Maze_assignment

March 16, 2021

1 Search: Solving a Maze Using a Goal-based Agent

1.1 Instructions

Total Points: Undegraduates 10, graduate students 11

Complete this notebook and submit it. The notebook needs to be a complete project report with

- your implementation,
- documentation including a short discussion of how your implementation works and your design choices, and
- experimental results (e.g., tables and charts with simulation results) with a short discussion of what they mean.

Use the provided notebook cells and insert additional code and markdown cells as needed.

1.2 Introduction

The agent has a map of the maze it is in (i.e., the environment is 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 can just follow the path and does not need percepts. 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 and the goal is to traverse all nodes. We are interested in dynamically creating a search tree with the aim of finding a good/the best path to the goal state. Follow the pseudo code presented in the text book 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.

Here is the small example maze:

[1]:
```
# import os
# script_dir = os.getcwd()
# script_dir
```

[2]:
```
f = open("small_maze.txt", "r")
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 search may not form proper trees unless cycles are prevented. You need to deal with cycle detection in your code.

1.3 Parsing and pretty printing the maze

The maze can also be displayed in color using code in the file `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. I represent a position as a 2-tuple of the form `(row, col)`.

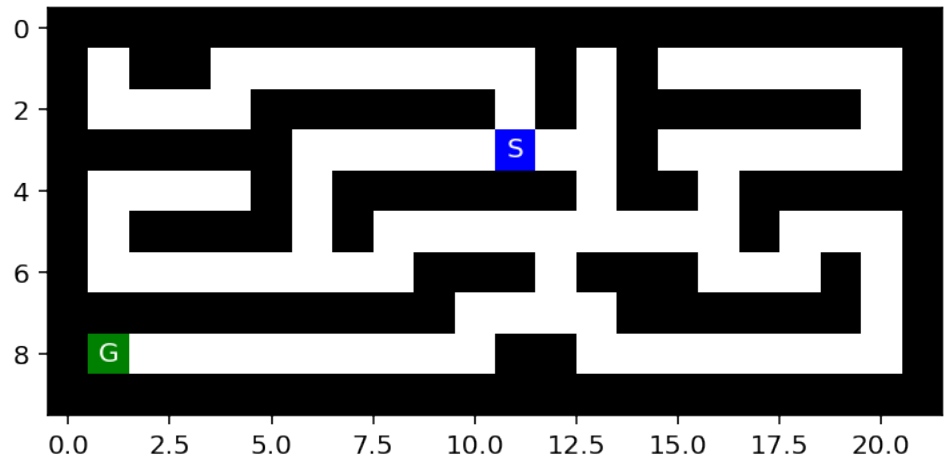
[3]:
```
%run maze_helper.py
import random

maze = parse_maze(maze_str)

print(type(maze))
print(maze)

# look at two positions in the maze
print("Position(0,0):", maze[0, 0])

# there is also a helper function called `look(maze, pos)`
print("Position(8,1):", look(maze, (8, 1)))
```

```
<class 'numpy.ndarray'>
[['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X'
  'X' 'X' 'X' 'X']
 ['X' ' ' 'X' 'X' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' 'X' ' ' 'X' ' ' ' ' ' '
  ' ' ' ' ' ' 'X']
 ['X' ' ' ' ' ' ' ' ' 'X' 'X' 'X' 'X' 'X' 'X' ' ' 'X' ' ' 'X' 'X' 'X' 'X'
  'X' 'X' ' ' 'X']
 ['X' 'X' 'X' 'X' 'X' 'X' ' ' ' ' ' ' ' ' ' ' ' ' 'S' ' ' ' ' 'X' ' ' ' ' ' '
  ' ' ' ' ' ' 'X']
 ['X' ' ' ' ' ' ' ' ' 'X' ' ' 'X' 'X' 'X' 'X' 'X' 'X' ' ' 'X' 'X' ' ' 'X'
```

```
 'X' 'X' 'X' 'X']
['X' ' ' 'X' 'X' 'X' 'X' ' ' 'X' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' 'X'
 ' ' ' ' ' ' 'X']
['X' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' 'X' 'X' 'X' ' ' 'X' 'X' 'X' ' ' ' '
 ' ' 'X' ' ' 'X']
['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' ' ' ' ' ' ' ' ' ' ' 'X' 'X' 'X' 'X'
 'X' 'X' ' ' 'X']
['X' 'G' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' ' 'X' 'X' ' ' ' ' ' ' ' ' '
 ' ' ' ' ' ' 'X']
['X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X' 'X'
 'X' 'X' 'X' 'X']]
Position(0,0): X
Position(8,1): G
```

Visualize the maze

[4]:
```
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
# use higher resolution images in notebook

show_maze(maze)
```



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

[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)
```

1.4 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.

1.5 I have modified this class by adding some functions to meet the needs when working on the search algorithms.

[6]:
```
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 __eq__(self,other):# Here I redefined the object comparison method in␣
↪order to verify the node if visited
        if self.pos == other.pos: return True
        else : return False

    def __bool__(self): # define the bool attribute of the object as True
        return True

    def is_circle(self): # to verify if the node is in a circle
        node = self
        path = [node]

        while not node.parent is None:
            node = node.parent
            if node in path:
                return True
            path.append(node)

        return False

    def expand(self,maze,actions):# expand nodes
        width,height = maze.shape
        nodes_list = []
        for key,value in actions.items():
```

```
            curPos = (self.pos[0]+value[0],self.pos[1]+value[1])
            if curPos[0] < width and curPos[1] < height and maze[curPos]!="X":
                nodes_list.
↪append(Node(pos=curPos,parent=self,action=key,cost=self.cost+1))
        return nodes_list

    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.

Tree and memory management example: Create the root node and append the node for going east with a cost of 1.

[7]:
```
import sys

# print("Create root node")
# root = Node(pos = (0,0), parent = None, action = None, cost = 0)
# print("root:", root)
# print("root (reference):", repr(root))

# print()
# print("Go east with cost 1 (from the parent root)")
# node2 = Node(pos = (0,1), parent = root, action = "E", cost = 1)
# print("node2:", node2)
# print("parent of node2: ", node2.parent)
# print("parent of node2 (reference): ", repr(node2.parent))
# # Note: -1 is used because passing root to getrefcount adds one reference
# print("Root is now referenced twice. Reference count for garbage collection␣
↪(root node):", sys.getrefcount(root)-1)

# print()
# print("Note that the root node is safe from garbage collection as long as we␣
↪have also a reference in node2")
# root = None
# print(root)
# print("parent of node2 (reference to root node): ", repr(node2.parent))
```

```
# print("Reference count for garbage collection (root node):", sys.
 ↪getrefcount(node2.parent)-1)

# print()
# print("Path from root to node2")
# path = node2.get_path_from_root()
# print("References:", path)
# print("Positions:", [n.pos for n in path])
# print("Actions:", [n.action for n in path])
# print("Cost:", [n.cost for n in path])

# print()
# node3 = Node(pos=(0,2), parent=node2,action="E",cost=1)
# path = node3.get_path_from_root()
# print("References:", path)
# print("Positions:", [n.pos for n in path])
# print("Actions:", [n.action for n in path])
# print("Cost:", [n.cost for n in path])


# print()
# print("Once we delete the reference to node2, the reference count for all␣
 ↪nodes goes to zero and the whole tree is exposed to garbage collection.")
# node2 = None
```

2 Goal

Implement the following search algorithms for solving different mazes:

- Breadth-first search (BFS)
- Depth-first search (DFS)
- Greedy best-first search (GBFS)
- A* search

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. For each problem instance and each search algorithm, report the following in a table:

- The solution and its path cost
- Number of nodes expanded
- Maximum tree depth
- Maximum size of the frontier

Display each solution by marking every maze square (or state) visited and the squares on the final path.

2.1 Task 1: Defining the search Problem [1 point]

Define the components of the search problem:

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

3 Your answer goes here

- Initial state: S point in the maze
- Action : four directions such as east, west, north and south
- Transition model : set of actions from location S to location G
- Goal state : reaching the location of G point in the maze
- Path cost: the sum of step costs from S to G

3.1 This function is used for reading a maze and converting to a numpy array

[8]:
```
def read_maze_by_name(mazeName):
    f = open(mazeName+".txt", "r")
    return parse_maze(f.read())
```

3.2 This function is used for seeking multiple goals

I design these functions since I plan to implement the multiple goals of BFS, DFS and IDS at first

[9]:
```
def find_multiple_goal(maze):
    tmp = np.where(maze == 'G')
    result = []
    if len(tmp)>0:
        for i in range(len(tmp[0])):
            result.append((tmp[0][i],tmp[1][i]))
    return result
```

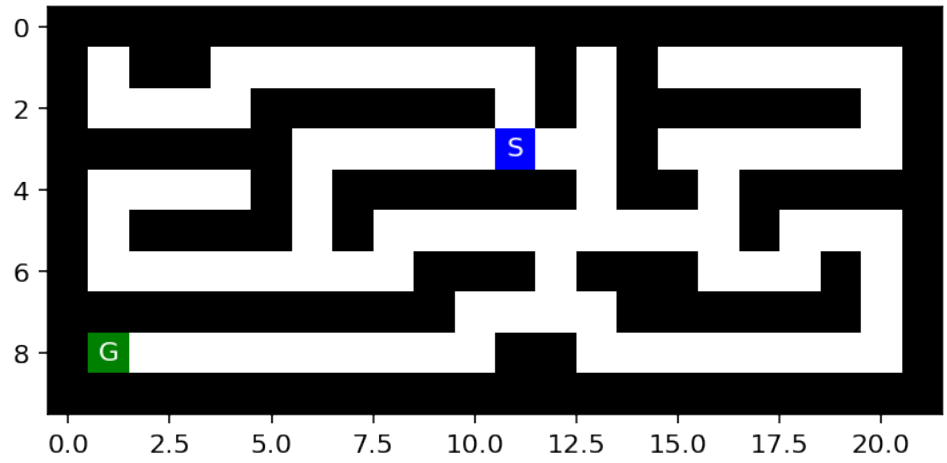
3.3 This function is used for displaying the multiple goals in the maze

[10]:
```
# it could show multiple goal states
def show_maze_v2(maze, fontsize = 10):
    cmap = colors.ListedColormap(['white', 'black', 'blue', 'green', 'red',␣
 ↪'gray', 'orange'])

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

    goals = find_multiple_goal(maze)
    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
    # Converts all string values to integers
    maze = maze.astype(np.int)

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

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

    for goal in goals:
        plt.text(goal[1], goal[0], "G", fontsize = fontsize, color = "white",
                horizontalalignment = 'center',
                verticalalignment = 'center')

    plt.show()
```

[11]:
```
# maze[(3,12)] = 'G'
# find_pos(maze,'S')

a = find_multiple_goal(maze)
print(a)

show_maze_v2(maze)
```

```
[(8, 1)]
```



3.4 This function is used for debugging and displaying the result as talked in the live session.

I find it very useful so that I can watch each step whether it meets the expectation

[12]:
```
# function for debugging
# it can display G,S,expanded node,path,wall,frontiers
# it's modified based on the function show_maze(which comes from the␣
 ↪maze_helper.py)
def show_path(maze, result, fontsize=10,vis=True):
    cmap = colors.ListedColormap(['white', 'black', 'blue', 'green', 'red',␣
 ↪'gray', 'orange'])

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

    goals = find_multiple_goal(maze)
    start = find_pos(maze, 'S')

    if len(result)>1:
        goal_nodes = result[0]#goal node list
        reached_nodes = result[1]#every node reached yet

        for goal_node in goal_nodes:
            for node in goal_node.get_path_from_root():
                if maze[node.pos] == ' ':
                    maze[node.pos] = 'P'

        for reached_node in reached_nodes:
            if maze[reached_node.pos] == ' ':
                maze[reached_node.pos] = '.'

        if len(result)>2:
            frontiers = result[2]
            for frontier in frontiers:
                if maze[frontier.pos] != 'S':
                    maze[frontier.pos] = 'F'

        if len(result)>3:
            print("Path cost:",result[3][0])
            print("Node expanded:",result[3][1])
            print("Max tree depth:",result[3][2])
            print("Max tree size:",result[3][3])
            print("Max frontier size:",result[3][4])
```

```
    # 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
    # Converts all string values to integers
    maze = maze.astype(np.int)

