Name: Haguar Tarek Dessouky ID: 6878

Name: Mariem Mostafa Mahmoud ID: 6873

Name: Nada Taher Elwazane ID: 6876

Name: Manar Amgad ID: 7113

Artificial Intelligence

Project 1

8-Puzzle Boardgame

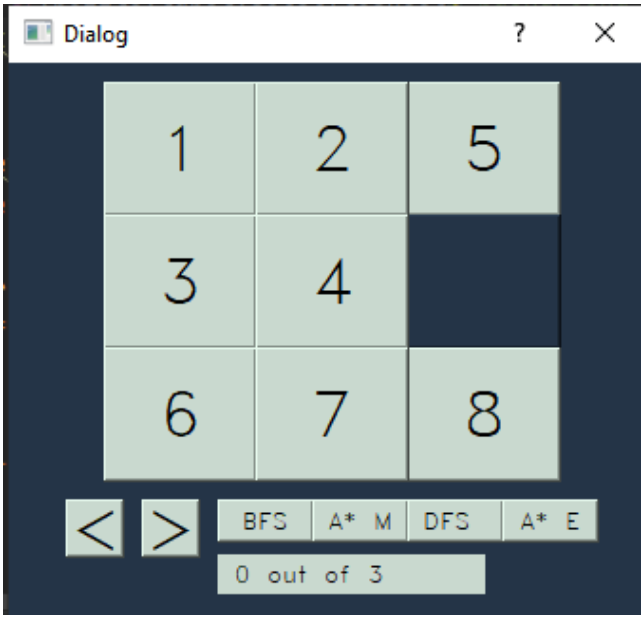
An 8 puzzle is a simple game consisting of a 3 x 3 grid (containing 9 squares). One of the squares is empty.
The object is to move to squares around into different positions and having the numbers displayed in the "goal state".

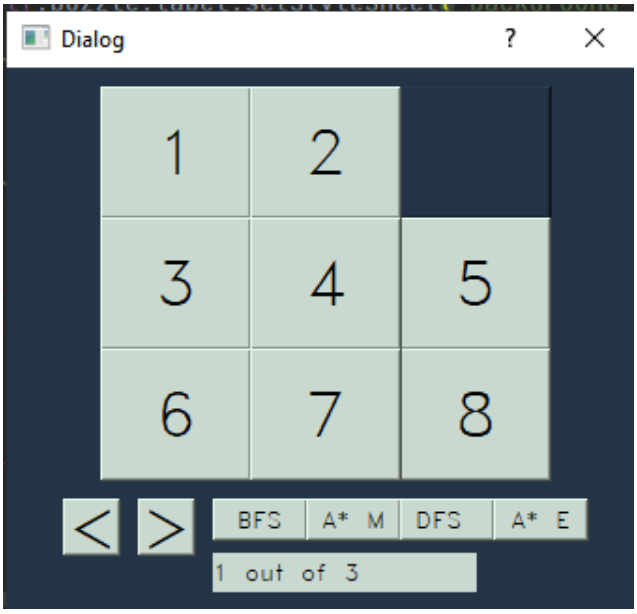
Sample Runs:

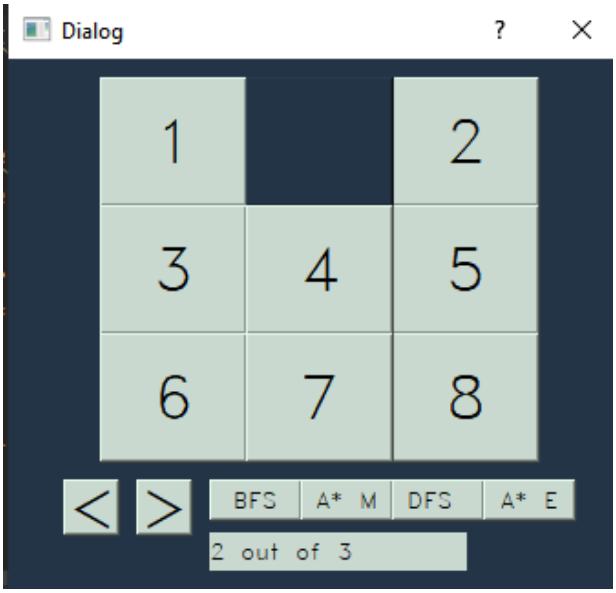
Once we run the code, the GUI window appears like the above figure. The 2 buttons at the bottom help the user to go the next state or the previous one.

The other buttons specifies the algorithms available to be used. The bottom label will show the number of remaining states.

