Artificial Intelligence and Computer Vision Laboratory

Report from laboratory 5

Ewa Kobiela

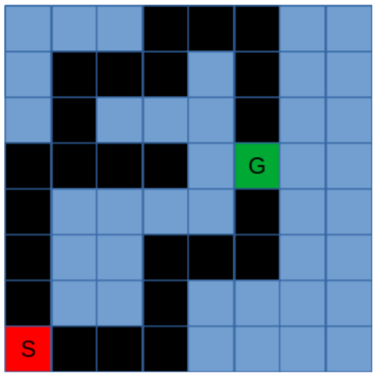
1 Scope of the document

This document is written as a documentation of tasks prepared for 5th laboratory of Artificial Intelligence and Computer Vision. It contains detailed descriptions of three programs designed to solve maze problems using Depth First Search, Bredth First Search and A* Algorithm.

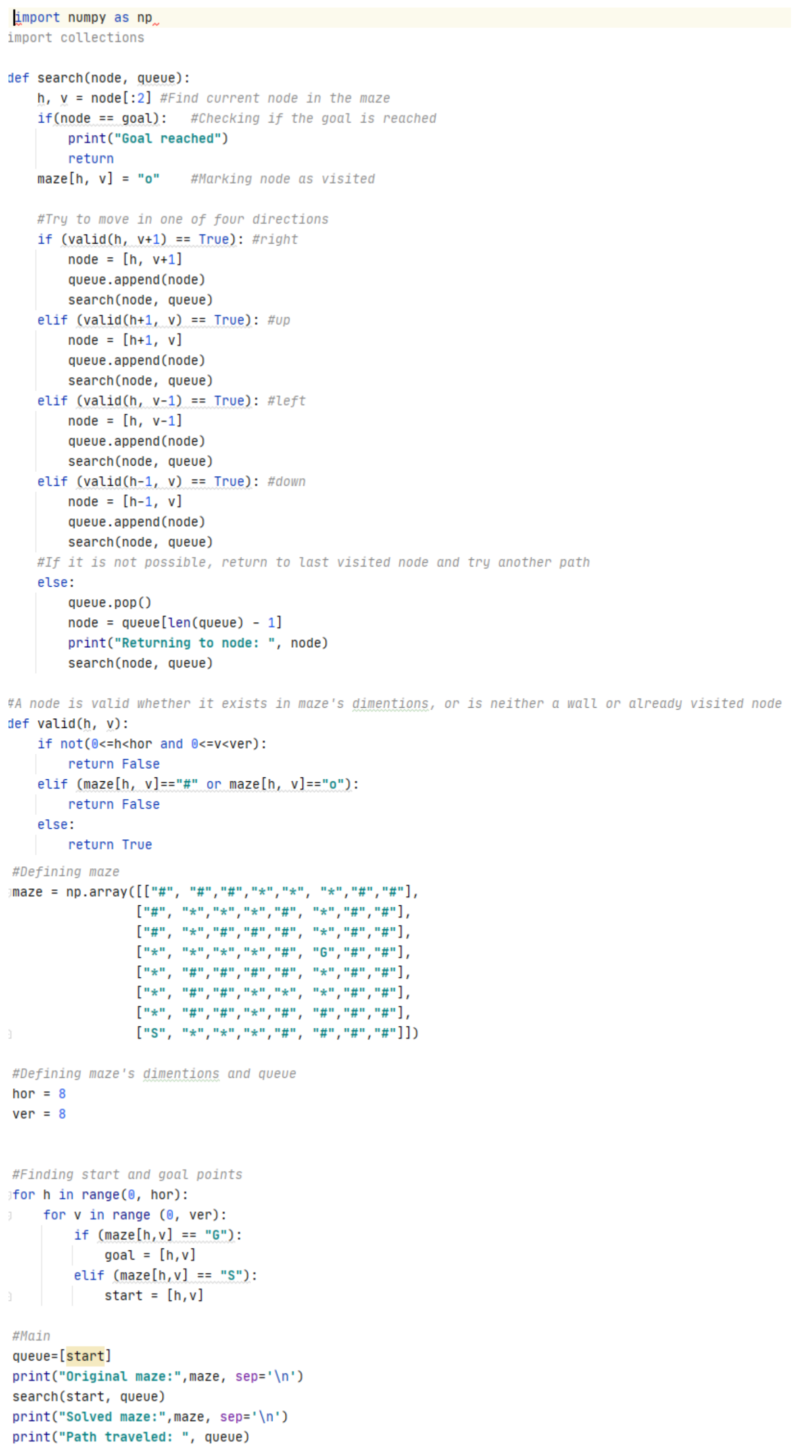
2 Depth First Search

The algorithm of Depth First Search is based on the idea of exploring only one path at a time. In my attempt a stack was used as a data structure to store information about path taken as its property ”last in - first out” perfectly satisfies the requirements.

The program takes a predefined maze as an input and returns a list of steps that are need to be taken in order to travel from the start point to the goal point. The maze is presented below in a graphical form.