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

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

        for goal in goals:
            plt.text(goal[1], goal[0], "G", fontsize = fontsize, color =␣
↪"white",
                    horizontalalignment = 'center',
                    verticalalignment = 'center')

        plt.show()
```

3.5 This function is used for verifying if the node is one of the goal state since there are multiple goals.

[13]:
```
# to verify if it reaches the goal state
def is_goal(frontier,goal_states):
    if frontier.pos in goal_states: return True
    else: return False
```

[14]:
```
# it's used in depth-first search without reached data structure only, in order␣
↪to get back tracking nodes
def get_back_tracking_nodes(start_node,targetParent):
    list = []
    node = start_node
    while not node.parent is targetParent:
        list.append(node)
        node = node.parent
    return list
```

3.6 Task 2: Breadth-first and Depth-first [4 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 during search in the tree, and the `reached` and `frontier` data structures. * DSF can be implemented using the BFS tree search algorithm and 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, you need to implement DFS in a different way without a `reached` data structure and by releasing nodes that are not needed anymore. * If DFS does not use a `reached` data structure, then its cycle checking abilities are limited. You will see in your experiments that open spaces are a problem.

3.7 This function is an implementation of Breadth-first search

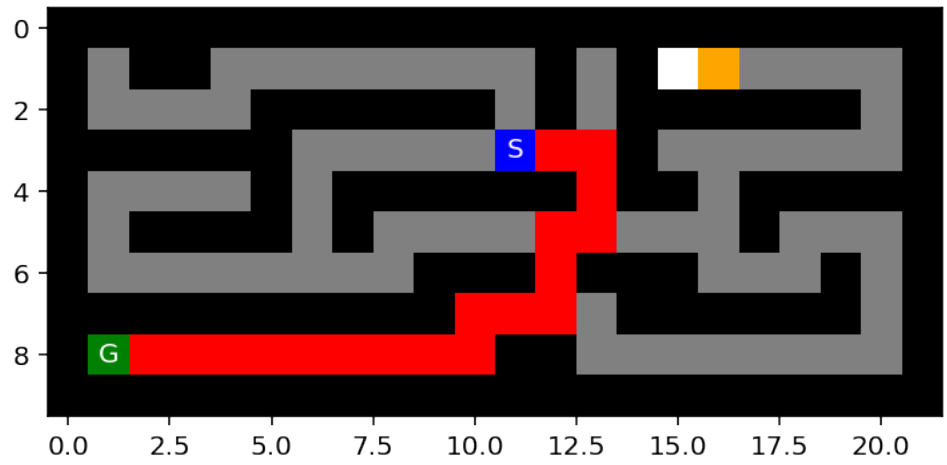
It's both complete and optimal.

[15]:
```
# Breadth-first search supporting multiple goals
# maze: the maze
# limit_steps: stop the loop after limite_steps times
# debug: if it's True, it prints every step of walking through the maze
def BFS(maze,limit_steps = 1000,debug=False,actions = {"N":(-1,0), "S":(1,0),␣
 ↪"W":(0,-1), "E":(0,1)},vis=False):
    frontiers = []
    reached = []
    initial_state = find_pos(maze, what = "S")
    goal_states = find_multiple_goal(maze)
    goal_nodes = []
    # measure variables
    measure_path_cost = 0
    measure_node_expanded = 0
    measure_max_tree_depth = 0
    measure_max_tree_size = 0
    measure_max_frontier_size = 0

    if actions == "R":
        actions = {"N":(-1,0), "S":(1,0), "W":(0,-1), "E":(0,1)}
        l = list(actions.items())
        random.shuffle(l)
        actions = dict(l)

    root = Node(pos=initial_state,parent=None,action=None,cost=0)
    reached.append(root)
    frontiers.append(root)
    measure_node_expanded += 1
```

```
    while frontiers :
        if debug: show_path(maze = maze,␣
↪result=(goal_nodes,reached,frontiers),vis=vis)

        #pop a frontier
        frontier = frontiers.pop(0)

        if len(frontiers)+1 > measure_max_frontier_size:␣
↪measure_max_frontier_size = len(frontiers)+1 # max frontier size
        if len(reached) > measure_max_tree_size: measure_max_tree_size =␣
↪len(reached)#max tree size
        measure_node_expanded += 1 # number of nodes expanded

        if frontier.cost > measure_max_tree_depth : measure_max_tree_depth =␣
↪frontier.cost

        if is_goal(frontier,goal_states) and (frontier not in goal_nodes):
            goal_nodes.append(frontier)
            if len(goal_nodes) == len(goal_states):
                measure_path_cost = (sum(x.cost for x in goal_nodes))# path cost
                return␣
↪goal_nodes,reached,frontiers,[measure_path_cost,measure_node_expanded,measure_max_tree_depth
        else :
            for subNode in frontier.expand(maze=maze,actions=actions):
                if (subNode not in reached) and (subNode not in frontiers):
                    reached.append(subNode)
                    frontiers.append(subNode)
        limit_steps -= 1
        if limit_steps<=0: break
    #if there is not a solution, it returns empty of goal_nodes
    return␣
↪([],reached,[],[measure_path_cost,measure_node_expanded,measure_max_tree_depth,measure_max_
```

[16]:
```
%time result = BFS(maze = maze,actions='R', debug=False)
show_path(maze,result)
```

```
CPU times: user 5.59 ms, sys: 307 µs, total: 5.89 ms
Wall time: 5.62 ms
Path cost: 19
Node expanded: 93
Max tree depth: 19
Max tree size: 93
Max frontier size: 8
```



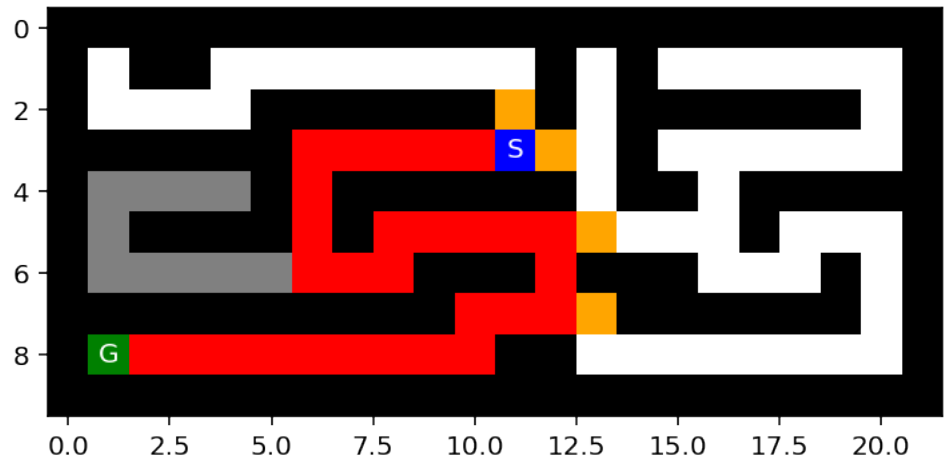
3.8 This function is an implementation for depth-first search with reached data structure

[17]:
```
# Depth-first search with reached data structure supporting multiple goals
# maze: the maze
# limit_steps: stop the loop after limite_steps times
# debug: if it's True, it prints every step of walking through the maze
def DFS_with_reached(maze,limit_steps = 1000,debug=False,actions = {"N":(-1,0),␣
↪"S":(1,0), "W":(0,-1), "E":(0,1)},vis=False):
    frontiers = []
    reached = []
    initial_state = find_pos(maze, what = "S")
    goal_states = find_multiple_goal(maze)
    goal_nodes = []
    # measure variables
    measure_path_cost = 0
    measure_node_expanded = 0
    measure_max_tree_depth = 0
    measure_max_tree_size = 0
    measure_max_frontier_size = 0

    if actions == "R":
        actions = {"N":(-1,0), "S":(1,0), "W":(0,-1), "E":(0,1)}
        l = list(actions.items())
        random.shuffle(l)
        actions = dict(l)

    width,height = maze.shape#width and height of maze
    root = Node(pos=initial_state,parent=None,action=None,cost=0)
    reached.append(root)
```

```
    frontiers.append(root)
    measure_node_expanded += 1

    while frontiers :
        if debug: show_path(maze = maze,␣
↪result=(goal_nodes,reached,frontiers),vis=vis)

        #pop a frontier
        frontier = frontiers.pop()

        if len(frontiers)+1 > measure_max_frontier_size:␣
↪measure_max_frontier_size = len(frontiers)+1 # max frontier size
        if len(reached) > measure_max_tree_size: measure_max_tree_size =␣
↪len(reached) # max tree size
        measure_node_expanded += 1
        if frontier.cost > measure_max_tree_depth: measure_max_tree_depth =␣
↪frontier.cost# max tree depth

        if is_goal(frontier,goal_states) and (frontier not in goal_nodes):
            goal_nodes.append(frontier)
            if len(goal_nodes) == len(goal_states):
                measure_path_cost = (sum(x.cost for x in goal_nodes))# path cost
                return␣
↪goal_nodes,reached,frontiers,[measure_path_cost,measure_node_expanded,measure_max_tree_depth
        else :
            for subNode in frontier.expand(maze=maze,actions=actions):
                if (subNode not in reached) and (subNode not in frontiers):
                    reached.append(subNode)
                    frontiers.append(subNode)
        limit_steps -= 1
        if limit_steps<=0: break
    #if there is not a solution, it returns empty of goal_nodes
    return␣
↪([],reached,frontiers,[measure_path_cost,measure_node_expanded,measure_max_tree_depth,measu
```

[18]:
```
%time result = DFS_with_reached(maze = maze,actions='R', debug=False)
show_path(maze,result)
```

```
CPU times: user 1.04 ms, sys: 6 µs, total: 1.05 ms
Wall time: 1.05 ms
Path cost: 29
Node expanded: 41
Max tree depth: 29
Max tree size: 44
Max frontier size: 5
```



3.9 This function is an implementation of depth-first search without reached data structure

The reached data structure I used in this function is only for debugging.

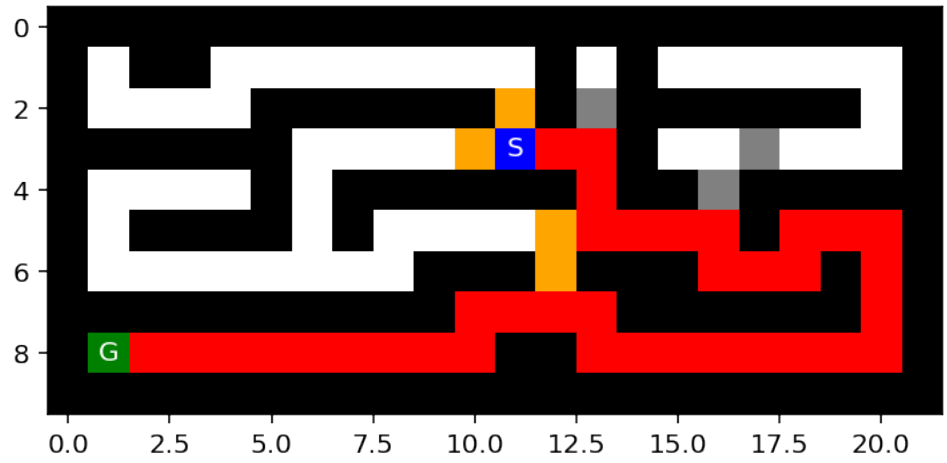
[19]:
```
# Depth-first search without reached data structure supporting multiple goals
# maze: the maze
# limit_steps: stop the loop after limite_steps times
# debug: if it's True, it prints every step of walking through the maze
# actions: the order of generating sub node from frontier
def DFS(maze,limit_steps = 1000,debug=False,actions = {"N":(-1,0), "S":(1,0),␣
 ↪"W":(0,-1), "E":(0,1)},depth_limit=300,vis=False):
    frontiers = []
    reached = []# used for calculating expanded nodes
    initial_state = find_pos(maze, what = "S")
    goal_states = find_multiple_goal(maze)
    goal_nodes = []
    # measure variables
    measure_path_cost = 0
    measure_node_expanded = 0
    measure_max_tree_depth = 0
    measure_max_tree_size = 0
    measure_max_frontier_size = 0

    if actions == "R":
        actions = {"N":(-1,0), "S":(1,0), "W":(0,-1), "E":(0,1)}
        l = list(actions.items())
        random.shuffle(l)
        actions = dict(l)
    # print(actions)
```

```
    width,height = maze.shape#width and height of maze
    root = Node(pos=initial_state,parent=None,action=None,cost=0)
    reached.append(root)
    frontiers.append(root)
    measure_node_expanded += 1

    while frontiers :
        if debug: show_path(maze = maze,␣
↪result=(goal_nodes,reached,frontiers),vis=vis)

        # if get_tree_size(frontiers) > measure_max_tree_size:␣
↪measure_max_tree_size = get_tree_size(frontiers)#max tree size

        frontier = frontiers.pop()

        if debug: print("each pop frontier",frontier.pos)

        if len(frontiers)+1 > measure_max_frontier_size:␣
↪measure_max_frontier_size = len(frontiers)+1 # max frontier size
        measure_node_expanded += 1

        if frontier.cost > measure_max_tree_depth : measure_max_tree_depth =␣
↪frontier.cost
        if len(reached) > measure_max_tree_size: measure_max_tree_size =␣
↪len(reached)# max tree size

        if is_goal(frontier,goal_states) and (frontier not in goal_nodes):
            goal_nodes.append(frontier)
            if len(goal_nodes) == len(goal_states):
                measure_path_cost = (sum(x.cost for x in goal_nodes))# path cost
                return␣
↪goal_nodes,reached,frontiers,[measure_path_cost,measure_node_expanded,measure_max_tree_depth
        else :
            back_tracking = True
            for subNode in frontier.expand(maze=maze,actions=actions):
                if subNode.is_circle() or (subNode in frontiers):
                    subNode = None
                    continue
                else :
                    back_tracking = False
                    frontiers.append(subNode)
                #this code is only used for calculating the expanded nodes
                if subNode not in reached: reached.append(subNode)

            if back_tracking and len(frontiers)>1 :
```

```
                for nodes_to_removed in␣
↪get_back_tracking_nodes(frontier,frontiers[-1].parent):
                    if nodes_to_removed in reached:
                        reached.remove(nodes_to_removed)
                    nodes_to_removed = None


        limit_steps -= 1
        if limit_steps<=0: break
    #if there is not a solution, it returns empty

    return␣
↪([],reached,frontiers,[measure_path_cost,measure_node_expanded,measure_max_tree_depth,measu
```

[20]:
```
%time result = DFS(maze = maze,actions='R', debug=False,depth_limit=50)
show_path(maze,result)
```

