A logo with a blue circle and red stripes

Description automatically generated

**T.C.**

**MARMARA UNIVERSITY**

**FACULTY of ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

**CSE 4082– Project 2**

Group Members

150119052 Sadık AKGEDİK

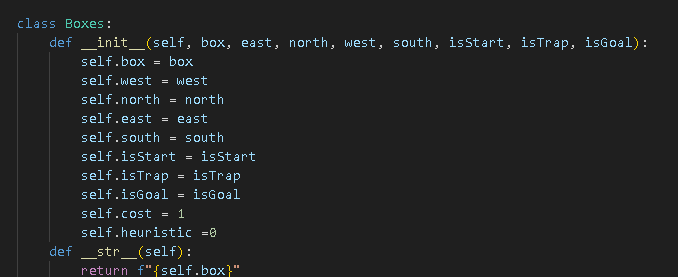
150120518 Evren KOYMAT

150118008 Osman Tunç POYRAZ

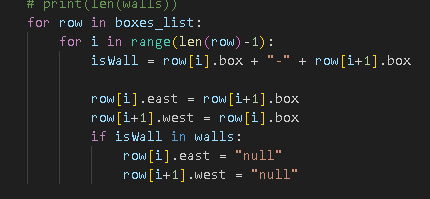
**1. Problem description**

The project involves a Python script designed to solve a 2D maze using different search algorithms. The core components of the design include the Boxes class, which represents individual cells in the maze filled with traps, and the implementation of various search algorithms in order to reach a goal state.

**2.Classes**  
Boxes Class:



In the code we hold the boxes as box class. Box object will held all the required data fields that it needs. When we defined the box for the maze we will assign all the values. We will take our input.txt and read the size of the maze, walls, trap nodes and goal nodes. After reading the input we will assign them according to their respected values. We can see at the below how walls are defined.



If there is a wall between nodes the neighboring the node will be node, otherwise it will be declared as its neighbor.

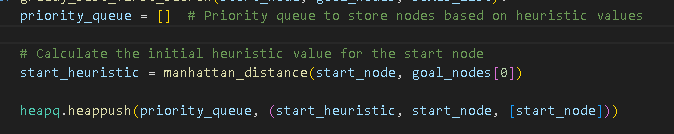
**3. Methods and Functions**

In the project there will be 6 different algorithms that we need to solve maze with.

But first of all I will talk about the General methods that I use for this project that are the same for the every algoritihms.

**1-> Priority Queue:**

We need some place to store our frontier, current node, its neighbors and the path of the current node. We will use priority queue for that.

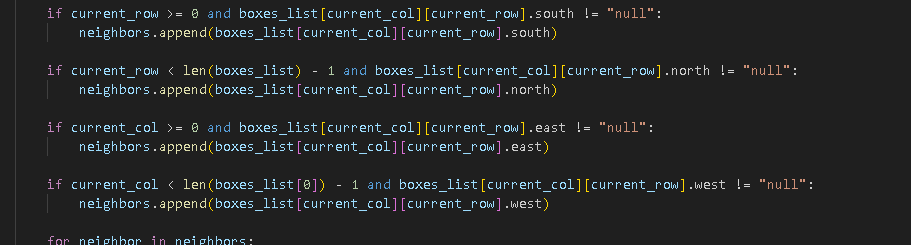


This is the initializion of our queue is above. The way we use our queue is below.



We push the queue the box and its value of how likeable it is, its neighbors and its path.

**2->Finding the Neighbors of Current Node**

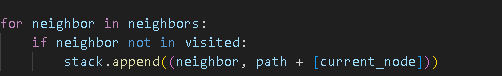


We use the code above to find the neighbors. We check the restrains by size then we check if is there a wall between the boxes. Then if its all true we append the neighbor of the box itself.

**Search algorihms:**

**1-> 1. Depth-First Search (DFS)**

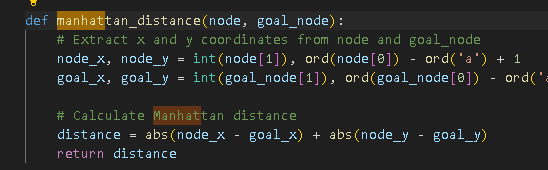
For the Depth First search this is code :



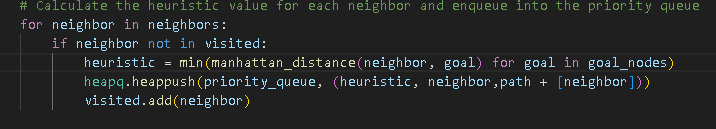
In this code if there is neighbor it is not visited we will add to stack and we update the path of the node that we added.

