    #g(n)+h(n) =
    g = node.cost
    temp = node
    while temp != root:
        temp = temp.parent
        g = g + temp.cost

    #get h(n)
    h = g + h
    return h

from collections import deque
frontier_size_max = 0
tree_depth_max = 0
number_of_nodes_expand = 0
visited = []

class Node:
    def __init__(self, pos, parent, action, cost):
        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 has None)
        self.cost = cost         # for uniform cost this is the depth. It is also g(n) for A* search

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

def ChildInfo(NodeInfo, MazeInfo, frontier):
```

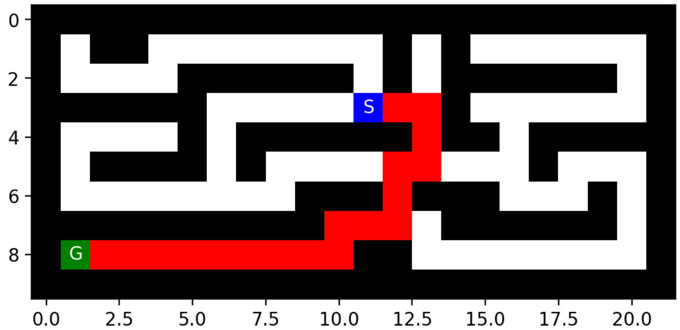
```
    childrenList = []
    PosE = NodeInfo.pos[0] + 1, NodeInfo.pos[1]
    PosW = NodeInfo.pos[0] - 1, NodeInfo.pos[1]
    PosN = NodeInfo.pos[0], NodeInfo.pos[1] - 1
    PosS = NodeInfo.pos[0], NodeInfo.pos[1] + 1
    if mh.look(MazeInfo, PosE) != 'X' and not PosE in frontier:
        newChild = Node(pos = PosE, parent = NodeInfo, action = "E", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosW) != 'X' and not PosW in frontier:
        newChild = Node(pos = PosW, parent = NodeInfo, action = "W", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosN) != 'X' and not PosN in frontier:
        newChild = Node(pos = PosN, parent = NodeInfo, action = "N", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosS) != 'X' and not PosS in frontier:
        newChild = Node(pos = PosS, parent = NodeInfo, action = "S", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    return childrenList


#a* initialization
def astarSearch(maze, root):
    global frontier_size_max
    global tree_depth_max
    global number_of_nodes_expand
    frontier_size_max = 0
    tree_depth_max = 0
    number_of_nodes_expand = 0

    que = deque()  # frontier as a queue
    reached = []
    que.append(root)  # queue intilization
    while (len(que) > 0):  # while frontier/stack is not empty
        currNode = que.popleft()  # node -> pop(frontier)
        reached.append(currNode.pos)  # updated reached
        childrenList = ChildInfo(currNode, maze, reached)
        number_of_nodes_expand = number_of_nodes_expand + 1
        if currNode.cost > tree_depth_max:
            tree_depth_max = currNode.cost
        if len(que) > frontier_size_max:
            frontier_size_max = len(que)
        List = []  # heuristic value, node
        for eachNode in childrenList:
            huer_val = AStarHeuristic(eachNode, maze)
            my_pair = eachNode, huer_val
            List.append(my_pair)
        for i in List:
            currentChild = i[0]
            if mh.look(maze, currentChild.pos) == 'G':
                return currentChild
            if not (currentChild.pos in reached):
                que.append(currentChild)
                reached.append(currentChild.pos)
    print('failed')
    return root
```

In [15]:

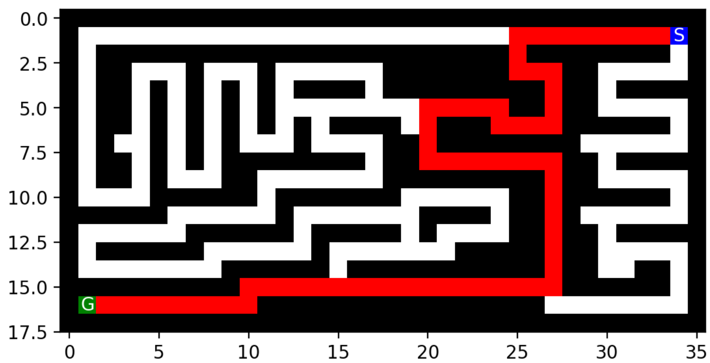
```
import pandas as pd
column_display=['MazeID', 'Cost', 'Max Frontier Size', 'Visited Nodes Number', 'Max Tree depth']
data_out = []
df = pd.DataFrame(data_out, columns=column_display)
MazefileNames = ["small_maze.txt", "medium_maze.txt", "large_maze.txt", "open_maze.txt", "wall_maze.txt", "loops_maze.txt", "empty_maze.txt", "empty_2_maze.tx
verbose = True

for i in MazefileNames:
    #parse maze:
    f = open(i, "r")
    maze_ID_txt = f.read()
    maze = mh.parse_maze(maze_ID_txt)
    #perform DFS and get stats:
    statrting_position = mh.find_pos(maze, what="S")
    root = Node(pos=statrting_position, parent=None, action=None, cost=0)  # make root node

    goalNode_curr = astarSearch(maze, root)
    max_frontier_curr = frontier_size_max

    num_nodes_curr = number_of_nodes_expand
    tree_depth_max_curr = tree_depth_max

    #  path
    List = []
    temp = goalNode_curr
    while temp != root:
        List.append(temp.pos)
        temp = temp.parent
    #update maze for displaying
    for x in List:
        maze[x[0]][x[1]] = "P"
    for x in visited:
        if(x not in List):
            maze[x[0]][x[1]] = "."
    maze[goalNode_curr.pos[0]][goalNode_curr.pos[1]] = "G"
```

```
    maze[statrting_position[0]][statrting_position[1]] = "S"