When Choosing the BFS Algorithm:





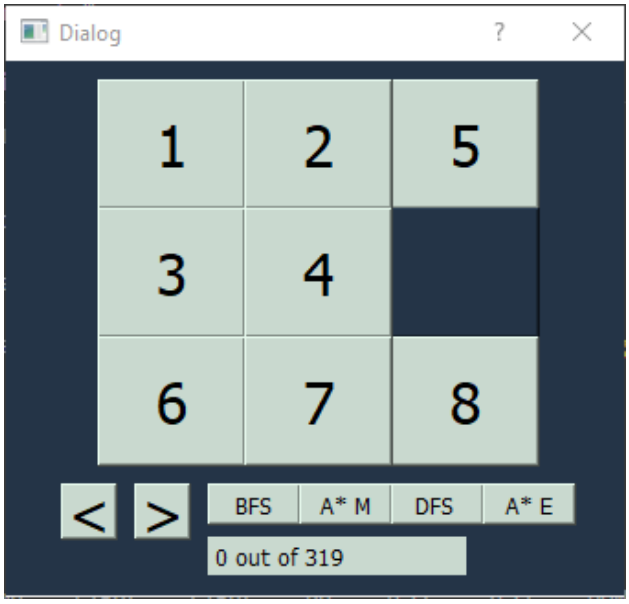


The path to goal is shown on the GUI window, the cost of the path, the number of nodes expanded, the search depth and the running time are printed on the console as shown into the next figure.

```
BFS
Goal achieved
12345678
Depth  3
Expanded  18
Total runtime of the BFS is 0.0009989738464355469
Cost 3
```

When choosing DFS Algorithm:

We'll only show the first few states as reaching the goal state would take 319 steps using DFS and the current standing tie breaker for the children of each state:




Dialog
?
1
2
5
3
4
6
7
8
<
>
BFS
A* M
DFS
A* E
2 out of 319

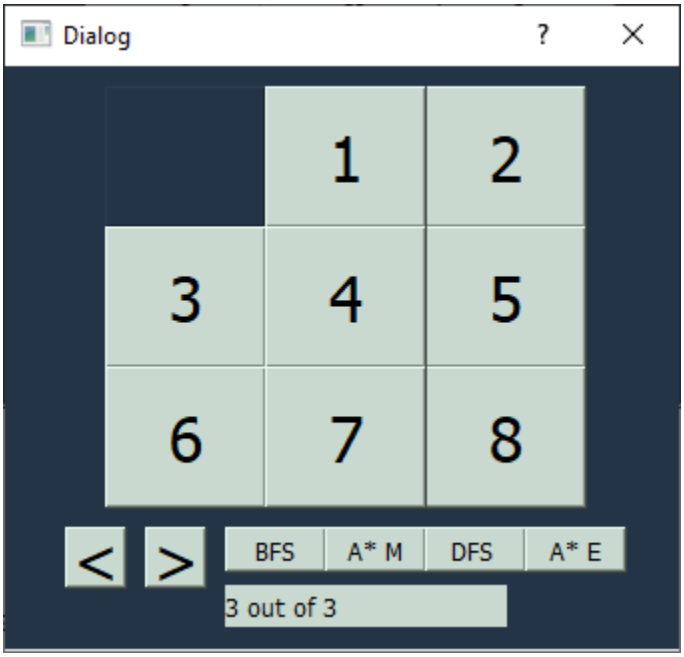

Dialog
?
1
2
5
6
3
4
7
8
<
>
BFS
A* M
DFS
A* E
3 out of 319


Dialog
?
1
2
5
6
3
4
7
8
<
>
BFS
A* M
DFS
A* E
4 out of 319

And so on..

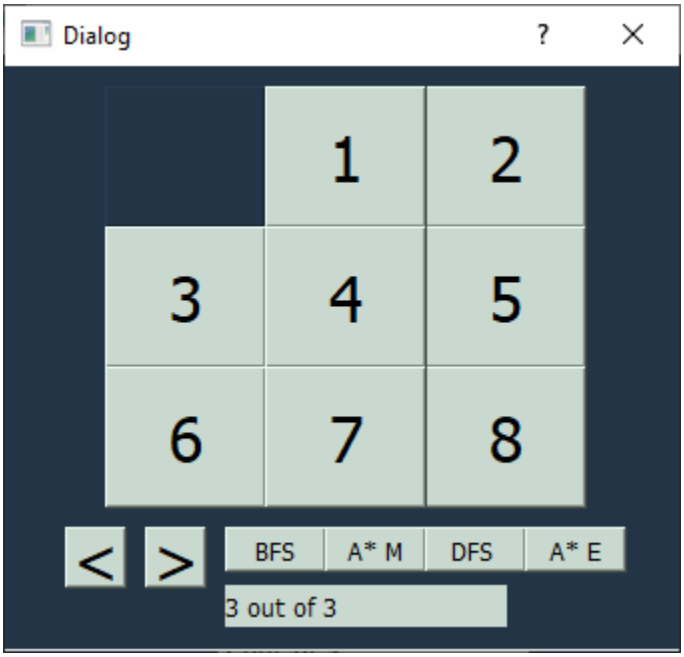
```
DFS
Goal achieved
Depth  319
Expanded  327
Total runtime of the DFS is 0.017989158630371094
Cost 319
```