**2->** **Greedy Best-First Search.**

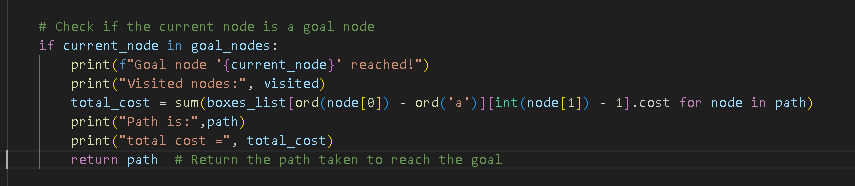
In this algorithm we use heuristic function to realize which nodes are closer to the Goal then the others. The heuristic function is the only thing that we use to make this decision. The Function that we are using is Manhattan distance. Which is:



Then we use queue to first push the neighbors into queue then according to heuristic value we choose our next box.

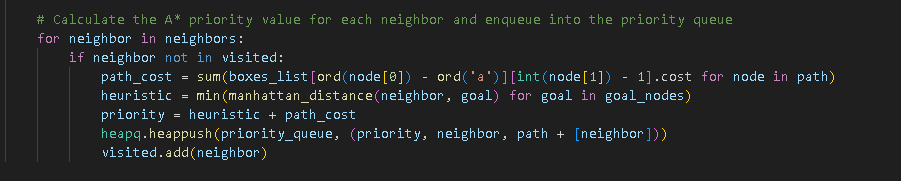


When we find the goal node we will print the path and its total cost.



**3->A\* Search**

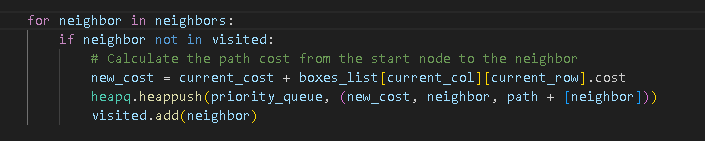
This is almost same as the greedy search algorithm but the likeability of the node is not only measured by the heuristic value its also measured by its cost to reach that specific node. We add them up together then use that value to decide our next node.



To get the path cost of the node which we store them up in our path array we use path\_cost.

**4-> Uniform Cost Search**

This search method only cares about how much box is costing you to move. The rest is mostly are the same. We still use priority queue to hold our values and our deciding value this time is the cost.



**5-> Iterative-Deepening Search**

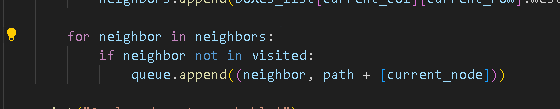
This search method takes a maximum depth limit as input as it increases the depth limit with each iteration until the goal is found. The

A screen shot of a computer program

Description automatically generated

**6->Breadth First Search Algorithm**

Its mostly same as the DFS Algorithm. In DFS we use stack to hold the values. In BFS we use queue to hold the values.



**4. User Interface**

menu(): Displays a menu for the user to choose a search algorithm.

main(): Implements the main execution logic, allowing the user to choose and run different search algorithms.

**5. Execution Flow**

* Read input data from a file.
* Initialize the grid of boxes.
* Adjust the connectivity of boxes based on walls information.
* Allow the user to choose a search algorithm from the menu.
* Execute the selected search algorithm.
* Display the results, including the path, total cost, and visited nodes.

**6. The Output**

The input will be in the form of the figure given below.

A screenshot of a computer

Description automatically generated

When the program is ran, it will generate the maze from the “input\_data.txt” file. After that it will give the user a menu to choose the desired searching algorithm.

A screenshot of a computer program

Description automatically generated

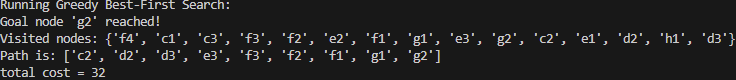
For all the options the outputs are following:

1-Depth-First Search

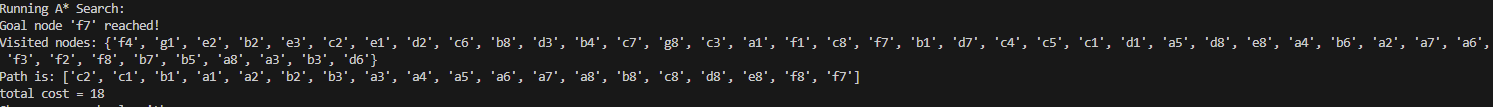
A group of white text on a black background

Description automatically generated

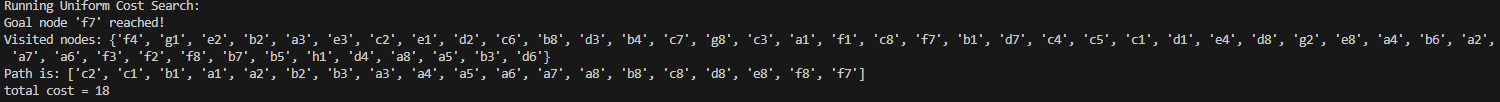
2-Greedy Best-First Search



3-A\* search



4-Uniform Cost Search



5-Iterative Deepening Search

A screenshot of a computer

Description automatically generated

6-Breadth first search

A black screen with white text

Description automatically generated