CS451/551 – Introduction to Artificial Intelligence
Assignment-1

HW1

Solving N-Puzzle Problem with Search Algorithms

TOLGA GUMUSCU

S017901

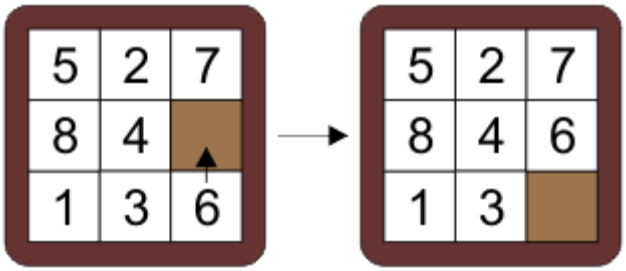
Table of Contents

Introduction

Our first homework is to bring the N-Puzzle Problem from 4 different algorithm initial states -> to the target state. In this context, the 4 algorithms we should use are breadth-first search (BFS), depth-first search (DFS), uniform cost search (UCS) and A * algorithm. 3 of the 7 classes given to us within the scope of the homework are ready, and we are responsible for coding the algorithm classes.

N-Puzzle Problem

N-Puzzle with 9 Slots: It is a game or problem consisting of 8 pieces that can be played and 1 space. In this system designed as 3X3,

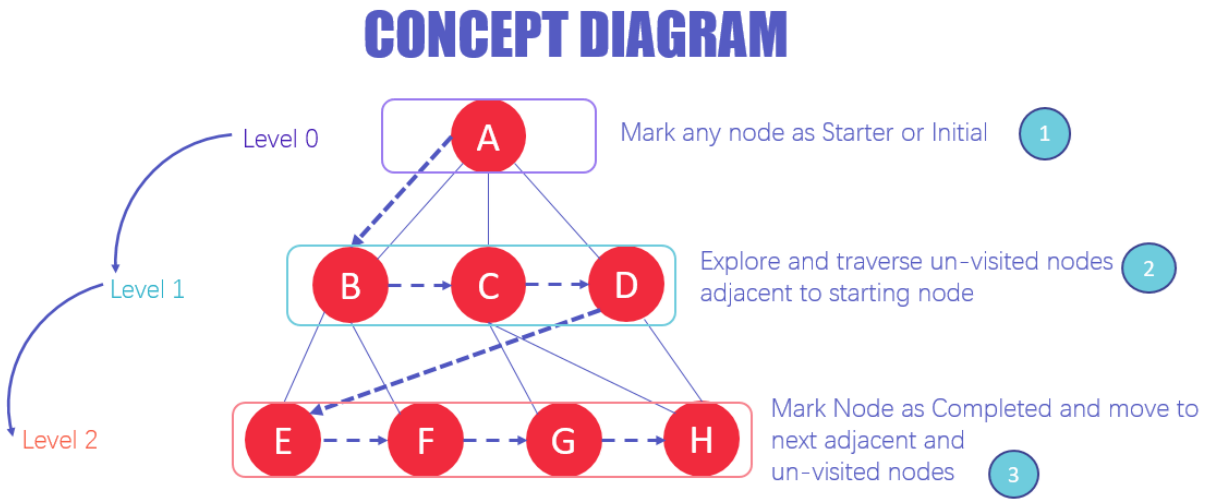


our aim is to make it a target from the current situation, there are also games consisting of pictures in the form of N-Puzzle.

The purpose of the puzzle is to place the tiles in numerical order. Tiles located in the same row or column of the open position can be moved by shifting them horizontally or vertically, respectively.

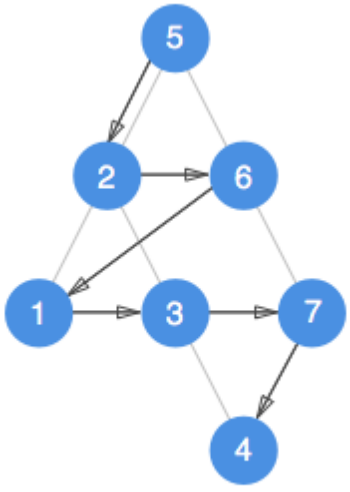
Algorithms

1) Breadth First Search



The working principle of the algorithm is as follows, it traverses and marks all the nodes in a graph, then determines a node as the starting point and roams all the nodes associated with the selected node (BFS provides access to the nodes one by one).

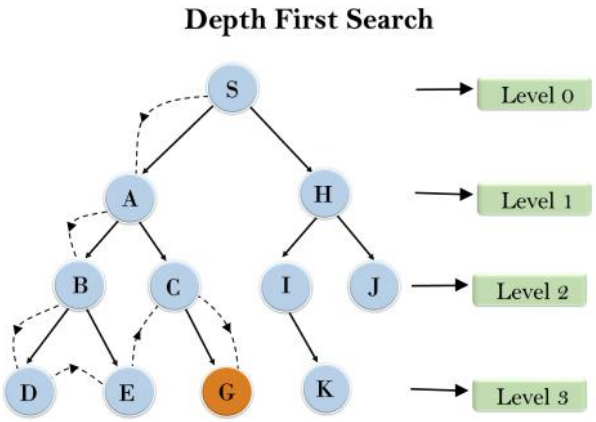
After BFS traverses and marks the starting node, it navigates and analyzes the closest uncharted nodes around it. After these cycles, this process is repeated until all the nodes in Graph are traversed.