```
CPU times: user 3.9 ms, sys: 8 µs, total: 3.91 ms
Wall time: 3.91 ms
Path cost: 37
Node expanded: 55
Max tree depth: 37
Max tree size: 45
Max frontier size: 6
```

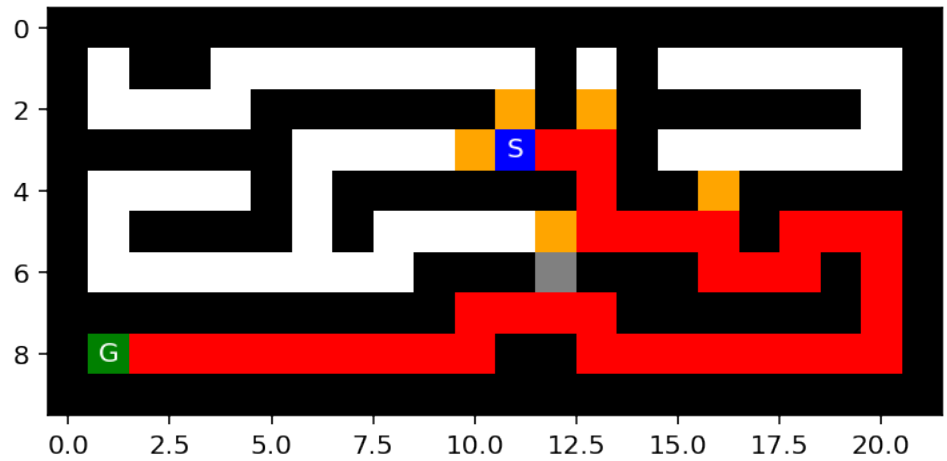


3.10 This function is an implementation for iterative deepening search

[21]:
```
# Iterative deepening search
# maze: the maze
# limit_steps: stop the loop after limite_steps times
# debug: if it's True, it prints every step of walking through the maze
# actions: the order of generating sub node from frontier
# depth_limit: the maximum depth the tree can reach. If it's set as 0, then␣
 ↪it'll gradually increase
def IDS(maze,limit_steps = 1000,debug=False,actions = {"N":(-1,0), "S":(1,0),␣
 ↪"W":(0,-1), "E":(0,1)},depth_limit=0,max_depth_limit=500):
    if depth_limit > 0:
        return␣
 ↪DFS(maze=maze,limit_steps=limit_steps,debug=debug,actions=actions,depth_limit=depth_limit)
    else:
        while depth_limit<max_depth_limit:
            depth_limit +=1
            result =␣
 ↪DFS(maze=maze,limit_steps=limit_steps,debug=debug,actions=actions,depth_limit=depth_limit)
            if len(result[0]) > 0:
                return result
```

[22]:
```
%time result = IDS(maze,depth_limit=0,actions='R')
show_path(maze,result)
```

```
CPU times: user 3.5 ms, sys: 8 µs, total: 3.5 ms
Wall time: 3.51 ms
Path cost: 37
Node expanded: 40
Max tree depth: 37
Max tree size: 44
Max frontier size: 7
```



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

Based on reached data structure, BFS can stop expanding the nodes visited yet so that it can avoid the loops.

If it's allowed to use reached data structure, DFS can also avoid the cycles, but it would lose the advantage of memory usage. Hence, in order to avoid the cycles, I need to verify if there are the same nodes among the chain of back tracking to the node's parents. However, this method is not perfect since it still falls into the cycle in some circumstances.

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

The implementation of BFS is both complete and optimal while the DFS is not. Because BFS will find the goal after eventually expanding all the nodes, it's complete. It's optimal since in this case, every step is the same cost.

In some extremely circumstances, the DFS would get stuck by adding the same nodes into frontier, because it pops the node from the frontier and after tens of iterations the same node is added into the frontier again.

b is maximum branching factor, m is max depth of the tree, d is depth of the optimal solution

time complexity of BFS: O(b*d),

space complexity of BFS: O(b^d)

time complexity of DFS: it's difficult to calculate since I do the back tracking removing in the loop of expanding nodes. If it's the with reached data structure one, it would be O(b^m)

space conplexity of DFS: O(b*m)

3.11 Task 3: Implement greedy best-first search and A* search [4 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).

[23]:
```
def euclidean(pos1,pos2):
    return np.sqrt(np.sum(np.square(np.subtract(pos1,pos2))))

def manhattan(pos1,pos2):
    return np.sum(np.abs(np.subtract(pos1,pos2)))

# heuristic function
# when the curCost doesn't set, it only estimates the distance from curPos to␣
 ↪targetPos
# when the curCost sets, it estimates the sum of distances between curPos to␣
 ↪targetPos plus current path cost
def cost_evaluation(curPos,targetPos,curCost=None,function=manhattan):
```

```
    if curCost == None:
        return function(curPos,targetPos)
    else:
        return function(curPos,targetPos)+curCost


print(manhattan((2,2),(3,1)))
print(euclidean((2,2),(3,1)))
print(cost_evaluation(curPos=(2,2),targetPos=(3,1)))
```

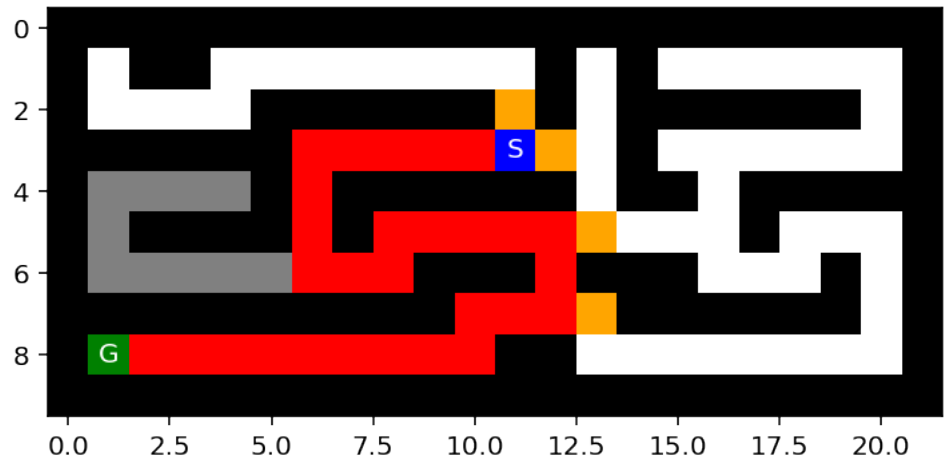
```
2
1.4142135623730951
2
```

[24]:
```
# Greedy best-first search
# maze: the maze
# limit_steps: stop the loop after limite_steps times
# debug: if it's True, it prints every step of walking through the maze
# actions: the order of generating sub node from frontier
def GBS(maze,limit_steps = 1000,debug=False,actions = {"N":(-1,0), "S":(1,0),␣
 ↪"W":(0,-1), "E":(0,1)}):
    frontiers = []
    reached = []
    initial_state = find_pos(maze, what = "S")
    goal_state = find_pos(maze, what = 'G')
    goal_states = [goal_state]
    goal_nodes = []
    # measure variables
    measure_path_cost = 0
    measure_node_expanded = 0
    measure_max_tree_depth = 0
    measure_max_tree_size = 0
    measure_max_frontier_size = 0

    if actions == "R":
        actions = {"N":(-1,0), "S":(1,0), "W":(0,-1), "E":(0,1)}
        l = list(actions.items())
        random.shuffle(l)
        actions = dict(l)

    width,height = maze.shape#width and height of maze
    root = Node(pos=initial_state,parent=None,action=None,cost=0)
    reached.append(root)
    frontiers.append(root)
    measure_node_expanded += 1

    while frontiers :
```

```
        if debug: show_path(maze = maze, result=(goal_nodes,reached,frontiers))

        if len(frontiers) > 1:
            frontiers.sort(key=lambda x: cost_evaluation(curPos = x.
↪pos,targetPos = goal_state))

        #pop a frontier
        frontier = frontiers.pop(0)

        if len(frontiers)+1 > measure_max_frontier_size:␣
↪measure_max_frontier_size = len(frontiers)+1 # max frontier size
        if len(reached) > measure_max_tree_size: measure_max_tree_size =␣
↪len(reached)# max tree size
        measure_node_expanded += 1
        if frontier.cost > measure_max_tree_depth : measure_max_tree_depth =␣
↪frontier.cost

        if is_goal(frontier,goal_states) and (frontier not in goal_nodes):
            goal_nodes.append(frontier)
            if len(goal_nodes) == len(goal_states):
                measure_path_cost = (sum(x.cost for x in goal_nodes))# path cost
                return␣
↪goal_nodes,reached,frontiers,[measure_path_cost,measure_node_expanded,measure_max_tree_depth
        else :
            for subNode in frontier.expand(maze=maze,actions=actions):
                if (subNode not in reached) and (subNode not in frontiers):
                    reached.append(subNode)
                    frontiers.append(subNode)
        limit_steps -= 1
        if limit_steps<=0: break
    #if there is not a solution, it returns empty of goal_nodes
    return␣
↪([],reached,[],[measure_path_cost,measure_node_expanded,measure_max_tree_depth,measure_max_
```

[25]:
```
%time result = GBS(maze=maze,debug=False)
show_path(maze,result)
```

```
CPU times: user 4.76 ms, sys: 1 µs, total: 4.76 ms
Wall time: 4.77 ms
Path cost: 29
Node expanded: 41
Max tree depth: 29
Max tree size: 44
Max frontier size: 5
```



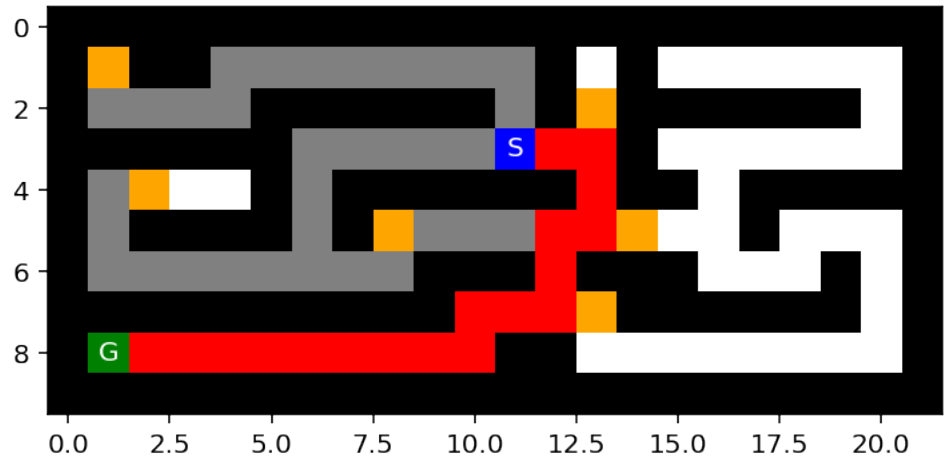
[26]:
```
# A* search
# maze: the maze
# limit_steps: stop the loop after limite_steps times
# debug: if it's True, it prints every step of walking through the maze
# actions: the order of generating sub node from frontier
def AStar(maze,limit_steps = 1000,debug=False,actions = {"N":(-1,0), "S":(1,0),␣
 ↪"W":(0,-1), "E":(0,1)}):
    frontiers = []
    reached = []
    initial_state = find_pos(maze, what = "S")
    goal_state = find_pos(maze, what = 'G')
    goal_states = [goal_state]
    goal_nodes = []
    # measure variables
    measure_path_cost = 0
    measure_node_expanded = 0
    measure_max_tree_depth = 0
    measure_max_tree_size = 0
    measure_max_frontier_size = 0

    if actions == "R":
        actions = {"N":(-1,0), "S":(1,0), "W":(0,-1), "E":(0,1)}
        l = list(actions.items())
        random.shuffle(l)
        actions = dict(l)

    width,height = maze.shape#width and height of maze

    root = Node(pos=initial_state,parent=None,action=None,cost=0)
    reached.append(root)
```

```
    frontiers.append(root)
    measure_node_expanded +=1

    while frontiers :
        if len(frontiers) > 1:
            frontiers.sort(key=lambda x: cost_evaluation(curPos = x.
↪pos,targetPos = goal_state, curCost=x.cost))

        if debug:
            show_path(maze = maze, result=(goal_nodes,reached,frontiers))
            for tmp_fron in frontiers: print(tmp_fron.pos," distance:␣
↪",cost_evaluation(tmp_fron.pos,goal_state,tmp_fron.cost))

        #pop a frontier
        frontier = frontiers.pop(0)

        if len(frontiers)+1 > measure_max_frontier_size:␣
↪measure_max_frontier_size = len(frontiers)+1 # max frontier size
        if len(reached) > measure_max_tree_size: measure_max_tree_size =␣
↪len(reached)# max tree size
        measure_node_expanded += 1
        if frontier.cost > measure_max_tree_depth : measure_max_tree_depth =␣
↪frontier.cost

        if is_goal(frontier,goal_states) and (frontier not in goal_nodes):
            goal_nodes.append(frontier)
            if len(goal_nodes) == len(goal_states):
                measure_path_cost = (sum(x.cost for x in goal_nodes))# path cost
                return␣
↪goal_nodes,reached,frontiers,[measure_path_cost,measure_node_expanded,measure_max_tree_depth
        else :
            for subNode in frontier.expand(maze=maze,actions=actions):
                if (subNode not in reached) and (subNode not in frontiers):
                    reached.append(subNode)
                    frontiers.append(subNode)

        limit_steps -= 1
        if limit_steps<=0: break
    #if there is not a solution, it returns empty of goal_nodes
    return␣
↪([],reached,[],[measure_path_cost,measure_node_expanded,measure_max_tree_depth,measure_max_
```

[27]:
```
%time result = AStar(maze=maze,debug=False)
show_path(maze,result)
```