    #ouptut solution data:
    fileName = i
    if verbose:
        print(fileName, ':')
        print('Solution Node: ', goalNode_curr)
        mh.show_maze(maze)

    data = [fileName, goalNode_curr.cost, max_frontier_curr, num_nodes_curr, tree_depth_max_curr]
    df.loc[len(df.index)] = data

astar_data = df
```

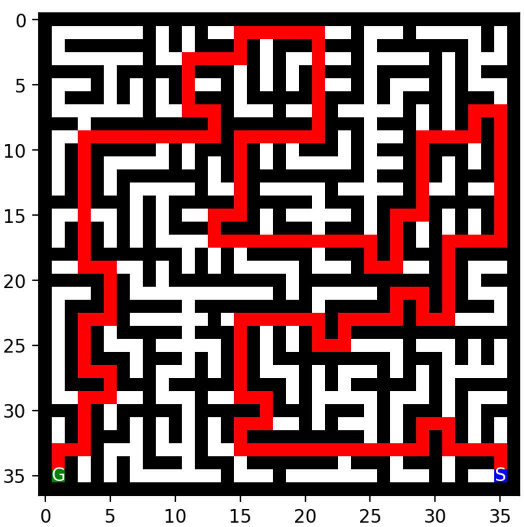
```
small_maze.txt :
Solution Node:  Node - pos = (8, 1); parent = <__main__.Node object at 0x000001E75AFCDFF0>; action = N; cost = 19
```



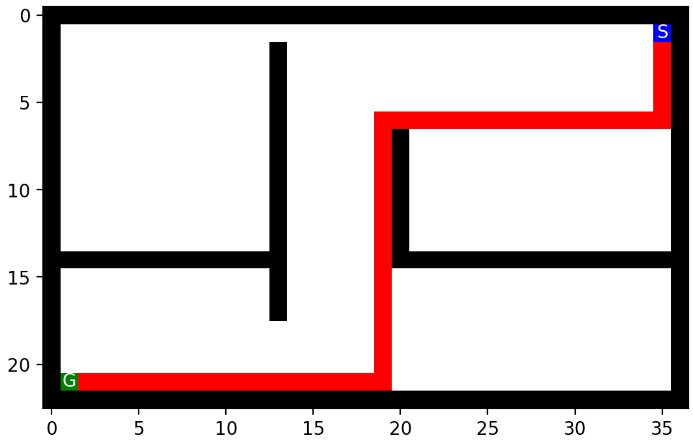

```
medium_maze.txt :
Solution Node:  Node - pos = (16, 1); parent = <__main__.Node object at 0x000001E75A82A590>; action = N; cost = 68
```



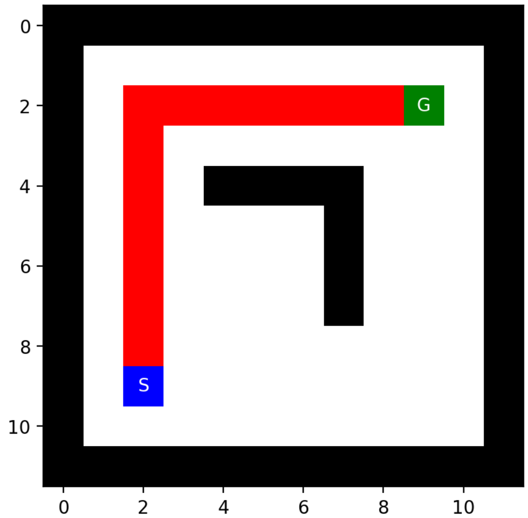

```
large_maze.txt :
Solution Node:  Node - pos = (35, 1); parent = <__main__.Node object at 0x000001E75A8A5780>; action = E; cost = 210
```



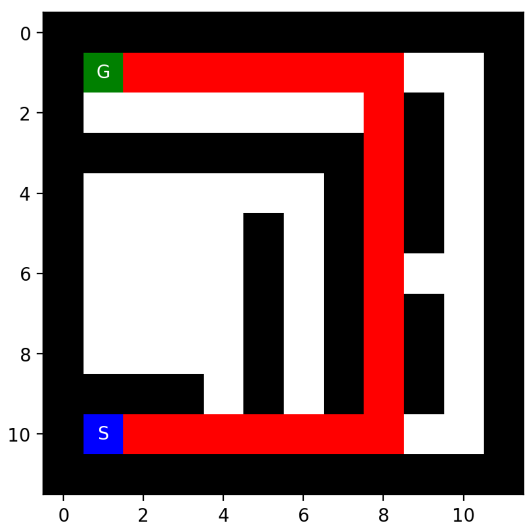

```
open_maze.txt :
Solution Node:  Node - pos = (21, 1); parent = <__main__.Node object at 0x000001E75C65C3A0>; action = N; cost = 54
```



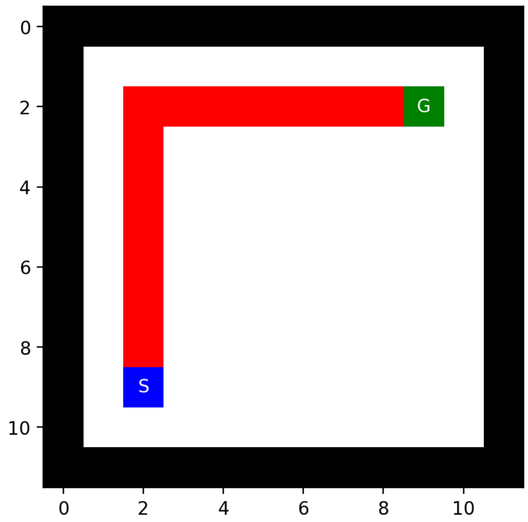
wall_maze.txt :
Solution Node: Node - pos = (2, 9); parent = <__main__.Node object at 0x000001E75C5FA6B0>; action = S; cost = 14



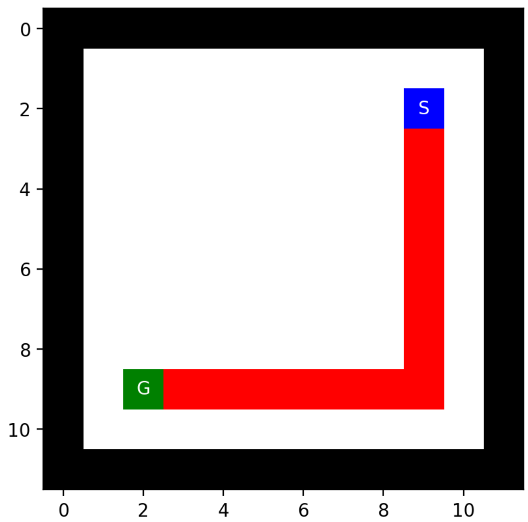
loops_maze.txt :
Solution Node: Node - pos = (1, 1); parent = <__main__.Node object at 0x000001E759FDDA50>; action = N; cost = 23



empty_maze.txt :
Solution Node: Node - pos = (2, 9); parent = <__main__.Node object at 0x000001E75B037700>; action = S; cost = 14