The code written to perform the path finding is depicted below (all scripts mentioned in this document are attached to the file).



```python
import numpy as np
import collections

def search(node, queue):
    h, v = node[:2] #Find current node in the maze
    if(node == goal):   #Checking if the goal is reached
        print("Goal reached")
        return
    maze[h, v] = "o"    #Marking node as visited

    #Try to move in one of four directions
    if (valid(h, v+1) == True): #right
        node = [h, v+1]
        queue.append(node)
        search(node, queue)
    elif (valid(h+1, v) == True): #up
        node = [h+1, v]
        queue.append(node)
        search(node, queue)
    elif (valid(h, v-1) == True): #left
        node = [h, v-1]
        queue.append(node)
        search(node, queue)
    elif (valid(h-1, v) == True): #down
        node = [h-1, v]
        queue.append(node)
        search(node, queue)
    #If it is not possible, return to last visited node and try another path
    else:
        queue.pop()
        node = queue[len(queue) - 1]
        print("Returning to node: ", node)
        search(node, queue)

#A node is valid whether it exists in maze's dimentions, or is neither a wall or already visited node
def valid(h, v):
    if not(0<=h<hor and 0<=v<ver):
        return False
    elif (maze[h, v]=="#" or maze[h, v]=="o"):
        return False
    else:
        return True
#Defining maze
maze = np.array([["#", "#","#","*","*", "*","#","#"],
                ["#", "*","*","*","#", "*","#","#"],
                ["#", "*","#","#","#", "*","#","#"],
                ["*", "*","*","*","#", "G","#","#"],
                ["*", "#","#","#","#", "*","#","#"],
                ["*", "#","#","*","*", "*","#","#"],
                ["*", "#","#","*","#", "#","#","#"],
                ["S", "*","*","*","#", "#","#","#"]])

#Defining maze's dimentions and queue
hor = 8
ver = 8


#Finding start and goal points
for h in range(0, hor):
    for v in range (0, ver):
        if (maze[h,v] == "G"):
            goal = [h,v]
        elif (maze[h,v] == "S"):
            start = [h,v]

#Main
queue=[start]
print("Original maze:",maze, sep='\n')
search(start, queue)
print("Solved maze:",maze, sep='\n')
print("Path traveled: ", queue)
```

The code consists of a main function and several functions to be called to perform a maze solving. Firstly, the maze is written into an array using the following symbols: "#" represents a wall, "*" represents a space, "S" is a start and "G" is a goal. Dimensions of the array are defined manually.

Determining positions of start and goal points is performed by the program, because their positions were changing in the stage of testing the program for different mazes.

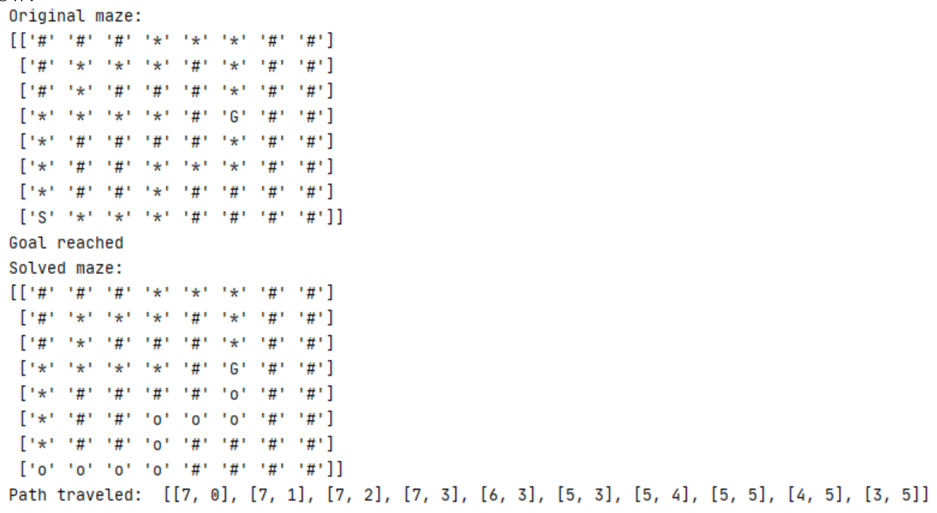
The attempt to find a valid path is started by defining a stack named **queue** in the form of list containing all elements of the path. Initially, it contains only an address of the start point. Then the **search()** function is called for the first time.

The **search()** function is recursive, which means that it is called for each valid node in the stack structure. First operation in this function, after getting information about the position in maze of the current node, is checking if the node is a goal. If yes, the function breaks, if not, the node is marked as visited using the "o" symbol. Then each of the four neighbours of the node is being check if they are valid.

The validation is performed in a separate function, called **valid()**. A valid node is a one which is not a wall or a visited node, and is within the dimensions of the maze. If the node is valid, a queue is appended by the node's address. Then the function search() calls itself with the valid node as an argument.

If none of the neighbouring nodes is valid, the last element from the stack is removed and the function search() calls itself with the one before last node as an argument - it takes a step back and tries to search for a different path coming from the previous node.