```
CPU times: user 5.79 ms, sys: 19 µs, total: 5.81 ms
```

```
Wall time: 5.82 ms
Path cost: 19
Node expanded: 54
Max tree depth: 19
Max tree size: 59
Max frontier size: 8
```



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

The implementation of AStar is both complete and optimal, while the implementation of GBFS is complete but not optimal.

b is maximum branching factor, m is max depth of the tree, d is depth of the optimal solution

time complexity of GBFS: worst case O(b^m)

space complexity of GBFS: O(b^d) since it has the same reached data structure as in the BFS

time complexity of A*:$O(b$m)

space complexity of A*: o(b^d) since it has the same reached data structure as in the BFS

3.12 Task 4: Comparison and discussion [2 points]

Run experiments to compare the implemented algorithms and present the results as tables and charts.

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.

[28]:
```
# Add code, table, charts.
import pandas as pd
from IPython.display import display

def print_table_from_maze(mazeName):
    data = []
    maze = read_maze_by_name(mazeName)
    bfs_result = BFS(maze,limit_steps=2000)
    dfs_result = DFS(maze,limit_steps=2000)
    gbs_result = GBS(maze,limit_steps=2000)
    astar_result = AStar(maze,limit_steps=2000)
    data = [bfs_result[3],dfs_result[3],gbs_result[3],astar_result[3]]
    print(mazeName)
    df = pd.DataFrame(data=data,index=["BFS","DFS",'GBS','A*'],columns=["path␣
 ↪cost",'nodes expanded','max tree depth','max tree size','max frontier size'])
    df.replace(0,"N/A",inplace=True)
    display(df)
    print("BFS")
    show_path(maze,bfs_result)
    print("DFS")
    show_path(maze,dfs_result)
    print("GBS")
    show_path(maze,gbs_result)
    print('A*')
    show_path(maze,astar_result)
```

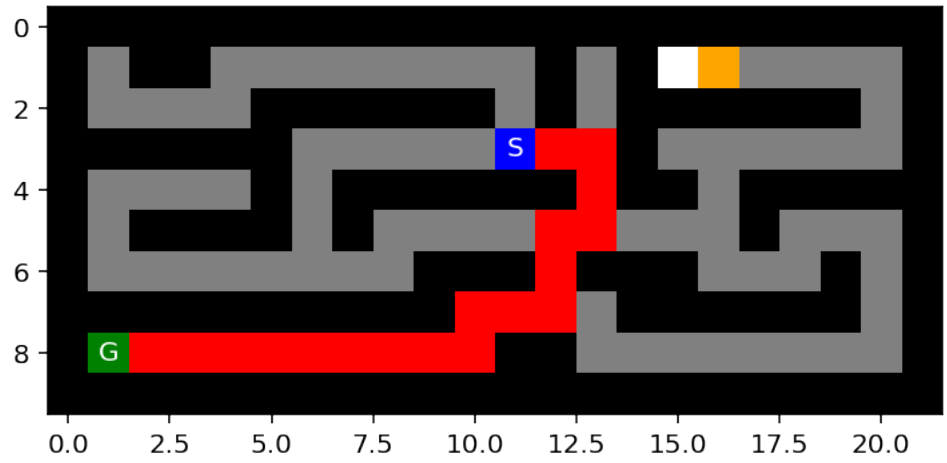
[29]:
```
print_table_from_maze("small_maze")
```

```
small_maze
```

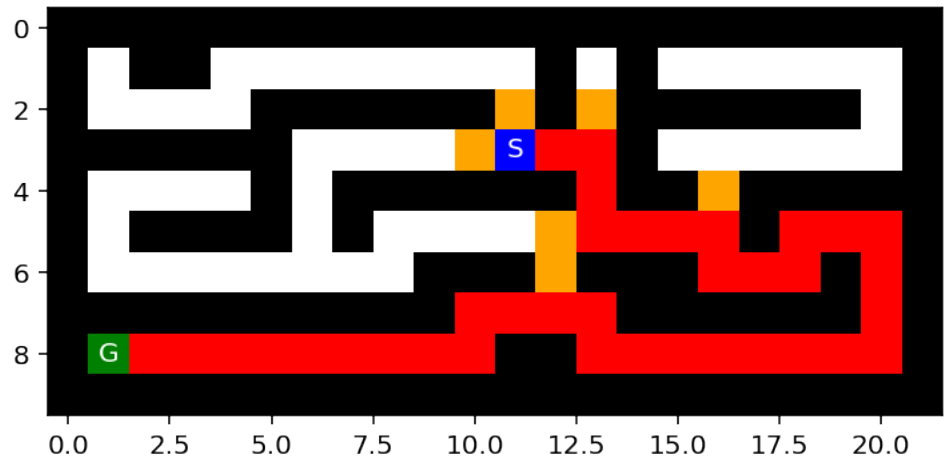
	path cost	nodes expanded	max tree depth	max tree size	max frontier size
BFS	19	93	19	93	8
DFS	37	39	37	44	7
GBS	29	41	29	44	5
A*	19	54	19	59	8

```
BFS
Path cost: 19
Node expanded: 93
Max tree depth: 19
Max tree size: 93
```

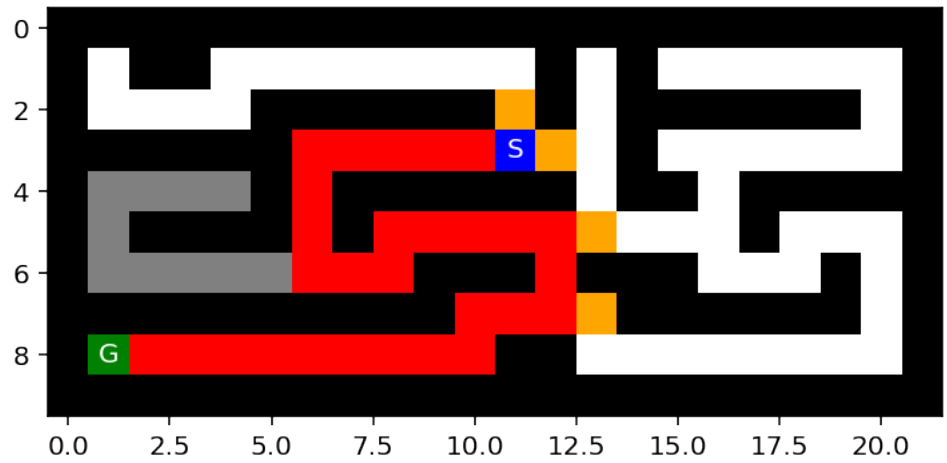
```
Max frontier size: 8
```



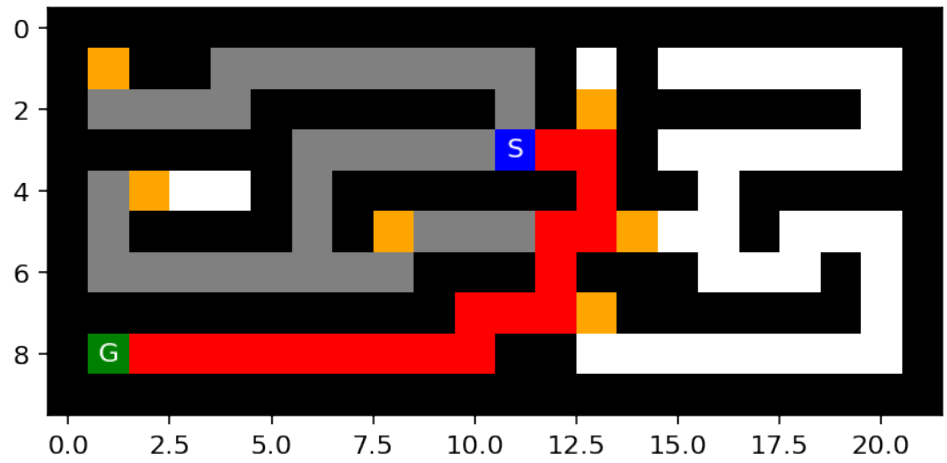

```
DFS
Path cost: 37
Node expanded: 39
Max tree depth: 37
Max tree size: 44
Max frontier size: 7
```




```
GBS
Path cost: 29
Node expanded: 41
Max tree depth: 29
Max tree size: 44
Max frontier size: 5
```



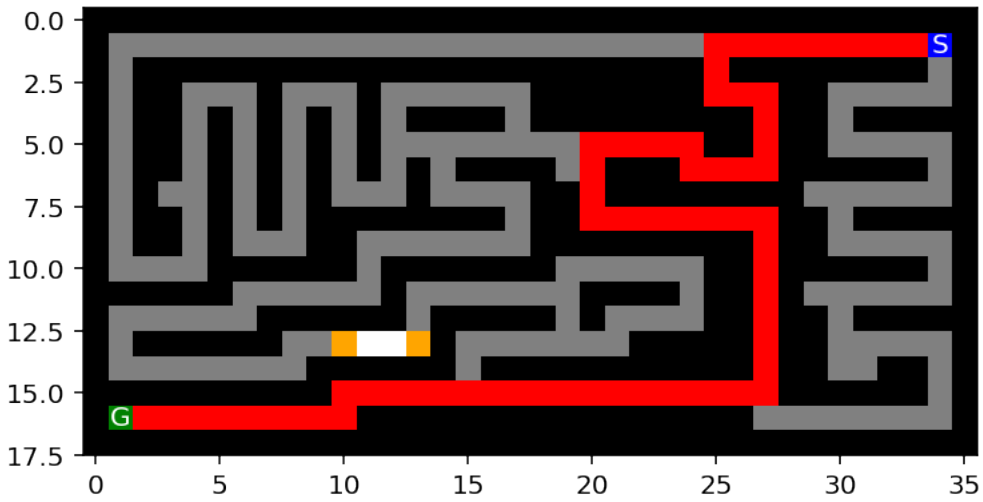

```
A*
Path cost: 19
Node expanded: 54
Max tree depth: 19
Max tree size: 59
Max frontier size: 8
```



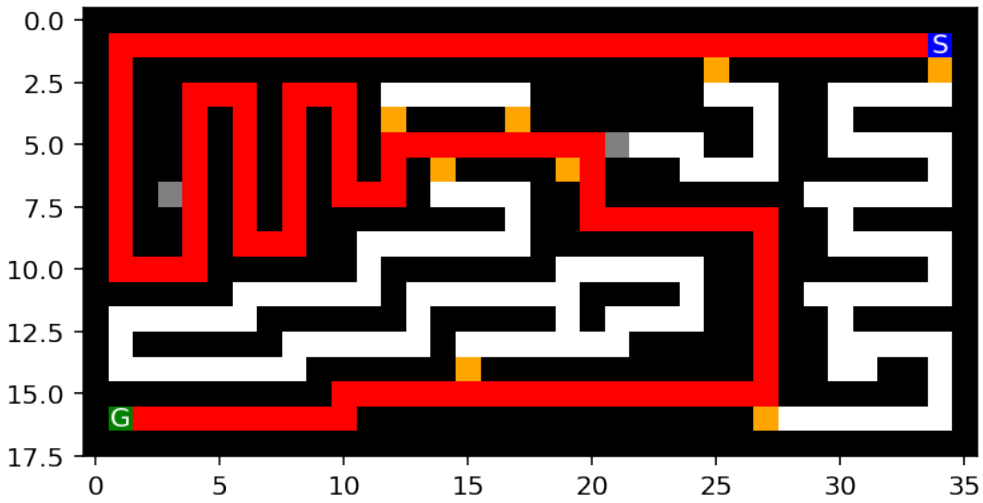
[30]:
```
print_table_from_maze("medium_maze")
```

```
medium_maze

     path cost  nodes expanded  max tree depth  max tree size  \
BFS         68             271              68            272
DFS        130             146             130            141
GBS         74              80              74             82
```

```
A*          68              223                 68              228

     max frontier size
BFS                  8
DFS                  9
GBS                  4
A*                   8

BFS
Path cost: 68
Node expanded: 271
Max tree depth: 68
Max tree size: 272
Max frontier size: 8
```



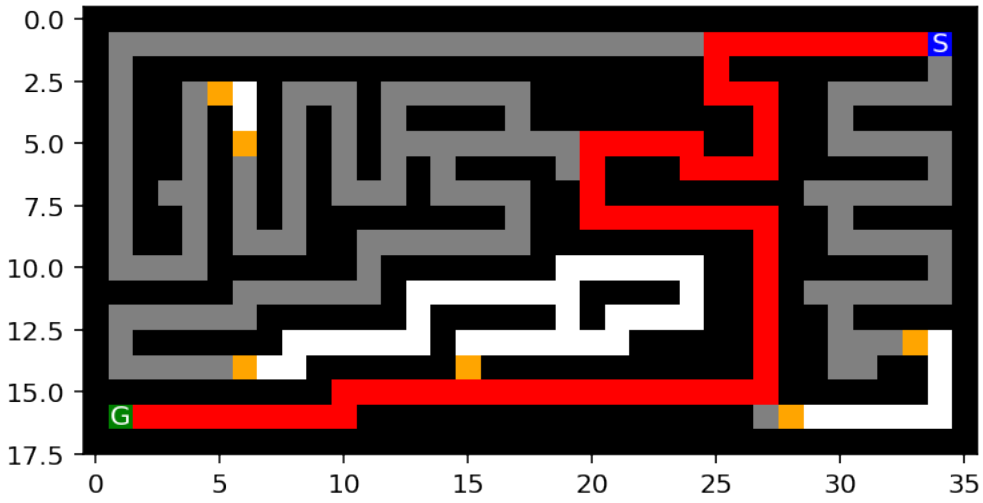

```
DFS
Path cost: 130
Node expanded: 146
Max tree depth: 130
Max tree size: 141
Max frontier size: 9
```




