Artificial Intelligence

Final Report of First Practical Assignment

Stylianos Tsagkarakis up201911231
Vasileios Konstantaras up201911213

1. Specification of the work

BoxWorld 2 is a solitary type game in which the player needs to move on the board with the the **ARROW** keys to reach the exit, in order to advance to the next level. The game has several obstacles that the player needs to overcome to finish each level.

- Boxes that can be pushed
- Holes that can be filled with boxes in order to move over them
- Starts with a specific number of lives, and if the player gets stuck, can press the **R** key to reload the same level with the expense of one life
- Save the game by pressing **B** key
- Load previous save by pressing **L** key

We need to:

1. Compare the different research methods based on the quality of the obtained solution, the total number of operations occurred and the time taken to obtain the solution.
2. The program must have some kind of visualization whether text-based or graphical so the user can see the evolution of the board.
3. The program must be able to solve the game by its own using the method and the board configuration provided by the user.
4. Optionally a game mode that the user plays and asks the program for "tips" could be implemented.

2. Research (articles, web pages, source codes)

- This project exists as a Github repository
- 8 Reasons to Use Python for AI
- Create Your Own Reinforcement Learning Environment
- Pathfinding in Strategy Games and Maze Solving

3. Formulating the problem as a research problem

- **Initial state:** position of the user on the table when the level begins
- **Final state:** is the exit position.

- The **different states** are represented by the different instances of the world and below is visible the operators.

- The **preconditions** are:
  1. where the player wants to be an empty space, meaning no boxes, holes or wall, then the space becomes the player's position and the previous one becomes empty.
  2. To move a box, it is required the next place after the box, towards the direction the user wants to move it, to be empty or a hole. If it is a hole, when the box falls in to the hole it becomes empty space. Otherwise the box covers the next spot, the user covers the spot previously owned by the box and the place that the user was before, becomes empty.

This is the state representation written in pseudocode:

```
Class World():
        map             #This contains the map of the level
        X               #The width of the map
        Y               #The length of the map
        finish          #The finish coordinates of the level
        userposition    #The position of the user in the map
        lives           #Remaining lives of the user
        path            #The path the leads to the finish
```

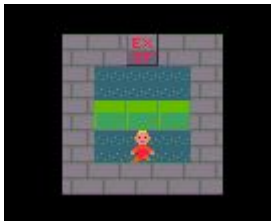
Every time that there is a successful movement a new instance of the world will be created with different values at the operators.

4. Approach

Our approach is based around having a very clean and structured code. That's why we have created different files that each of them contain functions that complete specific operations.
Firstly, to demonstrate the different levels of the game we create matrices that depict on them the different objects of the level, like walls, boxes, holes as well as the position of the user, this matrix is called map.

.

That's a screenshot from the actual game the program of level 3



The level is represented as a matrix

W = wall
U = user
F = end
B = box
I = Ice box
H = hole

In the file **Functions.py** is located one of the most important functions and is called *move()*. This function is responsible to move the user into the matrix at the direction chosen. Here all the necessary operations happen, while calling other functions that do checks, to ensure it is a possible movement
(if there is a wall up and the user choose to go up it will not allow the user to move).
To move the user he needs to interact with the position he choose to
move depending of what case of interaction.
This is explained in the next slide.

```
def switch_case (first, second):
    switcher = {
        "user"   : 'U' ,
        "box"    : 'B' ,
        "ice"    : 'I' ,
        "hole"   : 'H' ,
        "wall"   : 'W' ,
        "finish" : 'F' ,
        "space"  : '-'
    }
    first = switcher.get(first, "error")
    second = switcher.get(second, "error")
    action_func = first+"_and_"+second
    return action_func
```

First and second are supposed to be the items that interact during the game. After getting the function name, it will be called like that:

We wanted to eliminate if cases as much as possible, to avoid confusion and repetitiveness. Using python's string concatenation ability we will make a switch case like that:

```
function = switch_case(arg1, arg2)
function()

# list of possible action_funcs:

def user_and_space (world, direction):
def user_and_box (world, direction):
def user_and_finish (world, direction):
def user_and_ice (world, direction):
def user_and_wall (world, direction):
def user_and_hole (world, direction):
def box_and_wall (world, direction):
def box_and_hole (world, direction):
def ice_and_wall (world, direction):
def ice_and_hole (world, direction):
```

So when we want to use a search algorithm to to solve a specific level of the game, we just call ***move(direction)*** with the chosen direction and it returns a result whether the movement is successful, failed or if it reached the end of the level. Then inside the search algorithm we just update to the new position. The evaluation of the algorithms is based to the shortest path towards the end of the level.