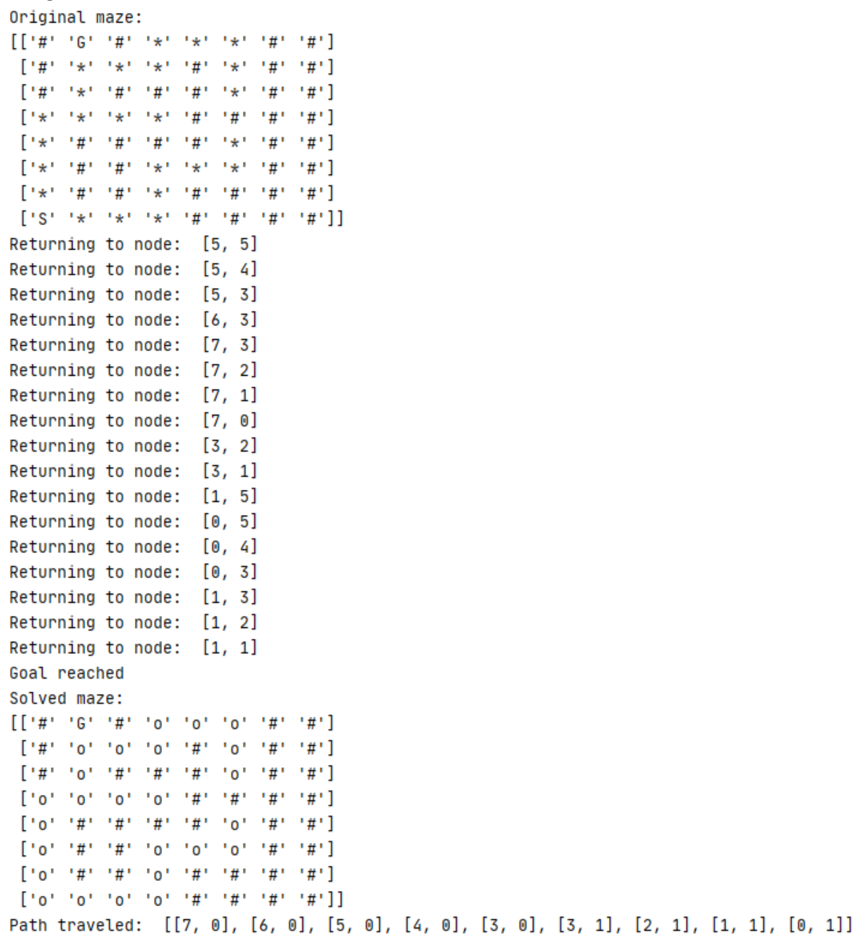
When the goal is reached the function recurrence breaks as it was mentioned above and program prints both mazes, the base one and the solved one on the console as well as the path that was taken. The console's output is depicted below.



```
Original maze:
[['#' '#' '#' '*' '*' '*' '#' '#']
 ['#' '*' '*' '*' '#' '*' '#' '#']
 ['#' '*' '#' '#' '#' '*' '#' '#']
 ['*' '*' '*' '*' '#' 'G' '#' '#']
 ['*' '#' '#' '#' '#' '*' '#' '#']
 ['*' '#' '#' '*' '*' '*' '#' '#']
 ['*' '#' '#' '*' '#' '#' '#' '#']
 ['S' '*' '*' '*' '#' '#' '#' '#']]
Goal reached
Solved maze:
[['#' '#' '#' '*' '*' '*' '#' '#']
 ['#' '*' '*' '*' '#' '*' '#' '#']
 ['#' '*' '#' '#' '#' '*' '#' '#']
 ['*' '*' '*' '*' '#' 'G' '#' '#']
 ['*' '#' '#' '#' '#' 'o' '#' '#']
 ['*' '#' '#' 'o' 'o' 'o' '#' '#']
 ['*' '#' '#' 'o' '#' '#' '#' '#']
 ['o' 'o' 'o' 'o' '#' '#' '#' '#']]
Path traveled:  [[7, 0], [7, 1], [7, 2], [7, 3], [6, 3], [5, 3], [5, 4], [5, 5], [4, 5], [3, 5]]
```

As one can notice, the path found by the algorithm is valid, but there is no guarantee that it is the shortest or the most efficient one.

The algorithm was tested for a number of different mazes. For each time it gave the correct results. Some of the tests are described below.



```
Original maze:
[['#' 'G' '#' '*' '*' '*' '#' '#']
 ['#' '*' '*' '*' '#' '*' '#' '#']
 ['#' '*' '#' '#' '#' '*' '#' '#']
 ['*' '*' '*' '*' '#' '#' '#' '#']
 ['*' '#' '#' '#' '#' '*' '#' '#']
 ['*' '#' '#' '*' '*' '*' '#' '#']
 ['*' '#' '#' '*' '#' '#' '#' '#']
 ['S' '*' '*' '*' '#' '#' '#' '#']]
Returning to node:  [5, 5]
Returning to node:  [5, 4]
Returning to node:  [5, 3]
Returning to node:  [6, 3]
Returning to node:  [7, 3]
Returning to node:  [7, 2]
Returning to node:  [7, 1]
Returning to node:  [7, 0]
Returning to node:  [3, 2]
Returning to node:  [3, 1]
Returning to node:  [1, 5]
Returning to node:  [0, 5]
Returning to node:  [0, 4]
Returning to node:  [0, 3]
Returning to node:  [1, 3]
Returning to node:  [1, 2]
Returning to node:  [1, 1]
Goal reached
Solved maze:
[['#' 'G' '#' 'o' 'o' 'o' '#' '#']
 ['#' 'o' 'o' 'o' '#' 'o' '#' '#']
 ['#' 'o' '#' '#' '#' 'o' '#' '#']
 ['o' 'o' 'o' 'o' '#' '#' '#' '#']
 ['o' '#' '#' '#' '#' 'o' '#' '#']
 ['o' '#' '#' 'o' 'o' 'o' '#' '#']
 ['o' '#' '#' 'o' '#' '#' '#' '#']
 ['o' 'o' 'o' 'o' '#' '#' '#' '#']]
Path traveled:  [[7, 0], [6, 0], [5, 0], [4, 0], [3, 0], [3, 1], [2, 1], [1, 1], [0, 1]]
```