```
GBS
Path cost: 74
Node expanded: 80
Max tree depth: 74
Max tree size: 82
Max frontier size: 4
```

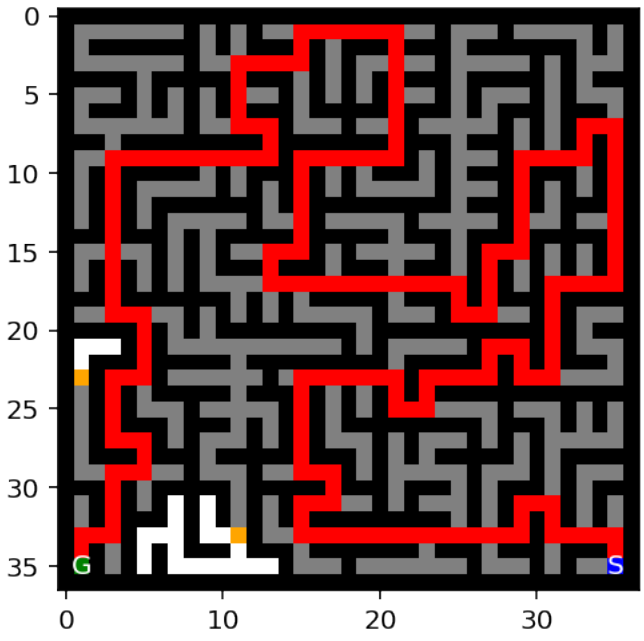

```
A*
Path cost: 68
Node expanded: 223
Max tree depth: 68
Max tree size: 228
```

```
Max frontier size: 8
```

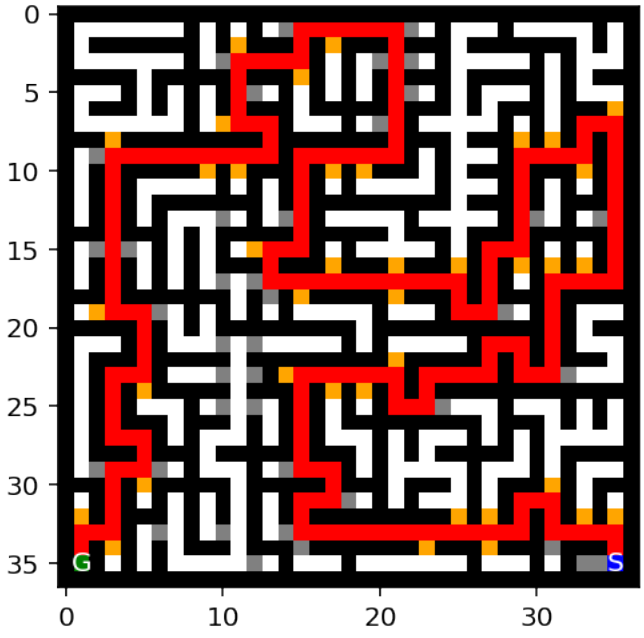


[31]:
```
print_table_from_maze("large_maze")
```

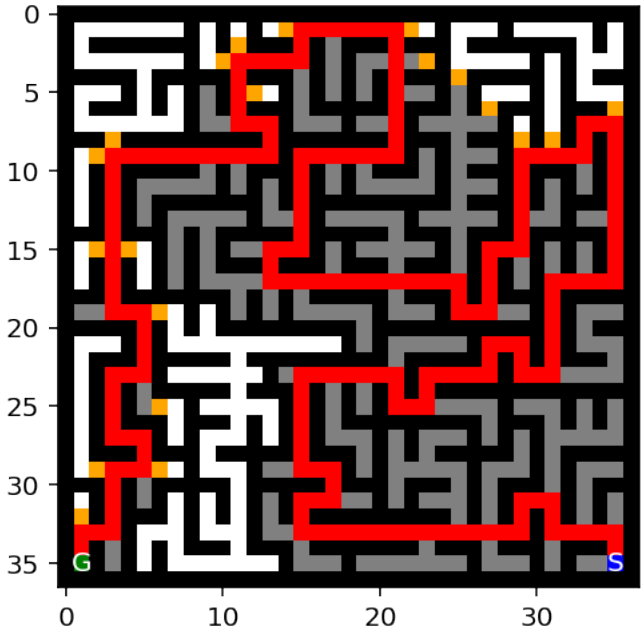
```
large_maze
     path cost  nodes expanded  max tree depth  max tree size  \
BFS        210             622             210            623
DFS        210             389             222            295
GBS        210             468             210            487
A*         210             551             210            557

     max frontier size
BFS                  8
DFS                 39
GBS                 21
A*                  12

BFS
Path cost: 210
Node expanded: 622
Max tree depth: 210
Max tree size: 623
Max frontier size: 8
```



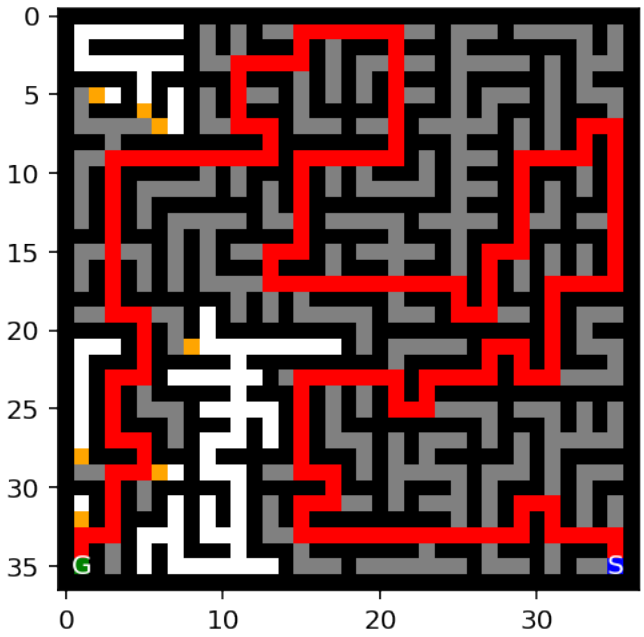

```
DFS
Path cost: 210
Node expanded: 389
Max tree depth: 222
Max tree size: 295
Max frontier size: 39
```



GBS
Path cost: 210
Node expanded: 468
Max tree depth: 210
Max tree size: 487
Max frontier size: 21

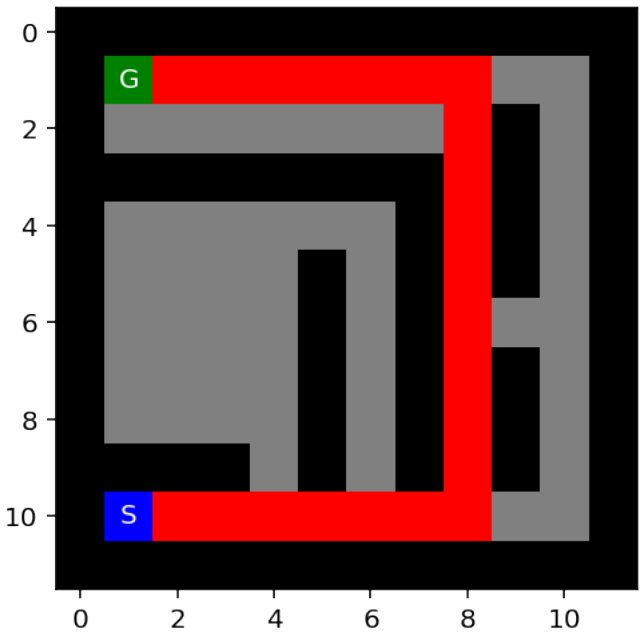


A*
Path cost: 210
Node expanded: 551
Max tree depth: 210
Max tree size: 557
Max frontier size: 12

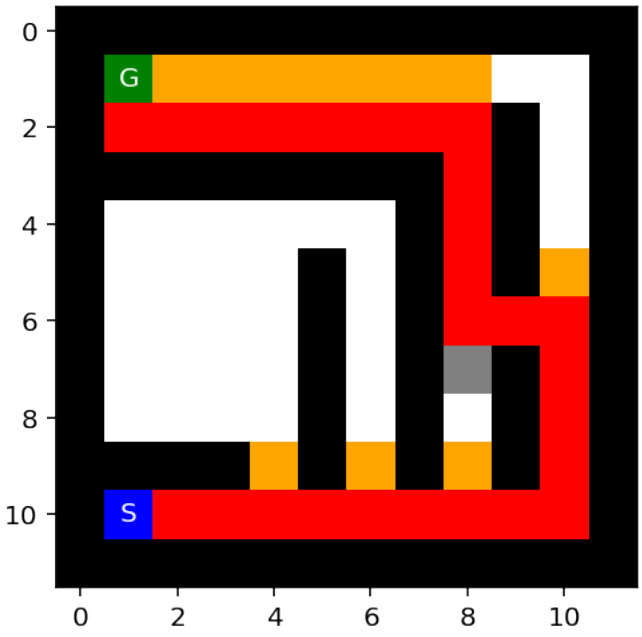


[32]:
```
print_table_from_maze("loops_maze")
```

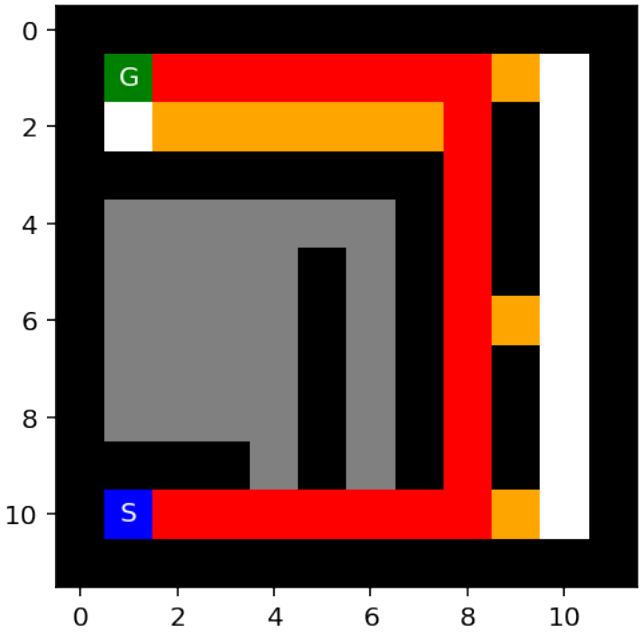
```
loops_maze

     path cost  nodes expanded  max tree depth  max tree size  \
BFS         23              73              23             72
DFS         27              31              27             40
GBS         23              53              23             61
A*          23              60              23             62

     max frontier size
BFS                  8
DFS                 12
GBS                 10
A*                   6

BFS
Path cost: 23
Node expanded: 73
Max tree depth: 23
Max tree size: 72
Max frontier size: 8
```



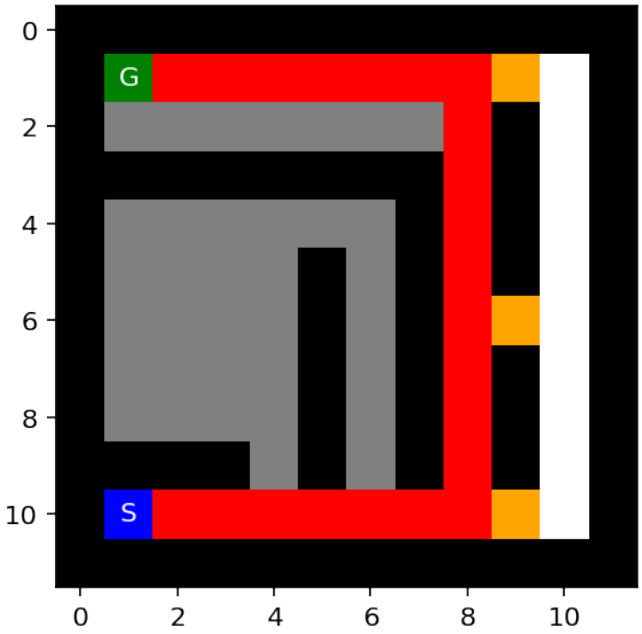

```
DFS
Path cost: 27
Node expanded: 31
Max tree depth: 27
Max tree size: 40
Max frontier size: 12
```



```
GBS
Path cost: 23
Node expanded: 53
Max tree depth: 23
Max tree size: 61
Max frontier size: 10
```




```
A*
Path cost: 23
Node expanded: 60
Max tree depth: 23
Max tree size: 62
Max frontier size: 6
```



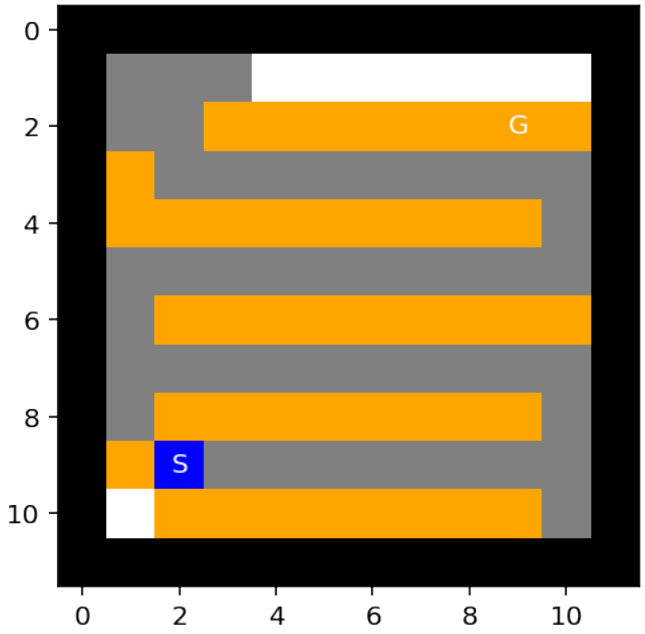
3.13 In this "empty_maze", the DFS sometimes can't get a solution because it gets stuck in the cycle.

[33]:
```
result =␣
 ↪DFS(read_maze_by_name('empty_maze'),debug=True,vis=False,limit_steps=70)
show_path(read_maze_by_name('empty_maze'),result)
#It can be seen in debugging output, the pos (1,1) was ever popped from the␣
 ↪frontier and then added again. Because it didn't use the reached data␣
 ↪structure
```