```
empty_2_maze.txt :
Solution Node:  Node - pos = (9, 2); parent = <__main__.Node object at 0x000001E75ACC15A0>; action = N; cost = 14
```



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

Discussion

Greedy Best First Search is complete. If there is a solution it can find it. It could also find the solution in our implementation. Greedy Best First Search is not optimal. It searches for a solution by expanding the "first the node with the lowest value h(n)," where h(n) is the heuristic function. Heuristic function might be misleading some cases and may not return the shortest path available (This is where A* moves in!). Complexity is the same as BFS: O(b^m) The best case scenario time complexity is O(bm) if the heuristic is 100% accurate (It is not for our case). The reached data structure was used for stat keeping. Thus, O(b^m * r) can be given as space complexity.

A* Search is complete because this informed search algorithm is guaranteed to expand the least number of nodes and always find a solution. As seen in theory and in the results, by nature A* search algorithm is always optimal because it returns the path with the shortest path cost. BFS was also optimal since the unit cost is same for every action. A* calculates the estimated cost of any path going through any frontier node. The time complexity and space complexity of A* is the number of nodes expanded (O(2^n)).

Task 4: Comparison and discussion [20 Points]

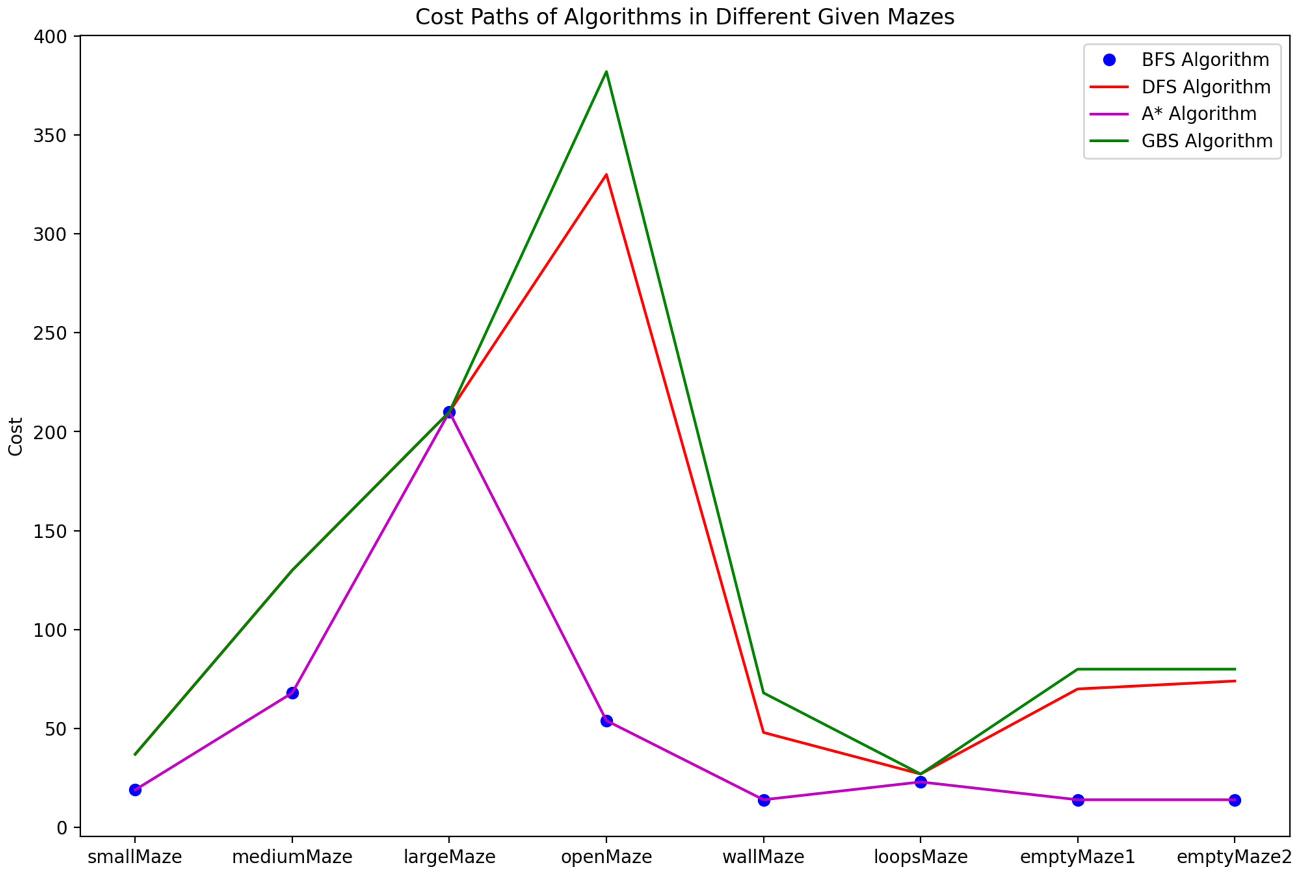
Run experiments to compare the implemented algorithms.

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 [16]:
```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
MazefileNames = ["smallMaze", "mediumMaze", "largeMaze", "openMaze", "wallMaze", "loopsMaze", "emptyMaze1", "emptyMaze2"]