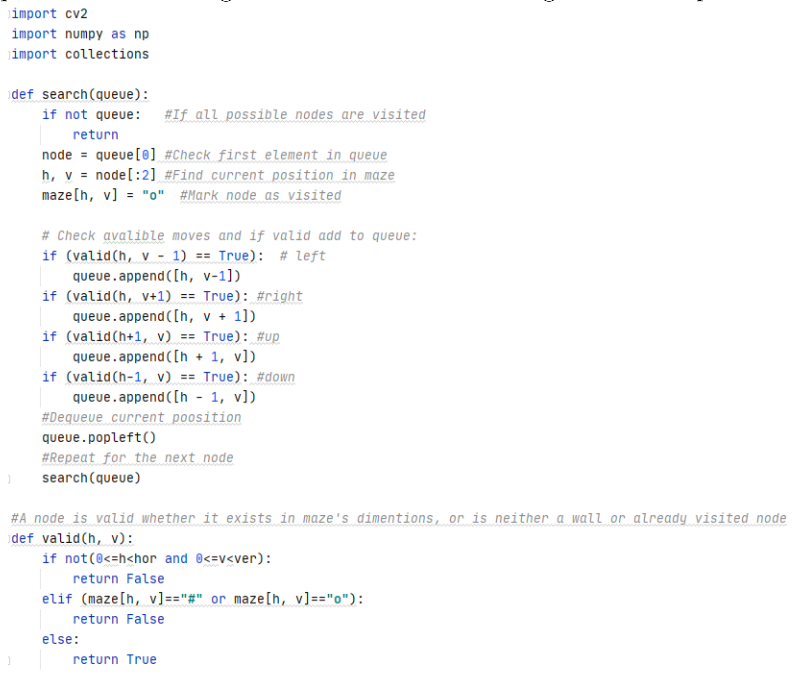
As the base algorithm firstly checks the node to the right which leads to finding the path in the first possible attempt, the goal was moved to the top of the maze and the previous path to the goal was left as a dead end.

As it is visible on the console output above, the goal was reached correctly. Although, the algorithm reached dead end twice and was returning to the nearest node. Paths for both returnings are displayed in the console window. It indicates that the algorithm is not efficient, as it passes the node at the point [1, 1], not knowing that it was just near the goal.

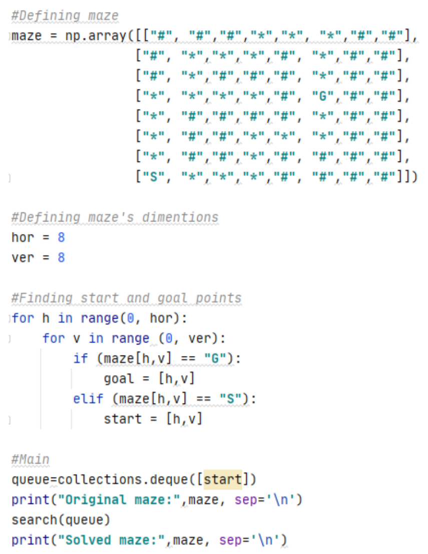
3 Breadth First Search

Breadth first search algorithm is believed to be more efficient than Depth First Search as it checks all neighbouring nodes before moving to the next one. It also checks all nodes existing in the maze before breaking the loop, which allows to distinguish the shortest path when more than one is discovered.

The input maze will not be repeated as it is the copy of the one used in Depth First Search algorithm. The code of the algorithm is depicted below.



```
import cv2
import numpy as np
import collections

def search(queue):
    if not queue:   #If all possible nodes are visited
        return
    node = queue[0] #Check first element in queue
    h, v = node[:2] #Find current position in maze
    maze[h, v] = "o"  #Mark node as visited

    # Check avalible moves and if valid add to queue:
    if (valid(h, v - 1) == True):  # left
        queue.append([h, v-1])
    if (valid(h, v+1) == True): #right
        queue.append([h, v + 1])
    if (valid(h+1, v) == True): #up
        queue.append([h + 1, v])
    if (valid(h-1, v) == True): #down
        queue.append([h - 1, v])
    #Dequeue current poosition
    queue.popleft()
    #Repeat for the next node
    search(queue)

#A node is valid whether it exists in maze's dimentions, or is neither a wall or already visited node
def valid(h, v):
    if not(0<=h<hor and 0<=v<ver):
        return False
    elif (maze[h, v]=="#" or maze[h, v]=="o"):
        return False
    else:
        return True
```

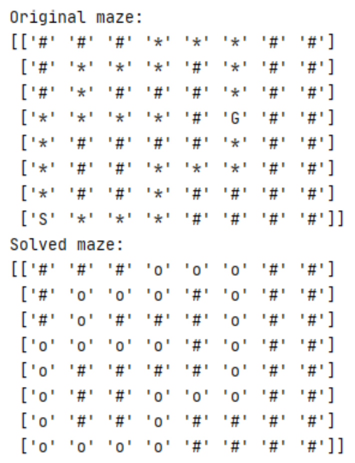


```
#Defining maze
maze = np.array([["#", "#","#","*","*", "*","#","#"],
                ["#", "*","*","*","#", "*","#","#"],
                ["#", "*","#","#","#", "*","#","#"],
                ["*", "*","*","*","#", "G","#","#"],
                ["*", "#","#","#","#", "*","#","#"],
                ["*", "#","#","*","*", "*","#","#"],
                ["*", "#","#","*","#", "#","#","#"],
                ["S", "*","*","*","#", "#","#","#"]])

#Defining maze's dimentions
hor = 8
ver = 8

#Finding start and goal points
for h in range(0, hor):
    for v in range (0, ver):
        if (maze[h,v] == "G"):
            goal = [h,v]
        elif (maze[h,v] == "S"):
            start = [h,v]

#Main
queue=collections.deque([start])
print("Original maze:",maze, sep='\n')
search(queue)
print("Solved maze:",maze, sep='\n')
```

The main function of the code is similar to the one performed in Depth First Search. The maze and its dimensions are defined and the program finds positions of the start and goal points. However, the data structure used in this algorithm is a queue which uses the property of ”fist in - first out”. As it is defined and filled with the address of the starting point, the **search()** function is called for the first time.