```
each pop frontier (9, 2)
each pop frontier (9, 3)
each pop frontier (9, 4)
each pop frontier (9, 5)
each pop frontier (9, 6)
each pop frontier (9, 7)
each pop frontier (9, 8)
each pop frontier (9, 9)
each pop frontier (9, 10)
each pop frontier (10, 10)
each pop frontier (8, 10)
each pop frontier (7, 10)
each pop frontier (7, 9)
each pop frontier (7, 8)
each pop frontier (7, 7)
each pop frontier (7, 6)
```

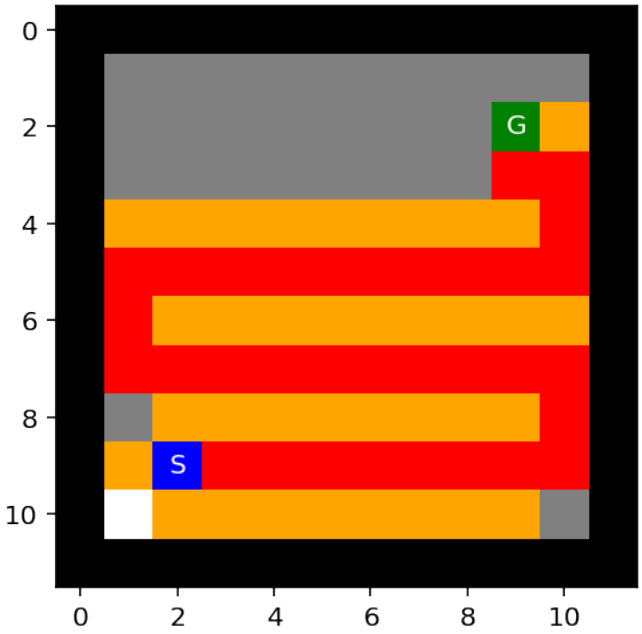
```
each pop frontier (7, 5)
each pop frontier (7, 4)
each pop frontier (7, 3)
each pop frontier (7, 2)
each pop frontier (7, 1)
each pop frontier (8, 1)
each pop frontier (6, 1)
each pop frontier (5, 1)
each pop frontier (5, 2)
each pop frontier (5, 3)
each pop frontier (5, 4)
each pop frontier (5, 5)
each pop frontier (5, 6)
each pop frontier (5, 7)
each pop frontier (5, 8)
each pop frontier (5, 9)
each pop frontier (5, 10)
each pop frontier (4, 10)
each pop frontier (3, 10)
each pop frontier (3, 9)
each pop frontier (3, 8)
each pop frontier (3, 7)
each pop frontier (3, 6)
each pop frontier (3, 5)
each pop frontier (3, 4)
each pop frontier (3, 3)
each pop frontier (3, 2)
each pop frontier (3, 1)
each pop frontier (2, 1)
each pop frontier (1, 1)
each pop frontier (1, 2)
each pop frontier (1, 3)
each pop frontier (1, 4)
each pop frontier (1, 5)
each pop frontier (1, 6)
each pop frontier (1, 7)
each pop frontier (1, 8)
each pop frontier (1, 9)
each pop frontier (1, 10)
each pop frontier (2, 2)
each pop frontier (2, 1)
each pop frontier (3, 1)
each pop frontier (1, 1)
each pop frontier (1, 2)
each pop frontier (1, 3)
each pop frontier (1, 4)
each pop frontier (1, 5)
each pop frontier (1, 6)
```

```
each pop frontier (1, 7)
each pop frontier (1, 8)
each pop frontier (1, 9)
each pop frontier (1, 10)
each pop frontier (1, 1)
each pop frontier (2, 1)
Path cost: 0
Node expanded: 71
Max tree depth: 52
Max tree size: 99
Max frontier size: 46
```



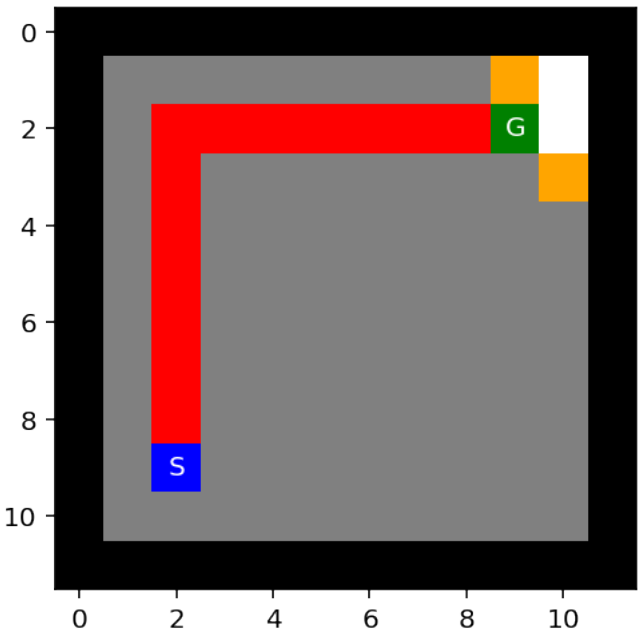
[34]:
```
#if I use DFS_with_reached, then it comes out a solution. It's not the␣
 ↪direction problem since the default direction is not random.
result =␣
 ↪DFS_with_reached(read_maze_by_name('empty_maze'),debug=False,vis=False,limit_steps=70)
show_path(read_maze_by_name('empty_maze'),result)
```

```
Path cost: 34
Node expanded: 64
Max tree depth: 52
Max tree size: 99
Max frontier size: 45
```

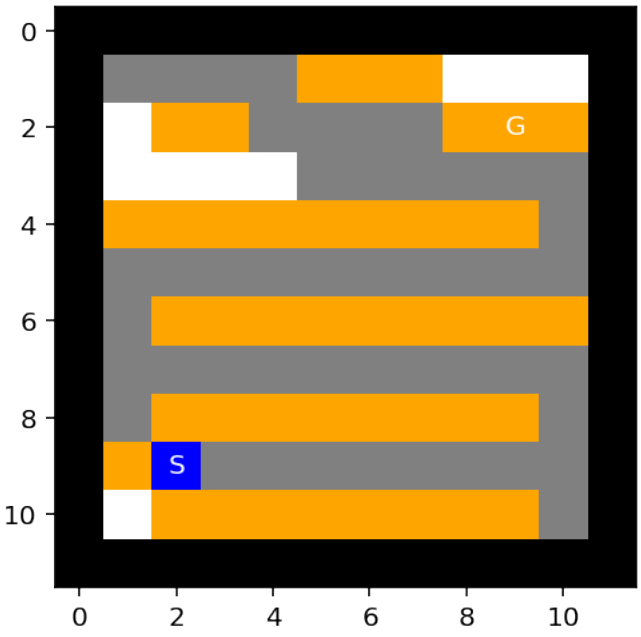


[35]:
```
print_table_from_maze('empty_maze')
```

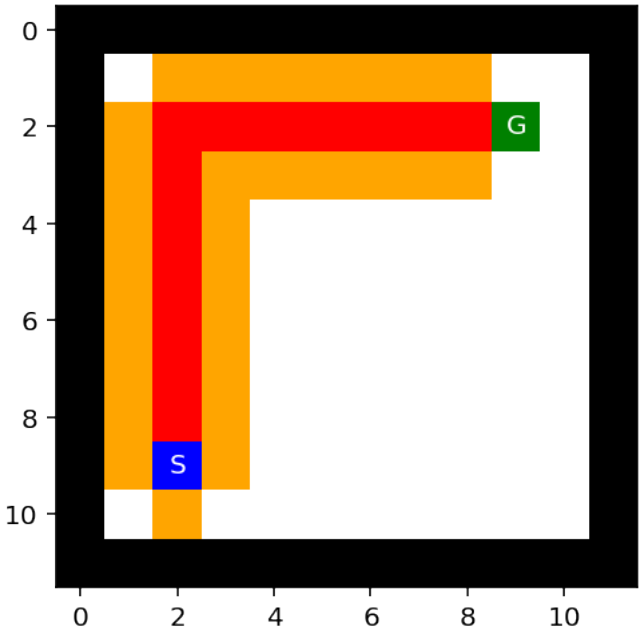
```
empty_maze

    path cost  nodes expanded  max tree depth  max tree size  \
BFS        14              97              14             98
DFS       N/A            2001              53             99
GBS        14              16              14             43
A*         14              65              14             94

     max frontier size
BFS                 12
DFS                 51
GBS                 29
A*                  31

BFS
Path cost: 14
Node expanded: 97
Max tree depth: 14
Max tree size: 98
Max frontier size: 12
```



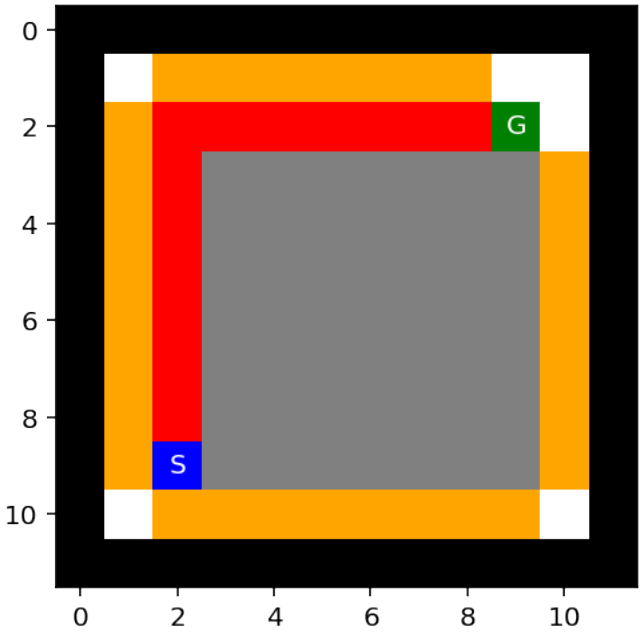

```
DFS
Path cost: 0
Node expanded: 2001
Max tree depth: 53
Max tree size: 99
Max frontier size: 51
```



```
GBS
Path cost: 14
Node expanded: 16
Max tree depth: 14
Max tree size: 43
Max frontier size: 29
```

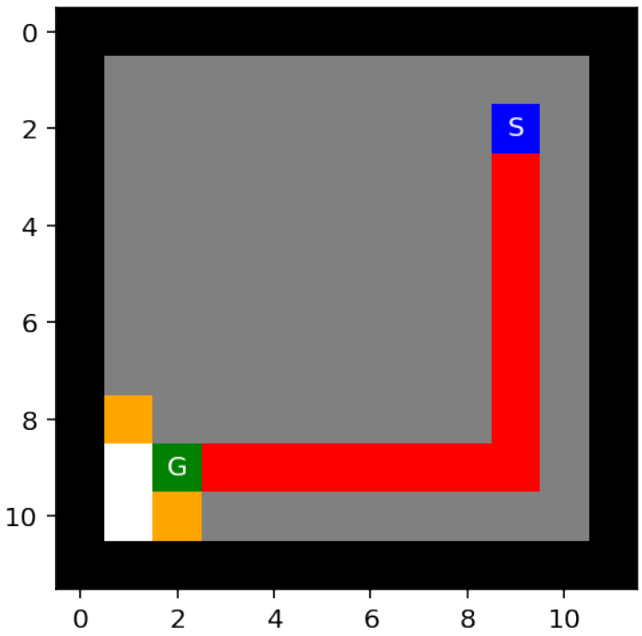



```
A*
Path cost: 14
Node expanded: 65
Max tree depth: 14
Max tree size: 94
Max frontier size: 31
```

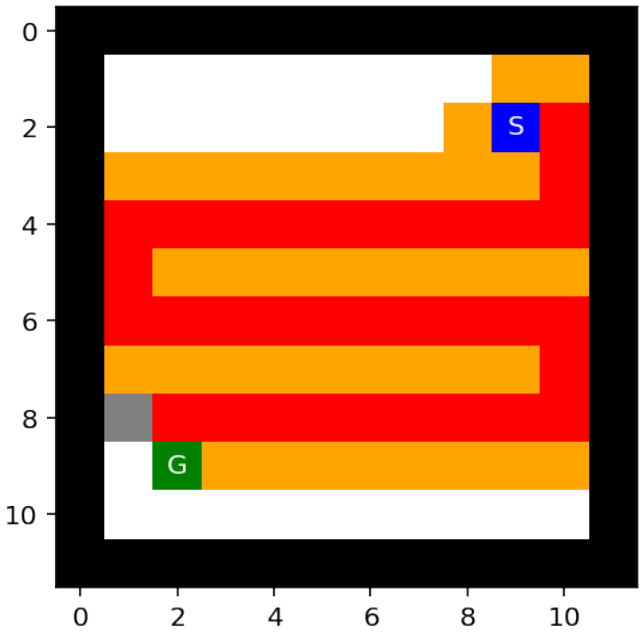


[36]:
```
print_table_from_maze('empty_2_maze')
```

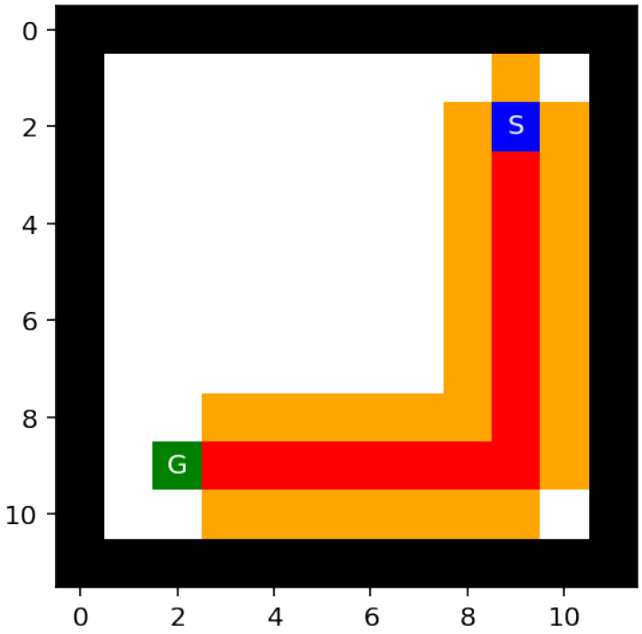
```
empty_2_maze

     path cost  nodes expanded  max tree depth  max tree size  \
BFS         14              97              14             98
DFS         34              48              45             85
GBS         14              16              14             43
A*          14              65              14             94

     max frontier size
BFS                 12
DFS                 40
GBS                 29
A*                  31