fig = plt.figure(figsize=(12, 8))
plt.plot(MazefileNames, BFS_data["Cost"].to_numpy(),'bo', label='BFS Algorithm')
plt.plot(MazefileNames, DFS_data["Cost"].to_numpy(),'r', label='DFS Algorithm')
plt.plot(MazefileNames, astar_data["Cost"].to_numpy(),'m', label='A* Algorithm')
```

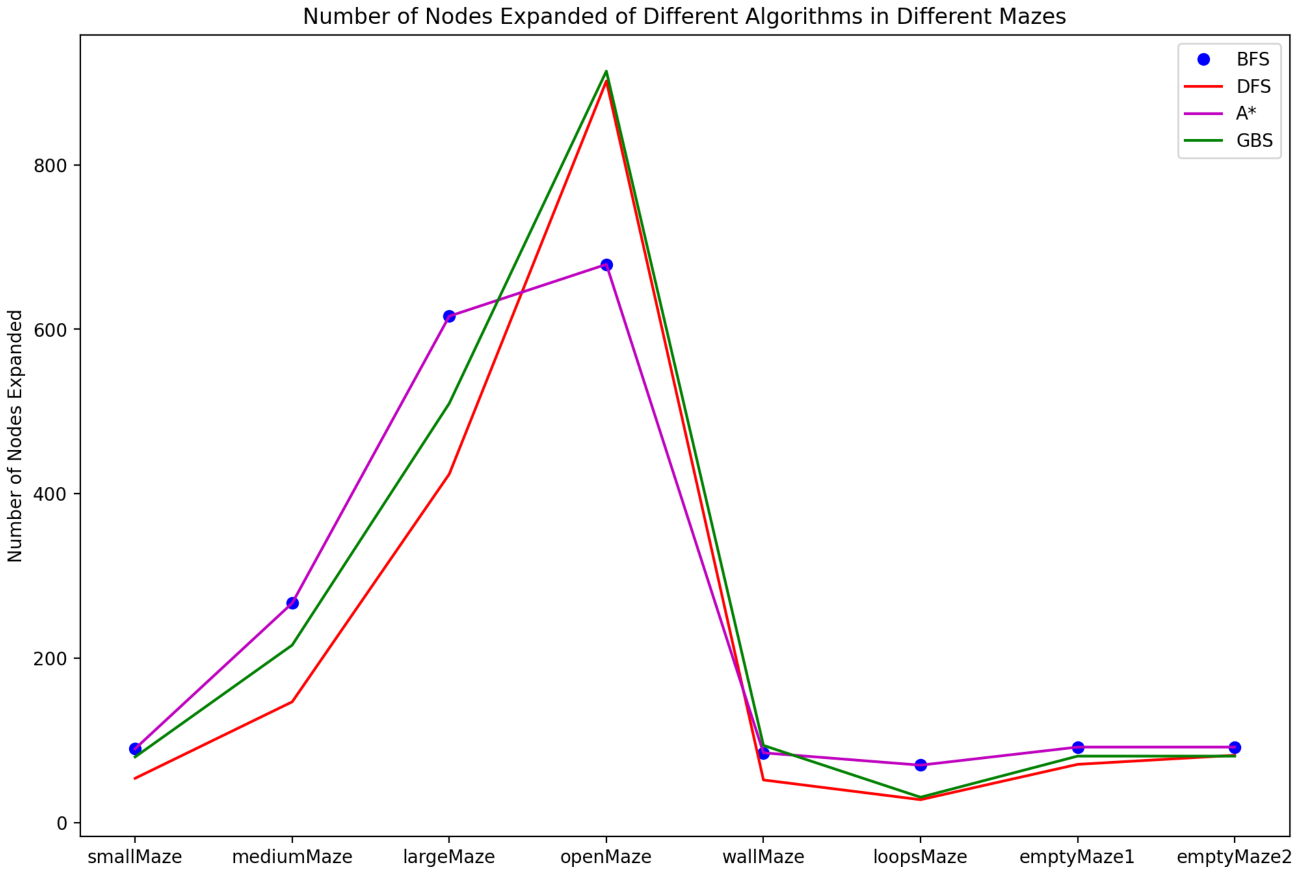
```
plt.plot(MazefileNames, greedy_data["Cost"].to_numpy(),'g', label='GBS Algorithm')
plt.ylabel('Cost')
plt.legend()
plt.title('Cost Paths of Algorithms in Different Given Mazes')
plt.show()
```



A* and BFS search algorthims had optimal and the best performance. (BFS is also optimal since every action has unit cost). GBS and DFS had similar performances which are worse than A* and BFS. In the empty and open mazes, DFS and GBS struggles a lot. Large maze has one solution for every maze.

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

In [17]:
```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
MazefileNames = ["smallMaze", "mediumMaze", "largeMaze", "openMaze", "wallMaze", "loopsMaze", "emptyMaze1", "emptyMaze2"]

fig = plt.figure(figsize=(12, 8))
plt.plot(MazefileNames, BFS_data["Visited Nodes Number"].to_numpy(),'bo', label='BFS')
plt.plot(MazefileNames, DFS_data["Visited Nodes Number"].to_numpy(),'r', label='DFS')
plt.plot(MazefileNames, astar_data["Visited Nodes Number"].to_numpy(),'m', label='A*')
plt.plot(MazefileNames, greedy_data["Visited Nodes Number"].to_numpy(),'g', label='GBS')

plt.legend()
plt.title('Number of Nodes Expanded of Different Algorithms in Different Mazes')
plt.ylabel('Number of Nodes Expanded')
plt.show()
```



Since suboptimal search algorithms focuses on returning solution quickly, they expand less nodes than both A* and BFS except for empty mazes which they struggled a lot.

In [18]:
```
BFS_data
```

Out[18]:

	MazeID	Cost	Max Frontier Size	Visited Nodes Number	Max Tree depth
0	small_maze.txt	19	7	90	18
1	medium_maze.txt	68	7	267	67
2	large_maze.txt	210	7	616	209
3	open_maze.txt	54	24	679	53
4	wall_maze.txt	14	10	85	13
5	loops_maze.txt	23	7	70	22
6	empty_maze.txt	14	11	92	13
7	empty_2_maze.txt	14	11	92	13

In [19]:
```
DFS_data
```

Out[19]:

	MazeID	Cost	Max Frontier Size	Visited Nodes Number	Max Tree depth
0	small_maze.txt	37	5	54	37
1	medium_maze.txt	130	8	147	130
2	large_maze.txt	210	37	424	222
3	open_maze.txt	330	326	902	330
4	wall_maze.txt	48	54	52	48
5	loops_maze.txt	27	12	28	27
6	empty_maze.txt	70	74	71	70
7	empty_2_maze.txt	74	73	82	74

In [20]:
```
greedy_data
```

Out[20]:

	MazeID	Cost	Max Frontier Size	Visited Nodes Number	Max Tree depth
0	small_maze.txt	37	5	80	44
1	medium_maze.txt	130	11	216	183
2	large_maze.txt	210	35	510	222
3	open_maze.txt	382	349	914	382
4	wall_maze.txt	68	58	94	72
5	loops_maze.txt	27	11	31	27
6	empty_maze.txt	80	72	81	80
7	empty_2_maze.txt	80	72	81	80

In [21]:

```
greedy_data
```

Out[21]:

	MazeID	Cost	Max Frontier Size	Visited Nodes Number	Max Tree depth
0	small_maze.txt	37	5	80	44
1	medium_maze.txt	130	11	216	183
2	large_maze.txt	210	35	510	222
3	open_maze.txt	382	349	914	382
4	wall_maze.txt	68	58	94	72
5	loops_maze.txt	27	11	31	27
6	empty_maze.txt	80	72	81	80
7	empty_2_maze.txt	80	72	81	80

In [22]:

```
astar_data
```

Out[22]:

	MazeID	Cost	Max Frontier Size	Visited Nodes Number	Max Tree depth
0	small_maze.txt	19	7	90	18
1	medium_maze.txt	68	7	267	67
2	large_maze.txt	210	7	616	209
3	open_maze.txt	54	24	679	53
4	wall_maze.txt	14	10	85	13
5	loops_maze.txt	23	7	70	22
6	empty_maze.txt	14	11	92	13
7	empty_2_maze.txt	14	11	92	13

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

Discussion

While searching example codes on the internet, implementing and debugging I understood more about each algorithm's nature. Comparing to the optimal BFS, the GBS path cost is higher but the Number of Nodes visited is lower. All of our search algorithm implementations uses a ChildInfo function which actually uses the environment info and find the available nodes. Global variables were used to increment stats such as path cost, tree depth, and frontier size. Like the algorithm in the book, the BFS used a FIFO queue for the frontier. DFS used a LIFO stack for the frontier. We used the same implementation as the BFS and DFS for the A* snd GBS. Only difference was the heuristics. The GBS heuristic was the Manhattan distance. The A* heuristic was Manhattan distance + the cost for coming to current node. I made a pandas dataframe for every algorithm, by using them in a loop dor every maze.

In the majority of cases, DFS & GBS expands less nodes than both the optimal search algorithms (this behavior is expected because DFS focus on returning the solution quickly). Also it makes sense that there is a very large spike in nodes expanded when using DFS/GBS on the empty maze because it almost randomly searches for a solution (taking a very long time). In summary, the BFS and A* search are optimal expanding more nodes on average than the DFS/GBS, which just randomly searhces for an exit point (DFS) or uses a heuristic to find the exit quickly (GBS).

Graduate student advanced task: IDS [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.

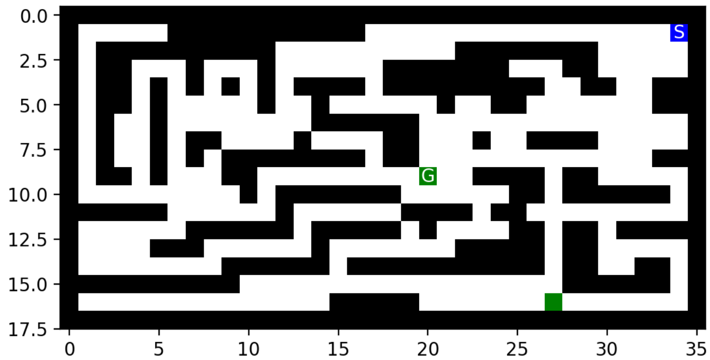
Graduate Task Mazes

In [23]:

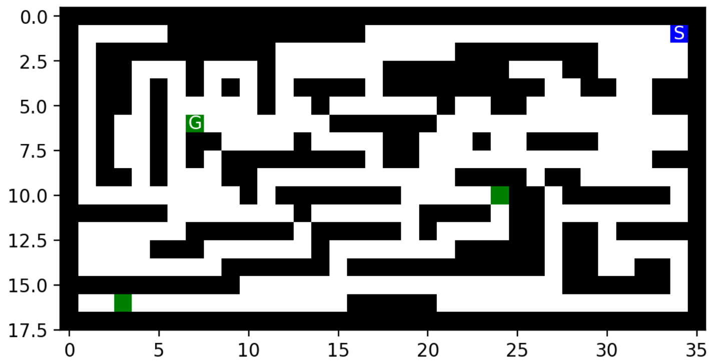
```
#DISPLAYING THE NEW MAZES:
MazefileNames = ["multi_goal_maze0.txt", "multi_goal_maze1.txt", "multi_goal_maze2.txt"]
```

```
for i in MazefileNames:
    f = open(i, "r")
    print(i, ':')
    maze_ID_txt = f.read()
    maze = mh.parse_maze(maze_ID_txt)
    mh.show_maze(maze)
```

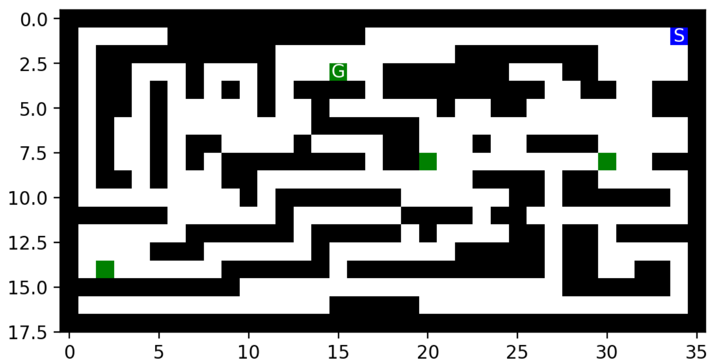
```
multi_goal_maze0.txt :
```



```
multi_goal_maze1.txt :
```



```
multi_goal_maze2.txt :
```



BFS ON NEW MAZES

In [24]:

```
import pandas as pd
#use pandas to dispaly as table
column_display=['MazeID', 'Cost', 'Max Frontier Size', 'Visited Nodes Number', 'Max Tree depth']
data_out = []
df = pd.DataFrame(data_out, columns=column_display)
MazefileNames = ["multi_goal_maze0.txt", "multi_goal_maze1.txt", "multi_goal_maze2.txt"]
verbose = True

# BFS
for i in MazefileNames:
    #parse maze:
    f = open(i, "r")
    maze_ID_txt = f.read()
    maze = mh.parse_maze(maze_ID_txt)
    #perform DFS and get stats:
    statrting_position = mh.find_pos(maze, what="S")
    root = Node(pos=statrting_position, parent=None, action=None, cost=0)  # root node
    goalNode_curr = BFS_implemented(maze, root)
    max_frontier_curr = frontier_size_max
    num_nodes_curr = number_of_nodes_expand
    tree_depth_max_curr = tree_depth_max
    #get final path
    List = []
    temp = goalNode_curr
    while temp != root:
        List.append(temp.pos)
```

```
        temp = temp.parent
    fileName = i
    if verbose:
        print(fileName, ':')
        print('Goal State: ', goalNode_curr)
        print('States on the path:', List)
        print('============================================================================')
    #append to the pandas dataframe
    data = [fileName, goalNode_curr.cost, max_frontier_curr, num_nodes_curr, tree_depth_max_curr]
    df.loc[len(df.index)] = data