The function **search()** is designed in the way, that it breaks whenever the queue has no more elements to check. It always considers the first node from the queue, which is the one that has been the longest in the structure. It checks all the neighbouring nodes and if any is valid it appends the queue with the address of this node. The validation requirements are identical as in the Depth First Search and once again are determined inside a separate function called **valid()**. At the end the considered node is removed from the queue and the function calls itself to check the next node.

When the goal is reached, console prints the base maze as well as the solved one. The output of the console is depicted below.



```
Original maze:
[['#' '#' '#' '*' '*' '*' '#' '#']
 ['#' '*' '*' '*' '#' '*' '#' '#']
 ['#' '*' '#' '#' '#' '*' '#' '#']
 ['*' '*' '*' '*' '#' 'G' '#' '#']
 ['*' '#' '#' '#' '#' '*' '#' '#']
 ['*' '#' '#' '*' '*' '*' '#' '#']
 ['*' '#' '#' '*' '#' '#' '#' '#']
 ['S' '*' '*' '*' '#' '#' '#' '#']]
Solved maze:
[['#' '#' '#' 'o' 'o' 'o' '#' '#']
 ['#' 'o' 'o' 'o' '#' 'o' '#' '#']
 ['#' 'o' '#' '#' '#' 'o' '#' '#']
 ['o' 'o' 'o' 'o' '#' 'o' '#' '#']
 ['o' '#' '#' '#' '#' 'o' '#' '#']
 ['o' '#' '#' 'o' 'o' 'o' '#' '#']
 ['o' '#' '#' 'o' '#' '#' '#' '#']
 ['o' 'o' 'o' 'o' '#' '#' '#' '#']]
```

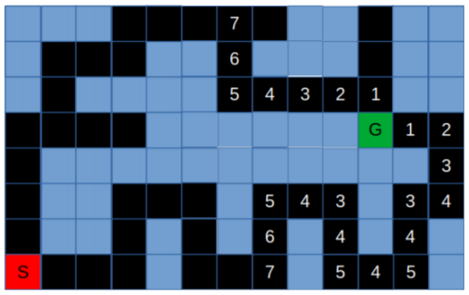
The output indicates, that there was two paths that reached the goal point. The algorithm was considering all possible nodes, including a dead end at [3, 3].

4 A* Algorithm

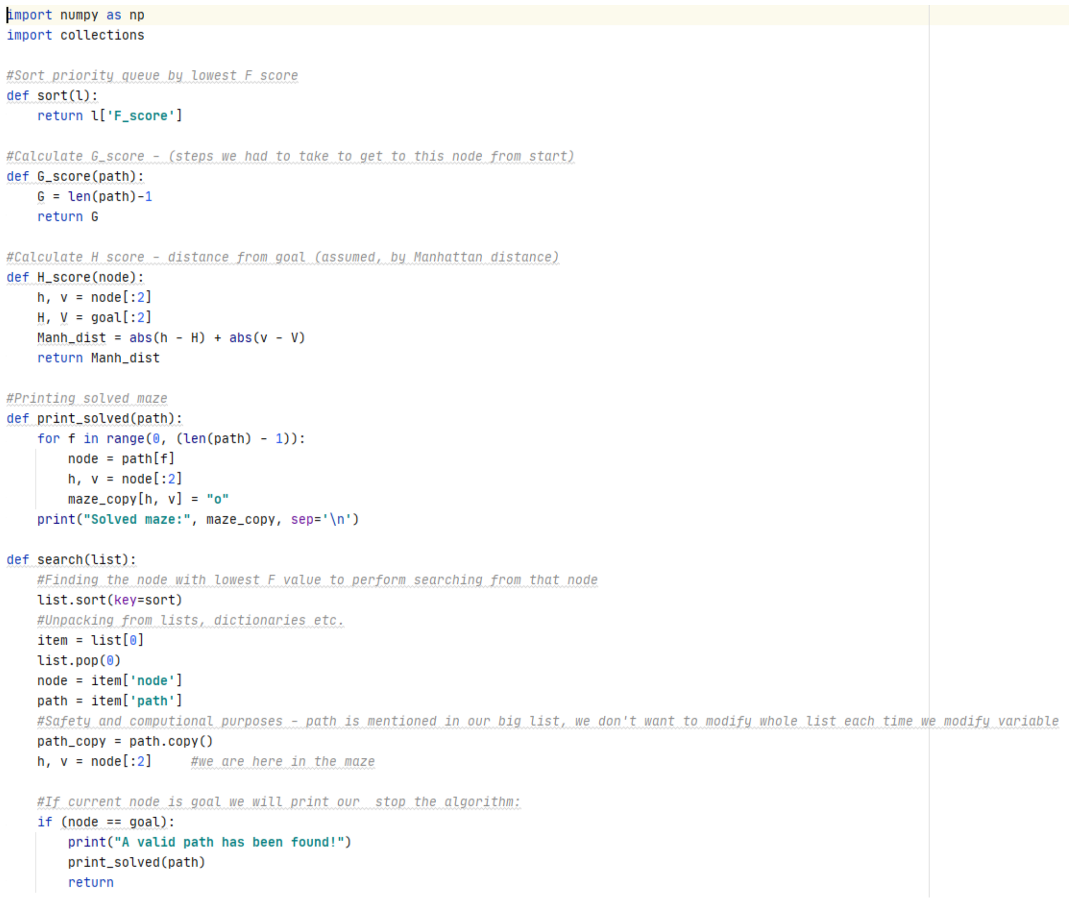
The A-star algorithm is the most complex one, as it performs not only the path finding but also the efficiency analysis of the path. It searches for the path with the lowest F score, which is defined as F(x) = G(x) + H(x), where G score is the number of steps that were already taken to reach a current node and H score is assumed number of steps that are needed to be taken to reach the goal from a current node. This way, it can find the most efficient path without considering all available nodes. H score is predicted in my implementation by calculating Manhattan distance between current and goal nodes.