When choosing A-star Euclidean Algorithm:




```
Astar Euclidean
Goal Achieved
12345678
Depth  3
Expanded  11
Total runtime of the Astar is 0.0019986629486083984
Cost 3
```

When choosing A-star Manhattan Algorithm:




```
Astar Manhattan
Goal Achieved
12345678
Depth  3
Expanded  4
Total runtime of the Astar is 0.0036132335662841797
Cost 3
```

The Code Structure:

We have 5 code files assigned:

Main: That is the main code which contains common functions used between all the classes, it contains the class state that has it sown attributes like the value, the action taken, the parent, the depth and the variable (f=g+h) used in the A* algorithm. The main contains the function children which gets the neighbors/children of the current state and finally the main code snippet from which we can start executing the whole code.

BFS.py: This file contains the BFS algorithm which begins which setting the time to compute it later after the algorithm is done, initializing the Frontier queue and the Frontier set (which is used in the search to have a more optimized running time), the explored set and the parent map. Then inserting the start state into the Frontier set and set.

Then, the main WHILE loop begins with the condition that the frontier is not empty, we pop the first element inserted in the Frontier (FIFO), adding the state to the explored. This line **depth = max(depth, s.depth)** is computing the max depth achieved each time we starts the while loop. And then we have the if condition which checks if the current state is a goal or not, if it is, the program will terminate, printing the corresponding data required. If not, it starts a for loop which is adding the state to the frontier if it is not in the frontier sets and not in the explored set, adding the child and its parent into the parent map initialized in the beginning of the code. After that, we end the timer here, printing the time taken and the path of the goal. If the frontier was emptied before reaching the goal, a "Goal is not found" will be printed on the console.

The file contains another function called **path** which computes and prints the path taken to achieve the goal as well as the cost. (We are assuming that each state costs 1).

DFS: This file contains the DFS algorithm which begins by setting the time to compute it later after the algorithm is done, initializing the Frontier stack and the Frontier set (which is used in the search to have a more optimized running time), the explored set and the parent map. Then inserting the start state into the Frontier and set.

Then, the main WHILE loop begins with the condition that the frontier is not empty, we pop the last element inserted in the Frontier (LIFO), adding the state to the explored. This line **depth = max(depth, s.depth)** is computing the max depth. And then we have the if condition which checks if the current state is a goal or not, if it is, the program will terminate, printing the corresponding data required. If not, it starts a for loop which is adding the state to the frontier

if it is not in the frontier sets and not in the explored set, adding the child and its parent into the parent map initialized in the beginning of the code. After that, we end the timer here, printing the time taken and the path of the goal. If the frontier was emptied before reaching the goal, a "Goal is not found" will be printed on the console.

The file contains another function called **path** which computes and prints the path taken to achieve the goal as well as the cost. (We are assuming that each state costs 1).

Astar: The Astar.py file's structure is the same as BFS and DFS, though it includes two heuristic functions; one uses a Euclidean heuristic and the other uses Manhattan's. For both functions we begin by looping over the goal state and getting each digit's goal position and adding them to an array where the index corresponds with the digit's value, so we can access them later for comparisons. For Euclidean, we calculate the position of the current state compared to the previous state using the equation:

$$h = \text{sqrt}((\text{currentcell:x} - \text{goal:x})2 + (\text{currentcell.y} - \text{goal:y})2)$$

and for Manhattan's we use this equation:

$$h = \text{abs}(\text{currentcell:x} - \text{goal:x}) + \text{abs}(\text{currentcell:y} - \text{goal:y})$$

We use a **mode** variable that is passed to the solve function to choose either Manhattan ("M") or Euclidean ("E"). The data structures used for frontier is a Priority Queue as well as an additional frontier for speedy search, of type set.

Note: We use a variable goal, that can be changed to any value.

Puzzle.py: It is the generated code of QT5 designs in order to be used and linked to the other files.

Board_gui.py: This file contains the main from which we can run the code associated with the GUI. It calls the BFS, DFS, A* functions to be executes in a GUI-way to the user.

Note: To run the code with a GUI please run the Board_gui.py file otherwise use the main.py file