BFS
Path cost: 14
Node expanded: 97
Max tree depth: 14
Max tree size: 98
Max frontier size: 12
```



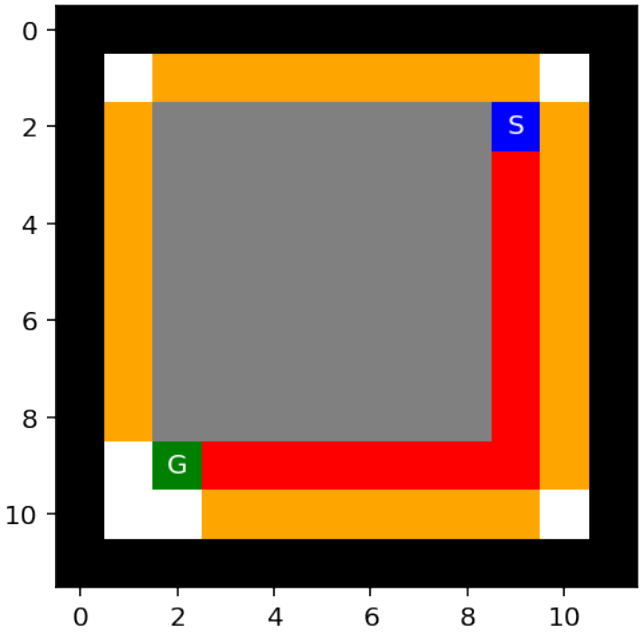

```
DFS
Path cost: 34
Node expanded: 48
Max tree depth: 45
Max tree size: 85
Max frontier size: 40
```



```
GBS
Path cost: 14
Node expanded: 16
Max tree depth: 14
Max tree size: 43
Max frontier size: 29
```



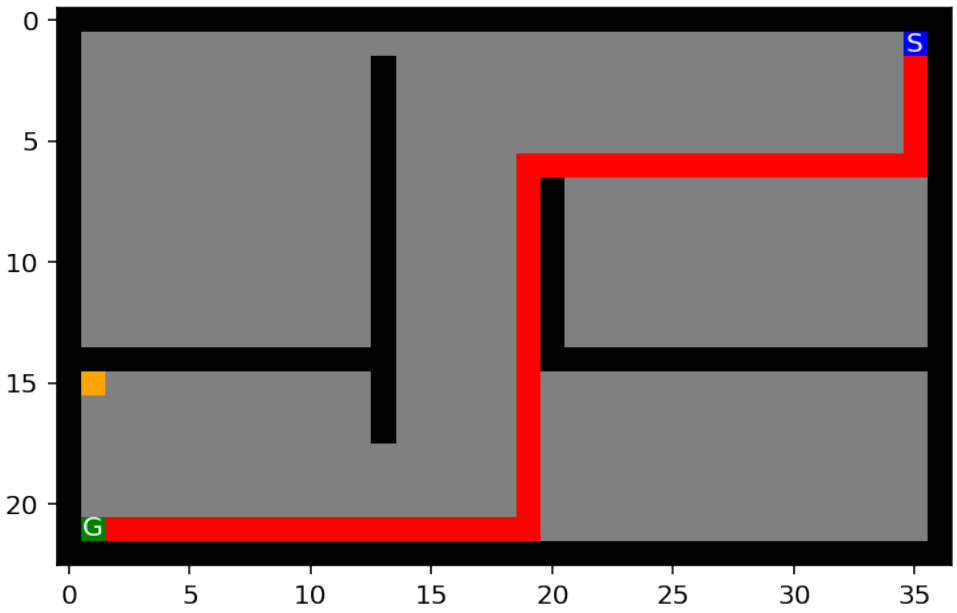

```
A*
Path cost: 14
Node expanded: 65
Max tree depth: 14
Max tree size: 94
Max frontier size: 31
```



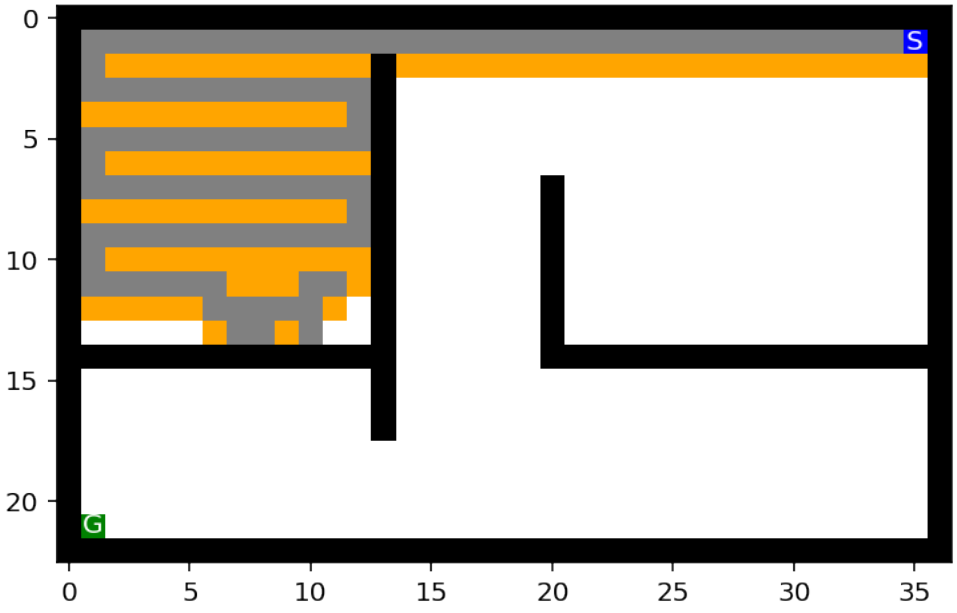
3.14 In this "open_maze", the DFS one has the same problem with the maze "empty_maze", which was aforementioned.

[37]:
```
print_table_from_maze('open_maze')
```

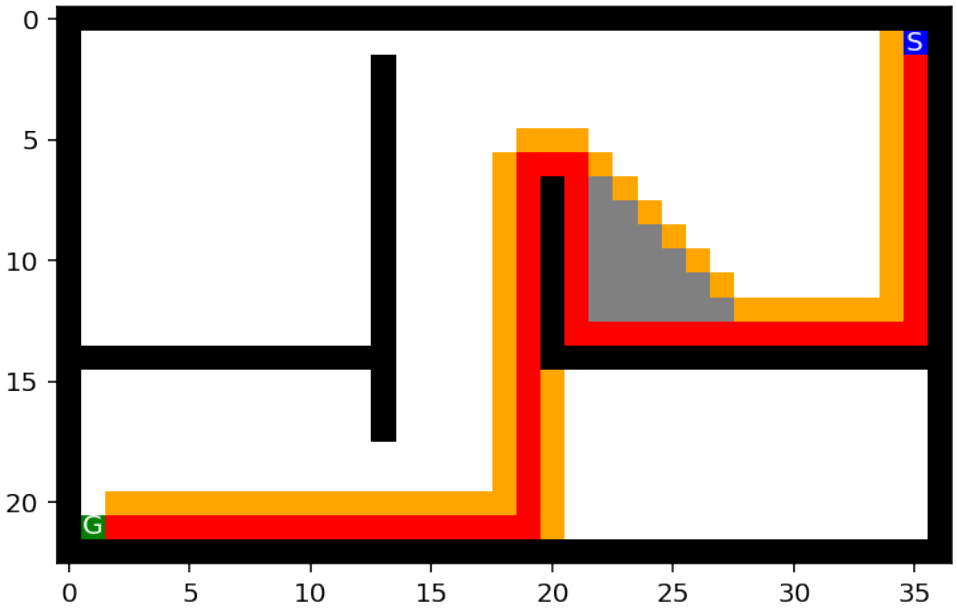
```
open_maze

    path cost  nodes expanded  max tree depth  max tree size  \
BFS        54             684              54            684
DFS       N/A            2001             112            201
GBS        68              91              68            155
A*         54             537              54            555

     max frontier size
BFS                 25
DFS                 95
GBS                 66
A*                  25

BFS
Path cost: 54
Node expanded: 684
Max tree depth: 54
Max tree size: 684
Max frontier size: 25
```



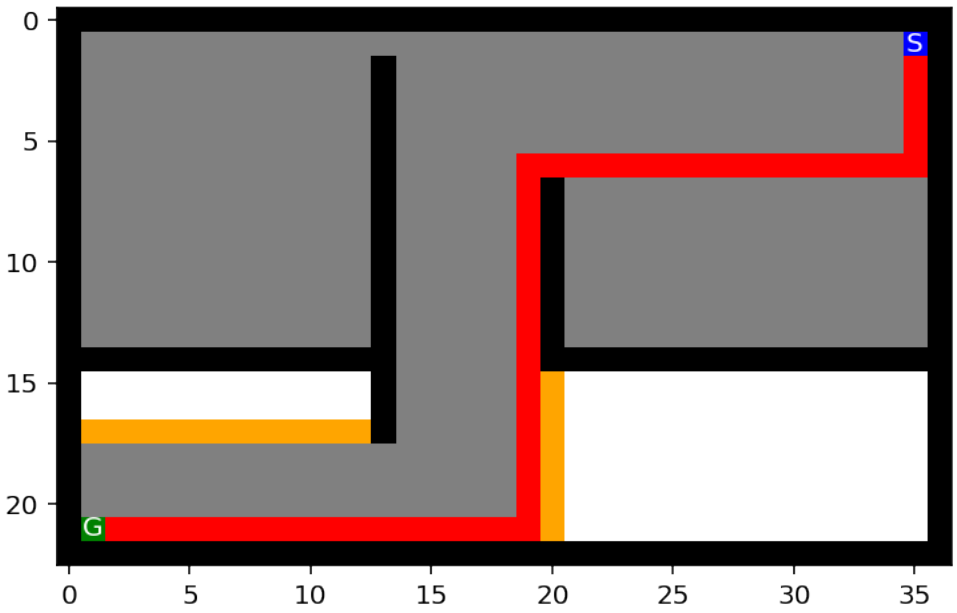

```
DFS
Path cost: 0
Node expanded: 2001
Max tree depth: 112
Max tree size: 201
Max frontier size: 95
```



```
GBS
Path cost: 68
Node expanded: 91
Max tree depth: 68
Max tree size: 155
Max frontier size: 66
```



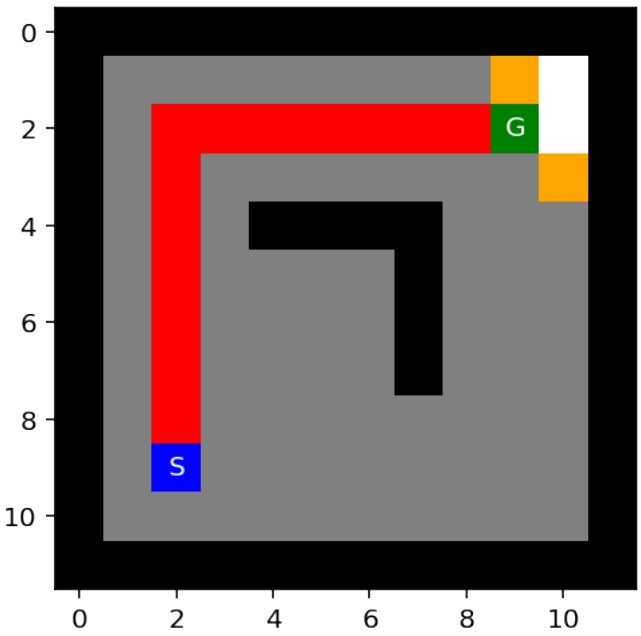