In order to meet the requirement of storing a greater number of information about each node than it was needed before, a data type of dictionary was used in the program. It was structured to resemble a priority queue with and F score as the priority number.

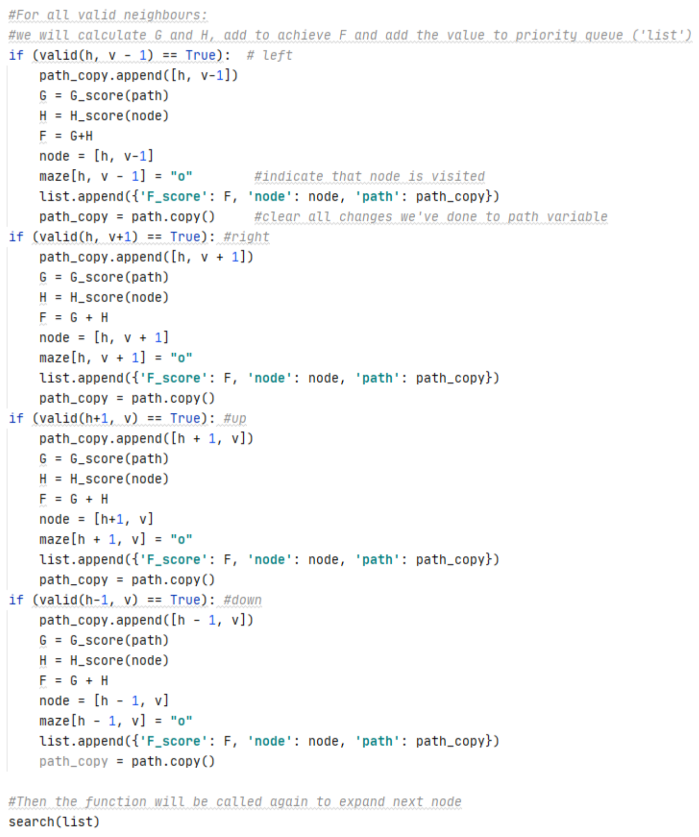
The maze used in this algorithm was different from the previous one and is depicted below in a graphical form.