BFS_multigoal = df
```

```
multi_goal_maze0.txt :
Goal State:  Node - pos = (16, 27); parent = <__main__.Node object at 0x000001E75C5CCF10>; action = E; cost = 22
States on the path: [(16, 27), (15, 27), (14, 27), (13, 27), (12, 27), (11, 27), (10, 27), (9, 27), (8, 27), (8, 28), (8, 29), (8, 30), (8, 31), (8, 32), (7, 32), (6, 32), (5, 32), (4, 32), (3, 32), (3, 33), (3, 34), (2, 34)]
============================================================================
multi_goal_maze1.txt :
Goal State:  Node - pos = (10, 24); parent = <__main__.Node object at 0x000001E75C5CD810>; action = S; cost = 25
States on the path: [(10, 24), (10, 23), (10, 22), (10, 21), (9, 21), (8, 21), (8, 22), (8, 23), (8, 24), (8, 25), (8, 26), (8, 27), (8, 28), (8, 29), (8, 30), (8, 31), (8, 32), (7, 32), (6, 32), (5, 32), (4, 32), (3, 32), (3, 33), (3, 34), (2, 34)]
============================================================================
multi_goal_maze2.txt :
Goal State:  Node - pos = (8, 30); parent = <__main__.Node object at 0x000001E75C5CD630>; action = N; cost = 11
States on the path: [(8, 30), (8, 31), (8, 32), (7, 32), (6, 32), (5, 32), (4, 32), (3, 32), (3, 33), (3, 34), (2, 34)]
============================================================================
```

Running DFS on these new mases:

In [25]:
```
#RUNNING DFS ON THE MAZES
import pandas as pd
#use pandas to dispaly as table
column_display=['MazeID', 'Cost', 'Max Frontier Size', 'Visited Nodes Number', 'Max Tree depth']
data_out = []
df = pd.DataFrame(data_out, columns=column_display)
MazefileNames = ["multi_goal_maze0.txt", "multi_goal_maze1.txt", "multi_goal_maze2.txt"]
verbose = True

for i in MazefileNames:
    #parse maze:
    f = open(i, "r")
    maze_ID_txt = f.read()
    maze = mh.parse_maze(maze_ID_txt)
    #perform DFS and get stats:
    statrting_position = mh.find_pos(maze, what="S")
    root = Node(pos=statrting_position, parent=None, action=None, cost=0)  # make root node
    goalNode_curr = DFS_implement(maze, root)
    max_frontier_curr = frontier_size_max
    num_nodes_curr = number_of_nodes_expand
    tree_depth_max_curr = tree_depth_max
    #get final path
    List = []
    temp = goalNode_curr
    while temp != root:
        List.append(temp.pos)
        temp = temp.parent
    #ouptut solution data:
    fileName = i
    if verbose:
        print(fileName, ':')
        print('Solution Node: ', goalNode_curr)
        print('Squares on final path:', List)
        print('=====================================================')
    #append to the dataframe created
    data = [fileName, goalNode_curr.cost, max_frontier_curr, num_nodes_curr, tree_depth_max_curr]
    df.loc[len(df.index)] = data

DFS_multigoal = df
```

```
multi_goal_maze0.txt :
Solution Node:  Node - pos = (9, 20); parent = <__main__.Node object at 0x000001E75C5CE920>; action = E; cost = 80
Squares on final path: [(9, 20), (8, 20), (8, 21), (8, 22), (8, 23), (8, 24), (8, 25), (8, 26), (8, 27), (9, 27), (10, 27), (11, 27), (11, 28), (11, 29), (11, 30), (11, 31), (11, 32), (11, 33), (11, 34), (10, 34), (9, 34), (9, 33), (9, 32), (8, 32), (8, 31), (8, 30), (7, 30), (7, 31), (7, 32), (7, 33), (7, 34), (6, 34), (6, 33), (6, 32), (6, 31), (6, 30), (6, 29), (6, 28), (6, 27), (6, 26), (6, 25), (6, 24), (6, 23), (6, 22), (6, 21), (6, 20), (5, 20), (5, 19), (5, 18), (5, 17), (4, 17), (3, 17), (3, 16), (3, 15), (3, 14), (3, 13), (3, 12), (2, 12), (2, 13), (2, 14), (2, 15), (2, 16), (2, 17), (1, 17), (1, 18), (1, 19), (1, 20), (1, 21), (1, 22), (1, 23), (1, 24), (1, 25), (1, 26), (1, 27), (1, 28), (1, 29), (1, 30), (1, 31), (1, 32), (1, 33)]
=====================================================
multi_goal_maze1.txt :
Solution Node:  Node - pos = (16, 3); parent = <__main__.Node object at 0x000001E759FDC760>; action = N; cost = 92
Squares on final path: [(16, 3), (16, 4), (16, 5), (16, 6), (16, 7), (16, 8), (16, 9), (16, 10), (15, 10), (15, 11), (15, 12), (15, 13), (15, 14), (15, 15), (15, 16), (15, 17), (15, 18), (15, 19), (15, 20), (15, 21), (15, 22), (15, 23), (15, 24), (15, 25), (15, 26), (15, 27), (14, 27), (13, 27), (12, 27), (11, 27), (11, 28), (11, 29), (11, 30), (11, 31), (11, 32), (11, 33), (11, 34), (10, 34), (9, 34), (9, 33), (9, 32), (8, 32), (8, 31), (8, 30), (8, 29), (8, 28), (8, 27), (8, 26), (8, 25), (8, 24), (8, 23), (8, 22), (8, 21), (9, 21), (9, 20), (9, 19), (9, 18), (9, 17), (8, 17), (7, 17), (7, 16), (7, 15), (7, 14), (6, 14), (6, 13), (5, 13), (5, 12), (4, 12), (3, 12), (2, 12), (2, 13), (2, 14), (2, 15), (2, 16), (2, 17), (1, 17), (1, 18), (1, 19), (1, 20), (1, 21), (1, 22), (1, 23), (1, 24), (1, 25), (1, 26), (1, 27), (1, 28), (1, 29), (1, 30), (1, 31), (1, 32), (1, 33)]
=====================================================
multi_goal_maze2.txt :
Solution Node:  Node - pos = (3, 15); parent = <__main__.Node object at 0x000001E75C5CCBE0>; action = S; cost = 27
Squares on final path: [(3, 15), (3, 14), (3, 13), (3, 12), (2, 12), (2, 13), (2, 14), (2, 15), (2, 16), (2, 17), (1, 17), (1, 18), (1, 19), (1, 20), (1, 21), (1, 22), (1, 23), (1, 24), (1, 25), (1, 26), (1, 27), (1, 28), (1, 29), (1, 30), (1, 31), (1, 32), (1, 33)]
=====================================================
```

IDS Algorithm:

In [26]:
```
import random

frontier_size_max = 0
tree_depth_max = 0
number_of_nodes_expand = 0
visited = []

class Node:
    def __init__(self, pos, parent, action, cost):
        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 has None)
        self.cost = cost         # for uniform cost this is the depth. It is also g(n) for A* search