```
A*
Path cost: 54
Node expanded: 537
Max tree depth: 54
Max tree size: 555
Max frontier size: 25
```



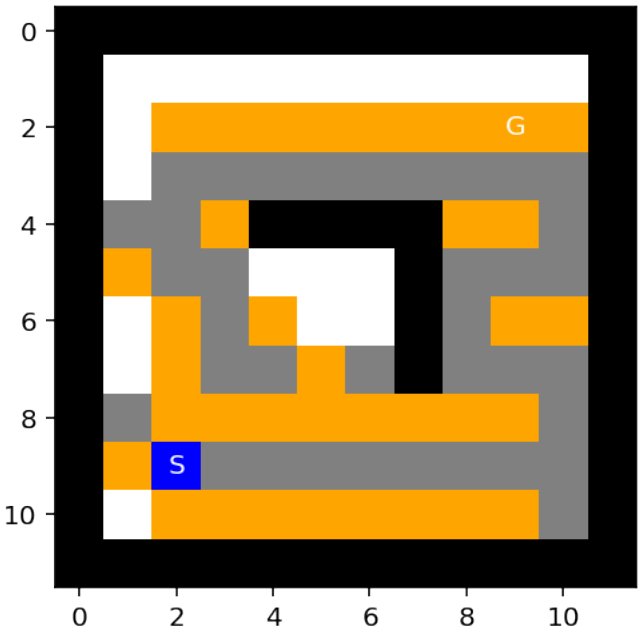
3.15 In this "wall_maze" maze, also the same problem.

[38]:
```
print_table_from_maze('wall_maze')
```

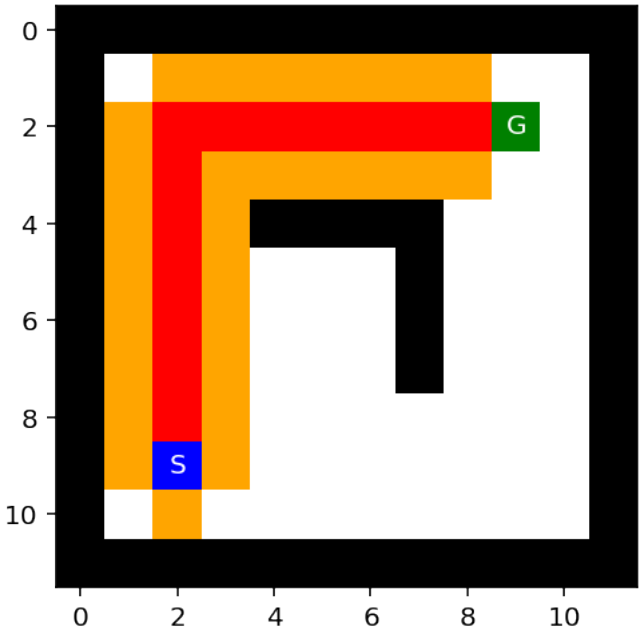
```
wall_maze

    path cost  nodes expanded  max tree depth  max tree size  \
BFS        14              90              14             91
DFS       N/A            2001              46             82
GBS        14              16              14             43
A*         14              58              14             87

     max frontier size
BFS                 11
DFS                 44
GBS                 29
A*                  31

BFS
Path cost: 14
Node expanded: 90
Max tree depth: 14
Max tree size: 91
Max frontier size: 11
```



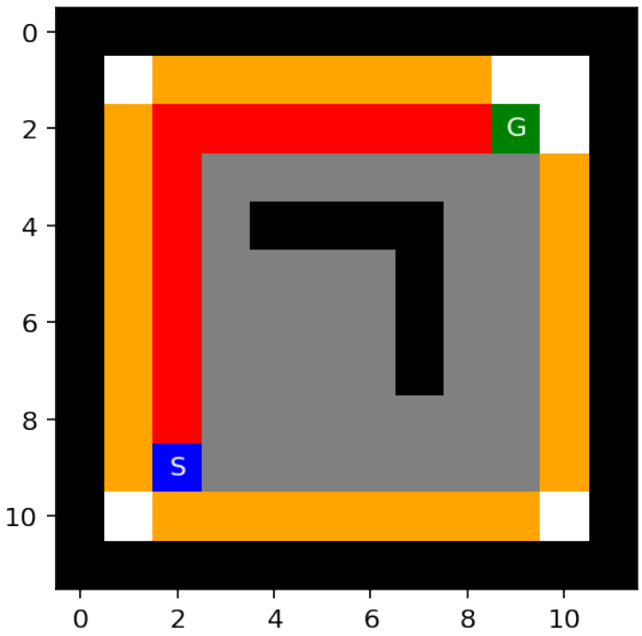

```
DFS
Path cost: 0
Node expanded: 2001
Max tree depth: 46
Max tree size: 82
Max frontier size: 44
```



```
GBS
Path cost: 14
Node expanded: 16
Max tree depth: 14
Max tree size: 43
Max frontier size: 29
```

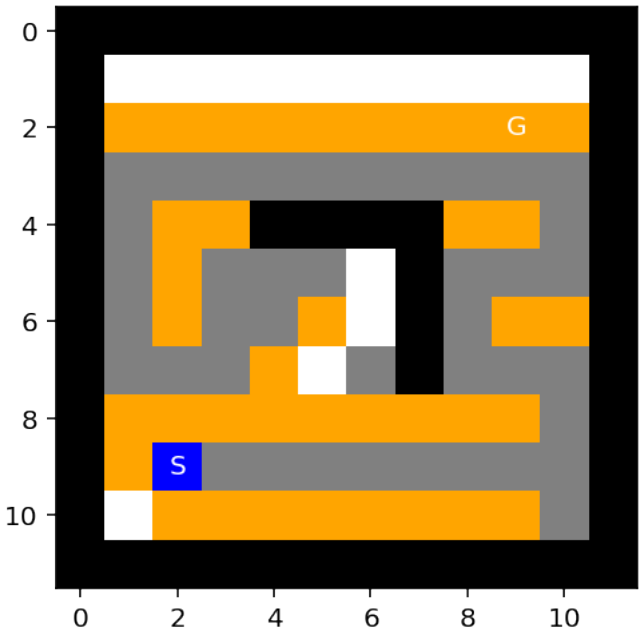



```
A*
Path cost: 14
Node expanded: 58
Max tree depth: 14
Max tree size: 87
Max frontier size: 31
```



[39]:
```
show_path(read_maze_by_name('wall_maze'),DFS(read_maze_by_name('wall_maze'),debug=False))
```

```
Path cost: 0
Node expanded: 1001
Max tree depth: 44
Max tree size: 82
Max frontier size: 44
```



Complete the following table for each maze.

Small maze

algorithm	path cost	nodes expanded	max tree depth	max tree size	max frontier size
BFS					
DFS					
GBS					
A*					

Medium Maze

...

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

A* performs well on both space usage and time consumption, when comparing to the BFS.

BFS performes well in the small size of maze. However, in the big maze, it is not recommended.

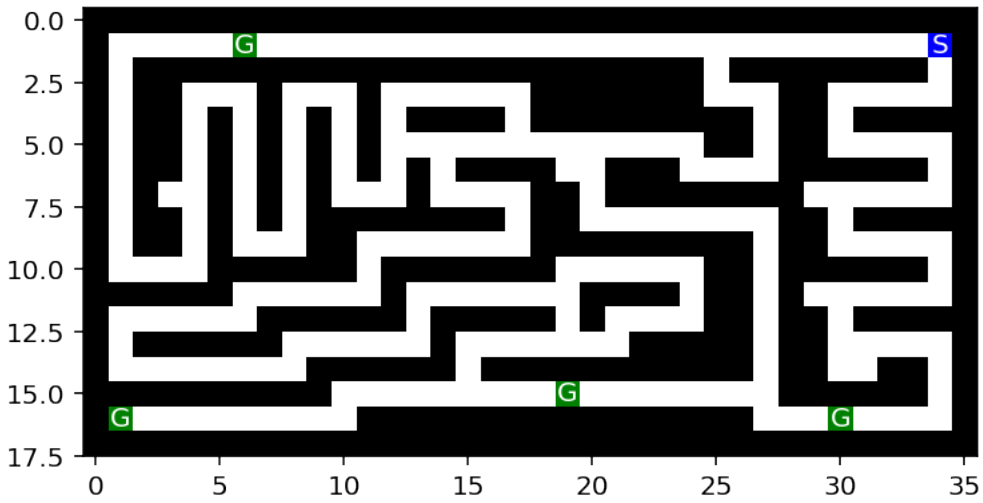
DFS without reached data structure has an advantage of space usage, but it don't always provide a solution.

3.16 Graduate student advanced task: Multiple Goals [1 point]

Undergraduate students: This is a bonus task you can attempt if you like [+1 Bonus point].

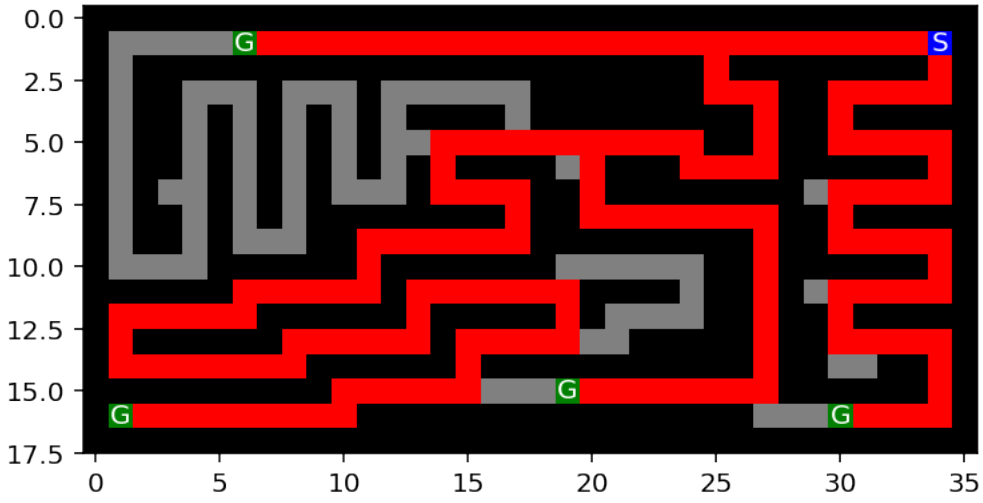
Create a few mazes with multiple goals by adding one or two more goals to the medium size maze. Solve the maze with your implementations for DFS, BFS, and implement IDS (iterative deepening search using DFS). Run experiments to show which implementations find the optimal solution.

[40]:
```
# Your code/answer goes here
maze_m = read_maze_by_name("medium_maze_multiple_goals")
show_maze_v2(maze_m)
```



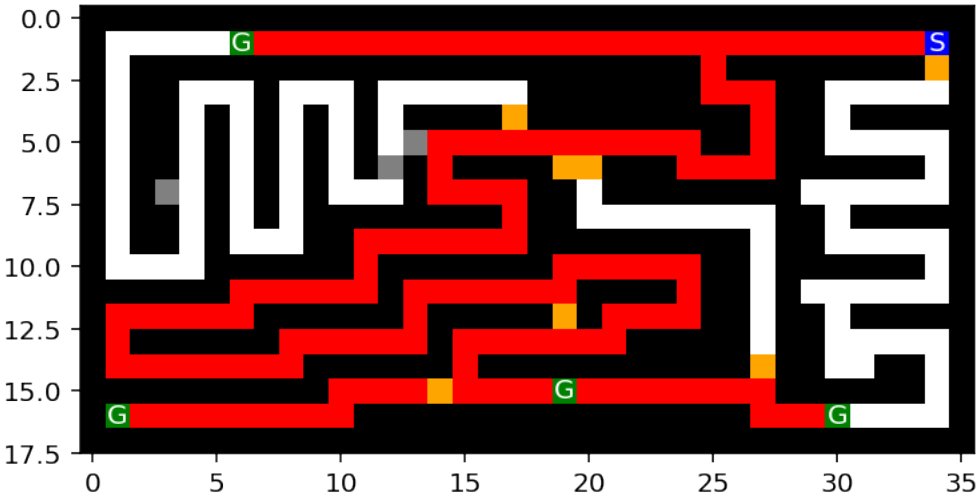
[41]:
```
%time result_m = BFS(maze_m)
show_path(maze_m,result_m)
```

```
CPU times: user 14.4 ms, sys: 325 µs, total: 14.7 ms
Wall time: 14.4 ms
Path cost: 222
Node expanded: 275
Max tree depth: 102
Max tree size: 274
Max frontier size: 7
```



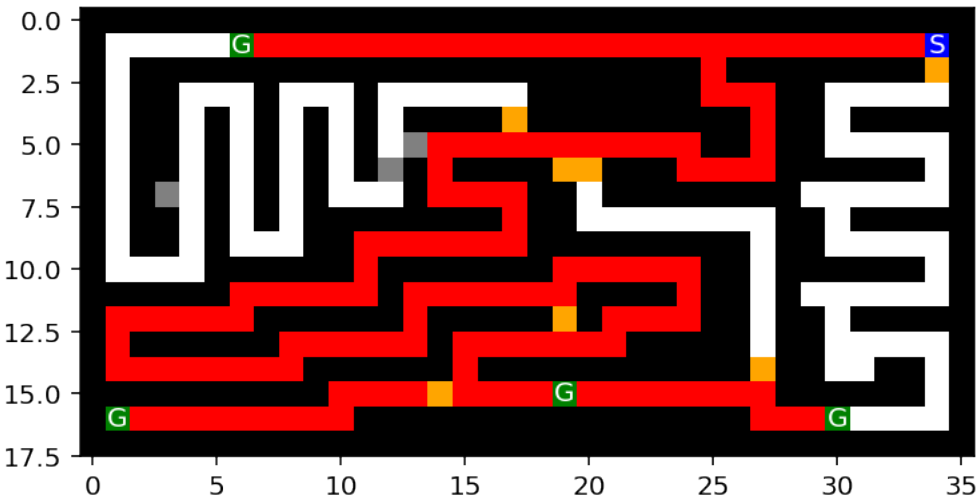
[42]:
```
%time result_m = DFS(maze_m)
show_path(maze_m,result_m)
```

```
CPU times: user 83.3 ms, sys: 668 µs, total: 84 ms
Wall time: 83.4 ms
Path cost: 336
Node expanded: 253
Max tree depth: 115
Max tree size: 140
Max frontier size: 8
```



[43]:
```
%time result_m = IDS(maze_m,depth_limit=100)
show_path(maze_m,result_m)
```

```
CPU times: user 85.1 ms, sys: 710 µs, total: 85.8 ms
Wall time: 85.2 ms
Path cost: 336
Node expanded: 253
Max tree depth: 115
Max tree size: 140
Max frontier size: 8
```



3.17 More advanced tasks to think about

Instead of defining each square as a state (which you probably did), 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 in 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.

[44]:
```
# Your code/answer goes here
```

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

[45]:
```
# 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?

[46]:
```
# Your code/answer goes here
```