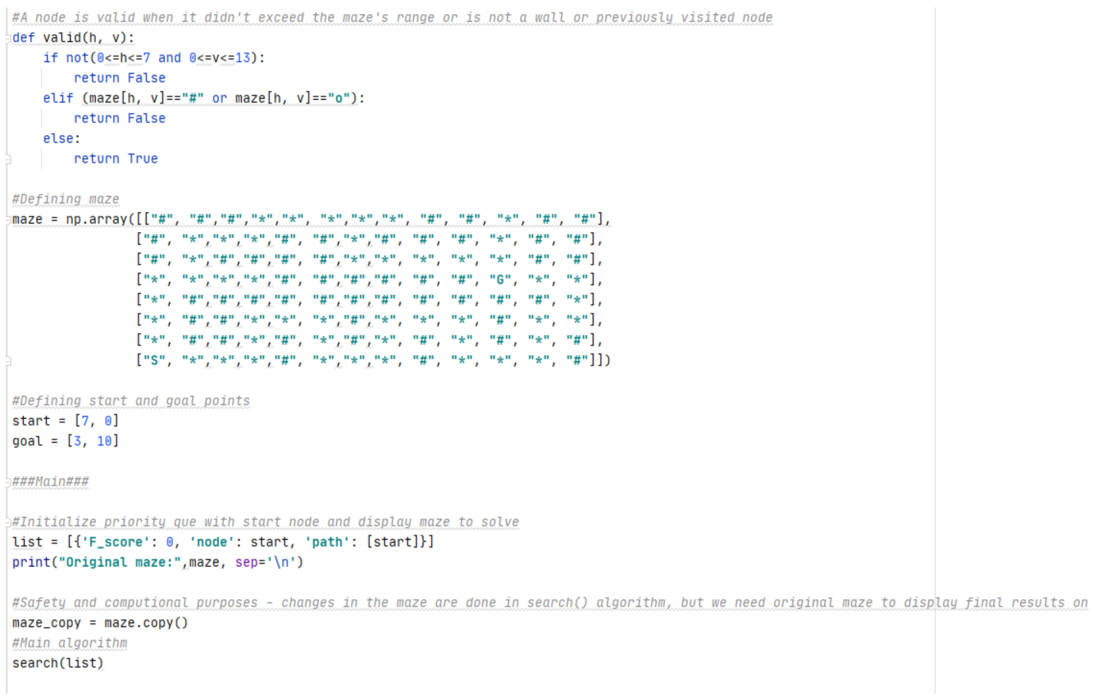
The code written to perform the maze solving is depicted below.



```python
import numpy as np
import collections

#Sort priority queue by lowest F score
def sort(l):
    return l['F_score']

#Calculate G_score - (steps we had to take to get to this node from start)
def G_score(path):
    G = len(path)-1
    return G

#Calculate H score - distance from goal (assumed, by Manhattan distance)
def H_score(node):
    h, v = node[:2]
    H, V = goal[:2]
    Manh_dist = abs(h - H) + abs(v - V)
    return Manh_dist

#Printing solved maze
def print_solved(path):
    for f in range(0, (len(path) - 1)):
        node = path[f]
        h, v = node[:2]
        maze_copy[h, v] = "o"
    print("Solved maze:", maze_copy, sep='\n')

def search(list):
    #Finding the node with lowest F value to perform searching from that node
    list.sort(key=sort)
    #Unpacking from lists, dictionaries etc.
    item = list[0]
    list.pop(0)
    node = item['node']
    path = item['path']
    #Safety and computional purposes - path is mentioned in our big list, we don't want to modify whole list each time we modify variable
    path_copy = path.copy()
    h, v = node[:2]     #we are here in the maze

    #If current node is goal we will print our  stop the algorithm:
    if (node == goal):
        print("A valid path has been found!")
        print_solved(path)
        return
```



```
#For all valid neighbours:
#we will calculate G and H, add to achieve F and add the value to priority queue ('list')
if (valid(h, v - 1) == True):  # left
    path_copy.append([h, v-1])
    G = G_score(path)
    H = H_score(node)
    F = G+H
    node = [h, v-1]
    maze[h, v - 1] = "o"        #indicate that node is visited
    list.append({'F_score': F, 'node': node, 'path': path_copy})
    path_copy = path.copy()     #clear all changes we've done to path variable
if (valid(h, v+1) == True): #right
    path_copy.append([h, v + 1])
    G = G_score(path)
    H = H_score(node)
    F = G + H
    node = [h, v + 1]
    maze[h, v + 1] = "o"
    list.append({'F_score': F, 'node': node, 'path': path_copy})
    path_copy = path.copy()
if (valid(h+1, v) == True): #up
    path_copy.append([h + 1, v])
    G = G_score(path)
    H = H_score(node)
    F = G + H
    node = [h+1, v]
    maze[h + 1, v] = "o"
    list.append({'F_score': F, 'node': node, 'path': path_copy})
    path_copy = path.copy()
if (valid(h-1, v) == True): #down
    path_copy.append([h - 1, v])
    G = G_score(path)
    H = H_score(node)
    F = G + H
    node = [h - 1, v]
    maze[h - 1, v] = "o"
    list.append({'F_score': F, 'node': node, 'path': path_copy})
    path_copy = path.copy()

#Then the function will be called again to expand next node
search(list)
```



```python
#A node is valid when it didn't exceed the maze's range or is not a wall or previously visited node
def valid(h, v):
    if not(0<=h<=7 and 0<=v<=13):
        return False
    elif (maze[h, v]=="#" or maze[h, v]=="o"):
        return False
    else:
        return True

#Defining maze
maze = np.array([["#", "#","#","*","*", "*","*","*", "#", "#", "*", "#", "#"],
                ["#", "*","*","*","#", "#","*","#", "#", "#", "*", "#", "#"],
                ["#", "*","#","#","#", "#","*","*", "*", "*", "*", "#", "#"],
                ["*", "*","*","*","#", "#","#","#", "#", "#", "G", "*", "*"],
                ["*", "#","#","#","#", "#","#","#", "#", "#", "#", "#", "*"],
                ["*", "#","#","*","*", "*","#","*", "*", "*", "#", "*", "*"],
                ["*", "#","#","*","#", "*","#","*", "#", "*", "#", "*", "#"],
                ["S", "*","*","*","#", "*","*","*", "#", "*", "*", "*", "#"]])

#Defining start and goal points
start = [7, 0]
goal = [3, 10]

###Main###

#Initialize priority que with start node and display maze to solve
list = [{'F_score': 0, 'node': start, 'path': [start]}]
print("Original maze:",maze, sep='\n')

#Safety and computional purposes - changes in the maze are done in search() algorithm, but we need original maze to display final results on
maze_copy = maze.copy()
#Main algorithm
search(list)
```

As previously, before the main function, the maze is firstly stored in the form of array and the start and goal points are defined.

In the main section of the program a priority queue in the form of a dictionary is initialized with the first element being a start node. A dictionary element has three keys: F score of the node, its address and the path, which is a list of all steps that were taken to reach the node. The base maze is then printed and its elements are copied to a new variable maze_copy, which will be used to display the results on the end of the program. Then the function **search()** is called for the first time.

As the **list** variable should resemble a priority queue, its values are being sorted each time at the beginning of the **search()** function. It is done inside a separate **sort()** function which returns a list sorted by the F score as a keyword. As the algorithm should always check first the path with lowest F score, this steps assures that the desired node is always the first element of the list.

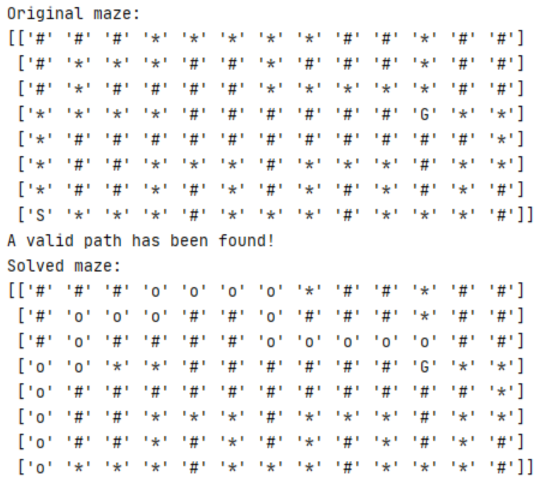
Then all useful informations are unpacked from the dictionary element, i.e. node address and the path that was already taken. The element is then removed from the priority queue and its path is copied to a **path_copy** variable as it is not desired to change it when checking for validation of the neighbouring nodes.