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

def ChildInfo(NodeInfo, MazeInfo, frontier):
    childrenList = []
    PosE = NodeInfo.pos[0] + 1, NodeInfo.pos[1]
    PosW = NodeInfo.pos[0] - 1, NodeInfo.pos[1]
    PosN = NodeInfo.pos[0], NodeInfo.pos[1] - 1
    PosS = NodeInfo.pos[0], NodeInfo.pos[1] + 1
    #look at all potential spaces
    if mh.look(MazeInfo, PosE) != 'X' and not PosE in frontier:
        newChild = Node(pos = PosE, parent = NodeInfo, action = "E", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosW) != 'X' and not PosW in frontier:
        newChild = Node(pos = PosW, parent = NodeInfo, action = "W", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosN) != 'X' and not PosN in frontier:
        newChild = Node(pos = PosN, parent = NodeInfo, action = "N", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    if mh.look(MazeInfo, PosS) != 'X' and not PosS in frontier:
        newChild = Node(pos = PosS, parent = NodeInfo, action = "S", cost = NodeInfo.cost + 1)
        childrenList.append(newChild)
    return childrenList


#IDS initialize
def IDS_Implement(maze, root):
    maxDepth = 1
    for maxDepth in range(0,1000):
        #print(maxDepth)
        currDepth = 0
        #variable intilization:
        global frontier_size_max
        global tree_depth_max
        global number_of_nodes_expand
        frontier_size_max = 0
        tree_depth_max = 0
        number_of_nodes_expand = 0
        stack = []
        global visited
        visited = []
        stack.append(root) #stack intilization
        currNode = root
        while ( len(stack) > 0 ): #while frontier/stack ISNAN empty
            prevNode = currNode
            currNode = stack.pop()

            number_of_nodes_expand = number_of_nodes_expand + 1
            if currNode.cost < prevNode.cost: #RELEASE PREV NODES
                prevNode = None
            visited.append(currNode.pos)
            if currNode.cost > tree_depth_max:
                tree_depth_max = currNode.cost
            if len(stack) > frontier_size_max:
                frontier_size_max = len(stack)
            if mh.look(maze, currNode.pos) == 'G': #GOAL STATE

                return currNode
            # potential children
            if tree_depth_max >= maxDepth:
                break
            childrenList = ChildInfo(currNode, maze, visited)
            random.shuffle(childrenList)
            for i in childrenList:
                if not (i in stack):
                    stack.append(i)
                    currDepth += 1
    print('failed')
    return root
```

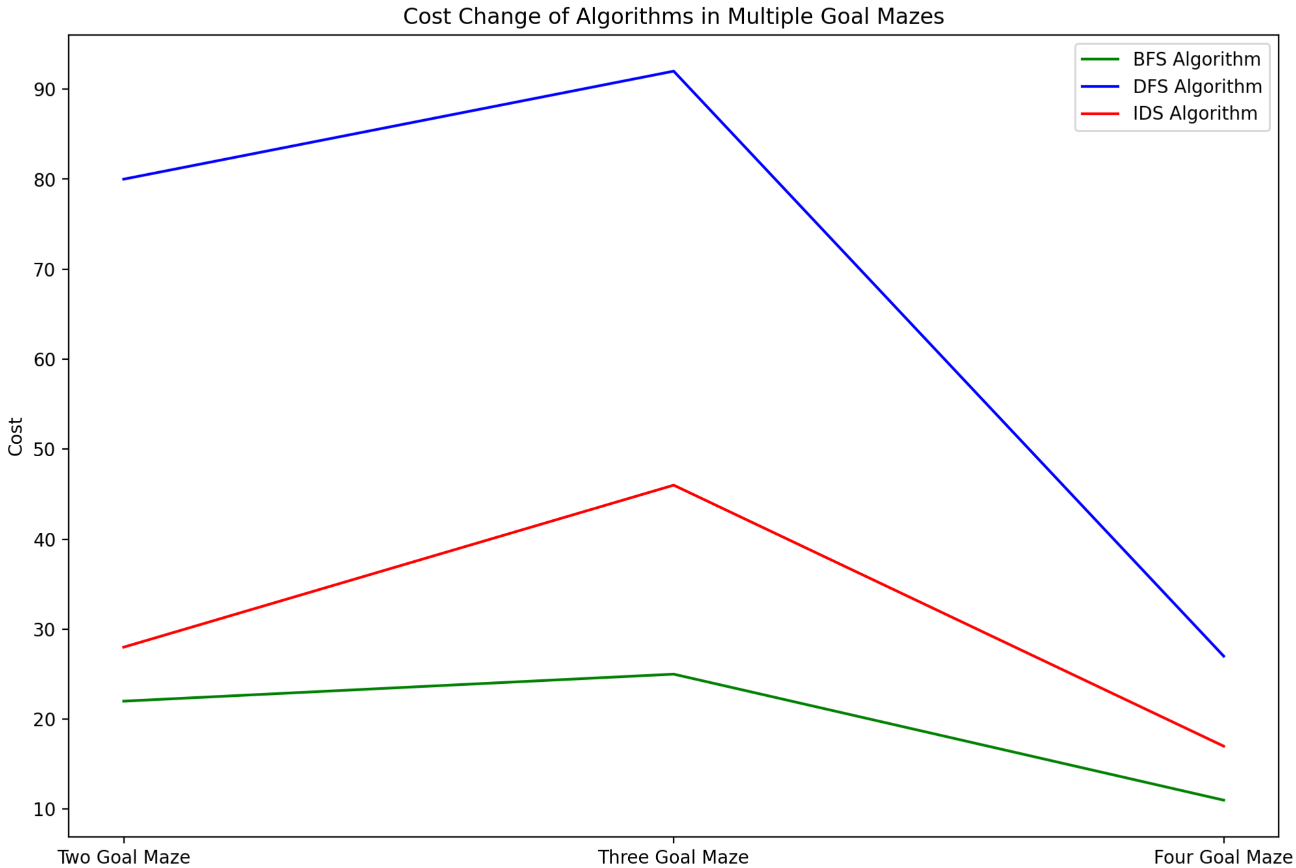
In [27]:
```
#RUNNING IDS ON  NEW MAZES
import pandas as pd
column_display=['MazeID', 'Cost', 'Max Frontier Size', 'Visited Nodes Number', 'Max Tree depth']
data_out = []
```

