Artificial Intelligence and Computer Vision

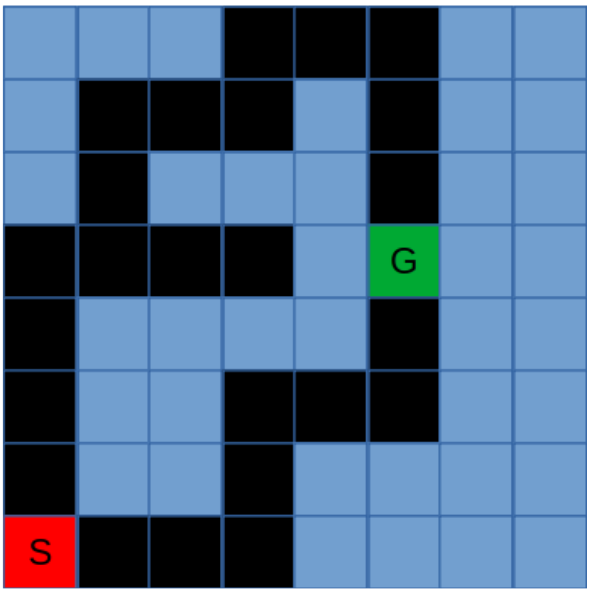
Laboratory 6

Report Sheet

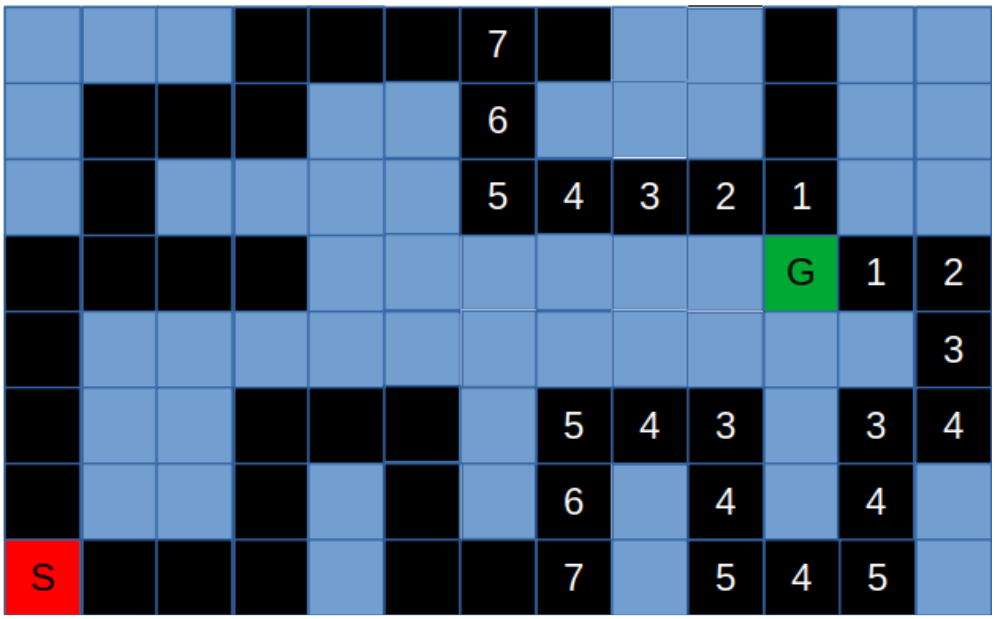
Jakub Bujak 249120

Task description:

1. **Implement Depth First Search or Breadth First Search Algorithm for given maze:**



2. **Implement A* algorithm for given maze. The numbers are showing the Manhattan distance from the goal.**



Report:

Firstly, we implement the libraries, which will help us develop the code for *Breadth First Search* algorithm. Here, we'll use Numpy, for array generating, and Collections, for appending and the queue element.

```
import numpy as np
import collections
```

The first function we define in the code is the path finding algorithm – here named *pathSearch()*. The algorithm works as such - it queues all possible moves, and then checks them one by one, sorting first in the queue those that deliver the shortest possible path to the goal target. After checking the two neighbouring positions, it marks the previous position it was in, with a "+" sign. Step by step it reaches the final destination. After having discovered the path, the algorithm dequeues the steps, to free the memory.

```
def pathSearch(queue):
    if not queue:
        return
    step = queue[0]
    x, y = step[:2]
    maze[x, y] = "+"

    # Checking move availability
    if(moveRules(x, y - 1) == True): # Go down
        queue.append([x, y - 1])

    if (moveRules(x, y + 1) == True):  # Go up
        queue.append([x, y + 1])

    if (moveRules(x + 1, y) == True):  # Go right
        queue.append([x + 1, y])

    if (moveRules(x - 1, y) == True):  # Go left
        queue.append([x - 1, y])
    # Reset queue
    queue.popleft()
    # Repeat steps
    pathSearch(queue)
```

The next function we define in the code is the moveRules() function. The defined piece of code behaves kind of like human eyes for the pathSearch() algorith - it helps the code understand where are the walls of the maze, and where it can actually move inside it.

```
def moveRules(x, y):
    if not (0 <= x < horizontal and 0 <= y < vertical):
        return False
    elif (maze[x, y] == "#" or maze[x,y] == "+"):
        return False
    else:
        return True
```

Next step is to define our maze. Finally, Numpy comes in handy here. We define the maze as an array 8x8.

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

# Defining dimensions
horizontal = 8
vertical = 8
```

In the last part we specify the points of start and our goal target for the code, so that it understands its position in the maze, and where to go. A simple double for loops is used in here.

```
# Find desired spots
for x in range(0, horizontal):
    for y in range(0, vertical):
        if(maze[x, y] == "G"):
            goal = [x, y]
        elif(maze[x, y] == "S"):
            start = [x, y]
```

Last, but not least, we run our path searching script and print the mazes – the original one, and with the path found respectively.

```
#Main
queue = collections.deque([start])
print("Original maze:", maze, sep='\n')
pathSearch(queue)
print("Solution to the maze: ", maze, sep='\n')
```

The results of the code:

```
Original maze:
[['#' '#' '#' '*' '*' '*' '#' '#']
 ['#' '*' '*' '*' '#' '*' '#' '#']
 ['#' '*' '#' '#' '#' '*' '#' '#']
 ['*' '*' '*' '*' '#' 'G' '#' '#']
 ['*' '#' '#' '#' '#' '*' '#' '#']
 ['*' '#' '#' '*' '*' '*' '#' '#']
 ['*' '#' '#' '*' '#' '#' '#' '#']
 ['S' '*' '*' '*' '#' '#' '#' '#']]
Solution to the maze:
[['#' '#' '#' '+' '+' '+' '#' '#']
 ['#' '+' '+' '+' '#' '+' '#' '#']
 ['#' '+' '#' '#' '#' '+' '#' '#']
 ['+' '+' '+' '+' '#' '+' '#' '#']
 ['+' '#' '#' '#' '#' '+' '#' '#']
 ['+' '#' '#' '+' '+' '+' '#' '#']
 ['+' '#' '#' '+' '#' '#' '#' '#']
 ['+' '+' '+' '+' '#' '#' '#' '#']]

Process finished with exit code 0
```

The results show us, that the algorithm has found two paths that deliver us the desired destination.