In this moment the function is able to check whether the current node is a goal point. If yes, it breaks the loop and prints the solved maze using **print_solved** function. If not, the next steps are performed, which are checking if any of nearest neighbouring nodes is valid.

The validation requirements are identical to the ones in Depth First Search or Breadth first Search. If the node is valid, its F score is calculated (thus G and

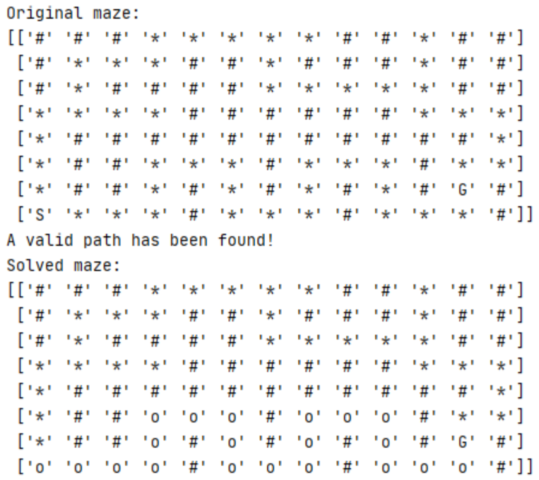
H score are calculated), the path is appended and the **list** variable is appended with the new dictionary element. When all possible nodes are checked, the function calls itself recursively.

According to the fact, that the algorithm performs its operations only for the path with lowest F value and breaks whenever the goal is reached for the first time, the path that was found should always be the shortest one. The output of the algorithm is depicted below and indicates that although there was two possible ways to reach the goal, program used only the upper one with least steps needed to be taken.



```
Original maze:
[['#' '#' '#' '*' '*' '*' '*' '*' '#' '#' '*' '#' '#']
 ['#' '*' '*' '*' '#' '#' '*' '#' '#' '#' '*' '#' '#']
 ['#' '*' '#' '#' '#' '#' '*' '*' '*' '*' '*' '#' '#']
 ['*' '*' '*' '*' '#' '#' '#' '#' '#' '#' 'G' '*' '*']
 ['*' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '*']
 ['*' '#' '#' '*' '*' '*' '#' '*' '*' '*' '#' '*' '*']
 ['*' '#' '#' '*' '#' '*' '#' '*' '#' '*' '#' '*' '#']
 ['S' '*' '*' '*' '#' '*' '*' '*' '#' '*' '*' '*' '#']]
A valid path has been found!
Solved maze:
[['#' '#' '#' 'o' 'o' 'o' 'o' '*' '#' '#' '*' '#' '#']
 ['#' 'o' 'o' 'o' '#' '#' 'o' '#' '#' '#' '*' '#' '#']
 ['#' 'o' '#' '#' '#' '#' 'o' 'o' 'o' 'o' 'o' '#' '#']
 ['o' 'o' '*' '*' '#' '#' '#' '#' '#' '#' 'G' '*' '*']
 ['o' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '*']
 ['o' '#' '#' '*' '*' '*' '#' '*' '*' '*' '#' '*' '*']
 ['o' '#' '#' '*' '#' '*' '#' '*' '#' '*' '#' '*' '#']
 ['o' '*' '*' '*' '#' '*' '*' '*' '#' '*' '*' '*' '#']]
```

When the goal point was moved in order to make the upper path longer, the algorithm correctly indicated the lower one as shorted. Output of the console for this test is depicted below.



```
Original maze:
[['#' '#' '#' '*' '*' '*' '*' '*' '#' '#' '*' '#' '#']
 ['#' '*' '*' '*' '#' '#' '*' '#' '#' '#' '*' '#' '#']
 ['#' '*' '#' '#' '#' '#' '*' '*' '*' '*' '*' '#' '#']
 ['*' '*' '*' '*' '#' '#' '#' '#' '#' '#' '*' '*' '*']
 ['*' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '*']
 ['*' '#' '#' '*' '*' '*' '#' '*' '*' '*' '#' '*' '*']
 ['*' '#' '#' '*' '#' '*' '#' '*' '#' '*' '#' 'G' '#']
 ['S' '*' '*' '*' '#' '*' '*' '*' '#' '*' '*' '*' '#']]
A valid path has been found!
Solved maze:
[['#' '#' '#' '*' '*' '*' '*' '*' '#' '#' '*' '#' '#']
 ['#' '*' '*' '*' '#' '#' '*' '#' '#' '#' '*' '#' '#']
 ['#' '*' '#' '#' '#' '#' '*' '*' '*' '*' '*' '#' '#']
 ['*' '*' '*' '*' '#' '#' '#' '#' '#' '#' '*' '*' '*']
 ['*' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '#' '*']
 ['*' '#' '#' 'o' 'o' 'o' '#' 'o' 'o' 'o' '#' '*' '*']
 ['*' '#' '#' 'o' '#' 'o' '#' 'o' '#' 'o' '#' 'G' '#']
 ['o' 'o' 'o' 'o' '#' 'o' 'o' 'o' '#' 'o' 'o' 'o' '#']]
```

All three algorithms designed for this laboratory turned out to be valid and satisfy a few important sets of tests.