```
df = pd.DataFrame(data_out, columns=column_display)
MazefileNames = ["multi_goal_maze0.txt", "multi_goal_maze1.txt", "multi_goal_maze2.txt"]
verbose = True

for i in MazefileNames:
    f = open(i, "r")
    maze_ID_txt = f.read()
    maze = mh.parse_maze(maze_ID_txt)
    # DFS
    statrting_position = mh.find_pos(maze, what="S")
    root = Node(pos=statrting_position, parent=None, action=None, cost=0)  # make the root
    goalNode_curr = IDS_Implement(maze, root)
    max_frontier_curr = frontier_size_max
    num_nodes_curr = number_of_nodes_expand
    tree_depth_max_curr = tree_depth_max
    #get final path
    List = []
    temp = goalNode_curr
    while temp != root:
        List.append(temp.pos)
        temp = temp.parent
    fileName = i
    if verbose:
        print(fileName, ':')
        print('Solution Node: ', goalNode_curr)
        print('Squares on final path:', List)
        print('All squares visited marked as visited in the visited List. Not printed due to size')
        print('------------------------------------------------------------------------')
    data = [fileName, goalNode_curr.cost, max_frontier_curr, num_nodes_curr, tree_depth_max_curr]
    df.loc[len(df.index)] = data

IDS_multigoal = df
```

```
multi_goal_maze0.txt :
Solution Node:  Node - pos = (9, 20); parent = <__main__.Node object at 0x000001E75A78DC60>; action = N; cost = 28
Squares on final path: [(9, 20), (9, 21), (9, 22), (8, 22), (8, 23), (8, 24), (8, 25), (7, 25), (6, 25), (6, 26), (6, 27), (6, 28), (6, 29), (5, 29), (5, 3
0), (6, 30), (6, 31), (6, 32), (5, 32), (4, 32), (4, 31), (3, 31), (3, 30), (2, 30), (2, 31), (1, 31), (1, 32), (1, 33)]
All squares visited marked as visited in the visited List. Not printed due to size
------------------------------------------------------------------------
multi_goal_maze1.txt :
Solution Node:  Node - pos = (6, 7); parent = <__main__.Node object at 0x000001E759FDE7D0>; action = N; cost = 46
Squares on final path: [(6, 7), (6, 8), (6, 9), (5, 9), (5, 10), (6, 10), (7, 10), (7, 11), (7, 12), (6, 12), (5, 12), (4, 12), (3, 12), (3, 13), (2, 13),
(2, 14), (2, 15), (2, 16), (2, 17), (1, 17), (1, 18), (2, 18), (2, 19), (2, 20), (1, 20), (1, 21), (1, 22), (1, 23), (1, 24), (1, 25), (1, 26), (1, 27), (1,
28), (1, 29), (1, 30), (2, 30), (2, 31), (3, 31), (4, 31), (4, 32), (3, 32), (3, 33), (3, 34), (2, 34), (2, 33), (1, 33)]
All squares visited marked as visited in the visited List. Not printed due to size
------------------------------------------------------------------------
multi_goal_maze2.txt :
Solution Node:  Node - pos = (8, 30); parent = <__main__.Node object at 0x000001E75A78E4D0>; action = N; cost = 17
Squares on final path: [(8, 30), (8, 31), (7, 31), (7, 32), (7, 33), (7, 34), (6, 34), (6, 33), (6, 32), (6, 31), (5, 31), (5, 32), (4, 32), (3, 32), (3, 3
3), (3, 34), (2, 34)]
All squares visited marked as visited in the visited List. Not printed due to size
------------------------------------------------------------------------
```

In [28]:

```
import matplotlib.pyplot as plt
from matplotlib.pyplot import figure
MazefileNames = ["Two Goal Maze", "Three Goal Maze", "Four Goal Maze"]

fig = plt.figure(figsize=(12, 8))
plt.plot(MazefileNames, BFS_multigoal["Cost"].to_numpy(),'g', label='BFS Algorithm')
plt.plot(MazefileNames, DFS_multigoal["Cost"].to_numpy(),'b', label='DFS Algorithm')
plt.plot(MazefileNames, IDS_multigoal["Cost"].to_numpy(),'r', label='IDS Algorithm')
plt.legend()
plt.title('Cost Change of Algorithms in Multiple Goal Mazes')
plt.ylabel('Cost')
plt.show()
```



*DFS works better with the addition of goal states.

*BFS is still optimal and DFS is more likely to return an solution quicker. BFS still has large complexity and DFS in not optimal.

*IDS combines some benefits of BFS and DFS. It returns a close result as BFS but with a less memory need.

*The space complexity of IDS is linear since each tree is released after the maximum depth is reached. IDS is a good solution with in terms of complexity and the cost.

In [29]:
```
DFS_multigoal
```

Out[29]:

	MazeID	Cost	Max Frontier Size	Visited Nodes Number	Max Tree depth
0	multi_goal_maze0.txt	80	49	91	82
1	multi_goal_maze1.txt	92	52	186	92
2	multi_goal_maze2.txt	27	15	32	27

In [30]:
```
IDS_multigoal
```

Out[30]:

	MazeID	Cost	Max Frontier Size	Visited Nodes Number	Max Tree depth
0	multi_goal_maze0.txt	28	29	39	28
1	multi_goal_maze1.txt	46	32	62	46
2	multi_goal_maze2.txt	17	17	18	17

In [31]:
```
BFS_multigoal
```

Out[31]:

	MazeID	Cost	Max Frontier Size	Visited Nodes Number	Max Tree depth
0	multi_goal_maze0.txt	22	10	118	21
1	multi_goal_maze1.txt	25	9	138	24
2	multi_goal_maze2.txt	11	7	37	10

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 Cost, how can we make sure that BFS and iterative deepening search is optimal? Change the code to do so.

In [32]:
```
# Your code/answer goes here
```

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

In [33]:
```
# 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 [34]:
```
# Your code/answer goes here
```