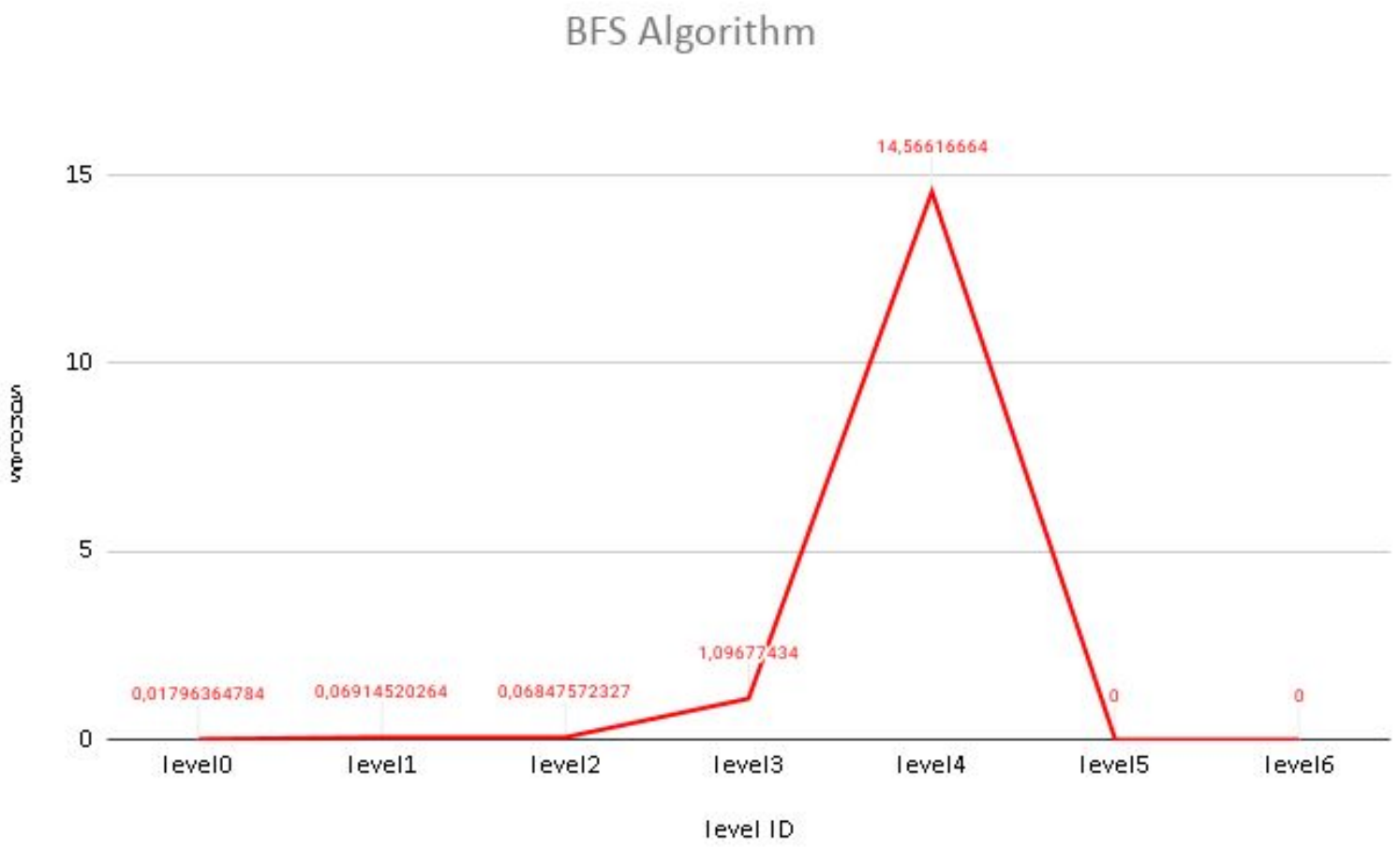
5. Algorithms Implemented

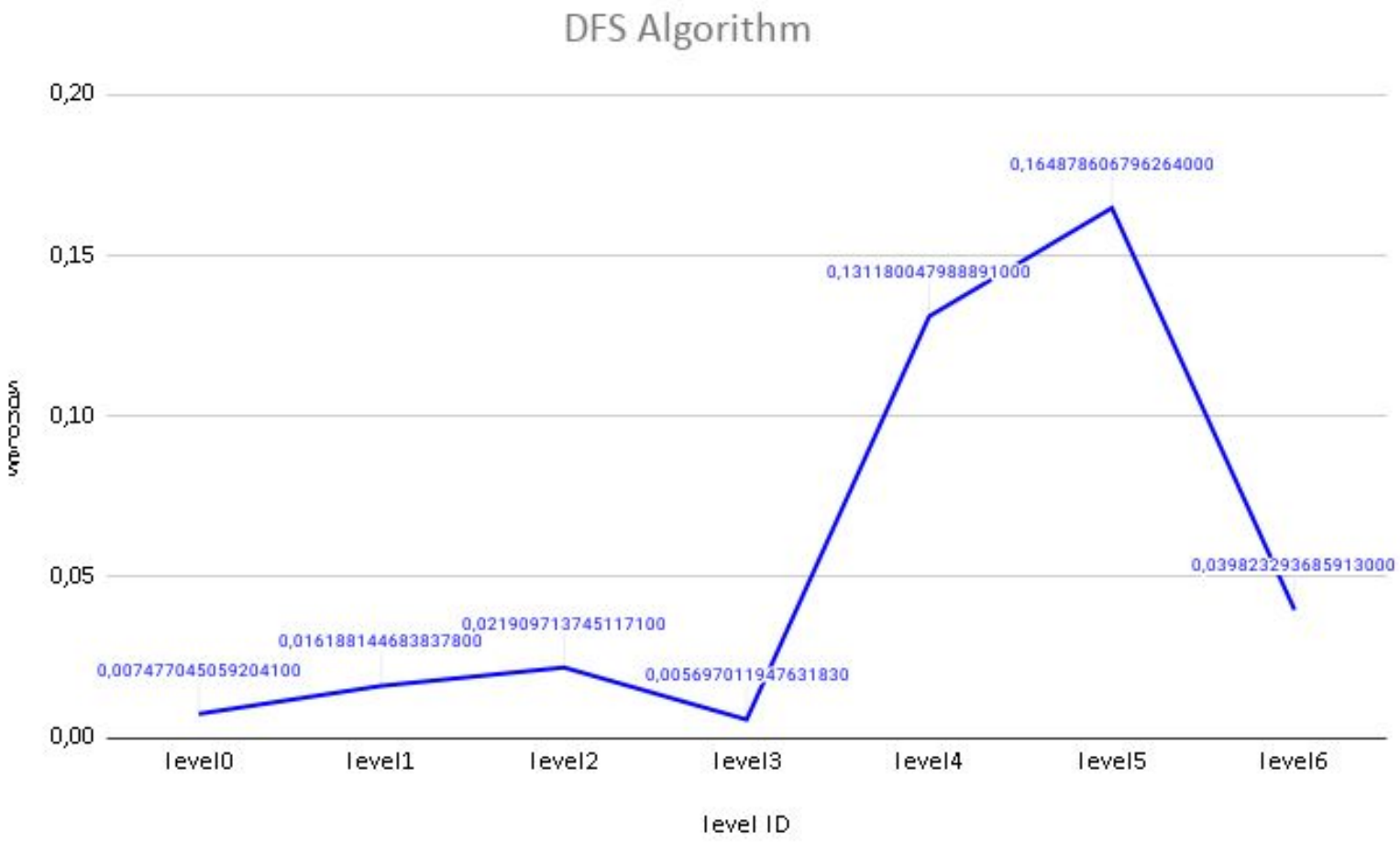
The algorithms we managed to implement to solve the game are:

1. **Breadth-First Search:**
   Uses a queue to keep track of the path, which is filled with instances of the world with the different movements applied.

2. **Depth-First Search**
   Uses a stack to keep track of the path, which is filled with instances of the world with the different movements applied.

6. Results


BFS Algorithm
15
10
5
0
14,56616664
1,09677434
0,01796364784
0,06914520264
0,06847572327
0
0
level0
level1
level2
level3
level4
level5
level6
level ID


DFS Algorithm
0,20
0,15
0,10
0,05
0,00
milliseconds
0,007477045059204100
0,016188144683837800
0,021909713745117100
0,005697011947631830
0,131180047988891000
0,164878606796264000
0,039823293685913000
level0
level1
level2
level3
level4
level5
level6
level ID

7. Conclusion

- We notice that DFS Algorithm is faster the the BFS Algorithm.
- Total time of DFS: **0,38 seconds** for 6 levels.
- Total time of BFS: **15,81 seconds** for 4 levels (couldn't find a path at the last two levels).

If A* was implemented it would (hypothetically) be even faster and more efficient than all of the other algorithms.