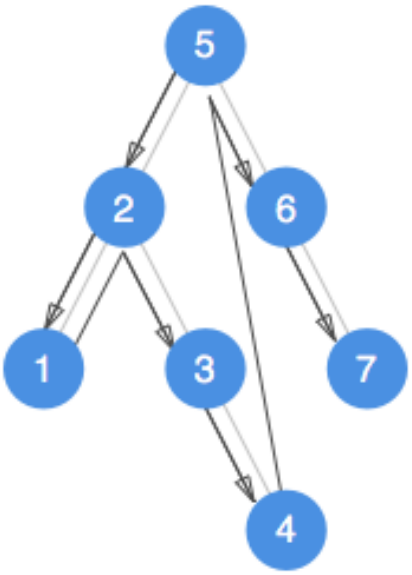
Breadth-first traversal

2)Depth First Search

The DFS algorithm starts from a root node we specify and selects any of its children. Then, this kid goes to any node we have not visited before. We have two rules in these elections, the first must be a direct route to the node we will choose, and the other one must not have been visited before. Following these rules, they are toured recursively.



However, we come to such a node that there is no place to go. Because all the nodes with which we have a direct connection have been visited before. In such a case, we go back to the node we last visited before this node and check if there are any other preferences. Since the algorithm works recursively, it allows us to go up at every point we get stuck. Our algorithm ends when we get to where we started and when we are already visiting all our kids.

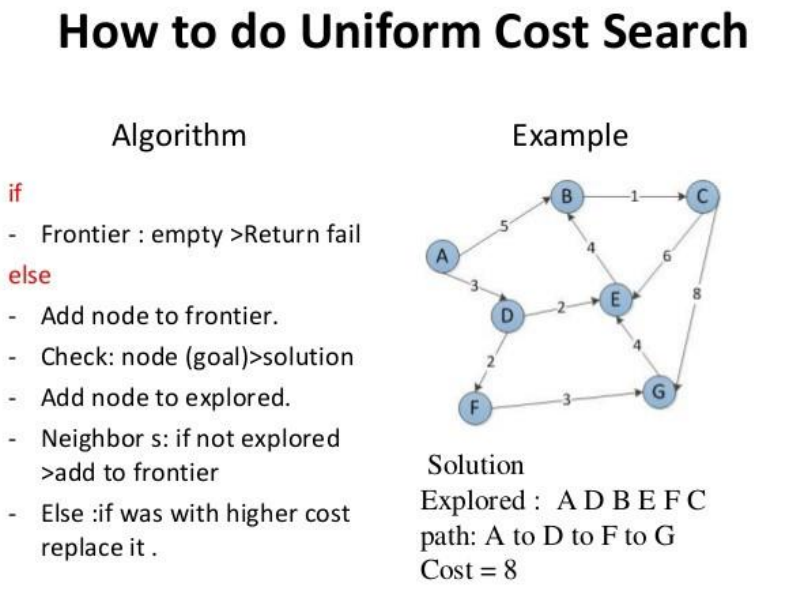


Depth-first traversal

3)Uniform Cost Search

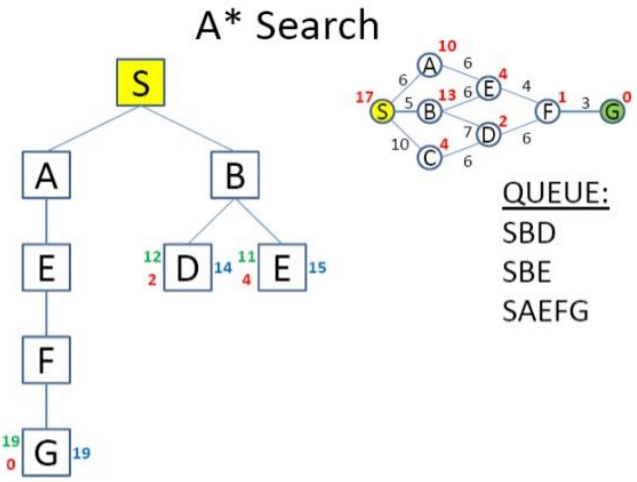
It is a term used in computer science for search algorithms. The algorithm works on weighted graphs. Since trees are an example of a graph, it is also possible for the algorithm to work on trees. The algorithm can be simply defined as follows:

1) Start from root node (root node)
2) Go to the lowest cost neighbor
3) If the searched node is found the bit, if not, go back to Step 2.



In this algorithm from the beginning state we will visit the nearby states and will pick the most un-expensive state then we will pick the following least exorbitant state from the all un-visited and contiguous conditions of the visited states, in this way we will attempt to arrive at the objective state (note we won't proceed with the way through an objective state), regardless of whether we arrive at the objective state we will keep looking for other potential ways(if there are various objectives) .

4) A*(A Star)



In short, it is the "best fit" algorithm to find the shortest nodes to go from a node to a target node.

The A * algorithm can be classified as an admissible heuristic algorithm. The reason for this is the function the algorithm uses to calculate the distance:

$$f(n) = g(n) + h(n)$$

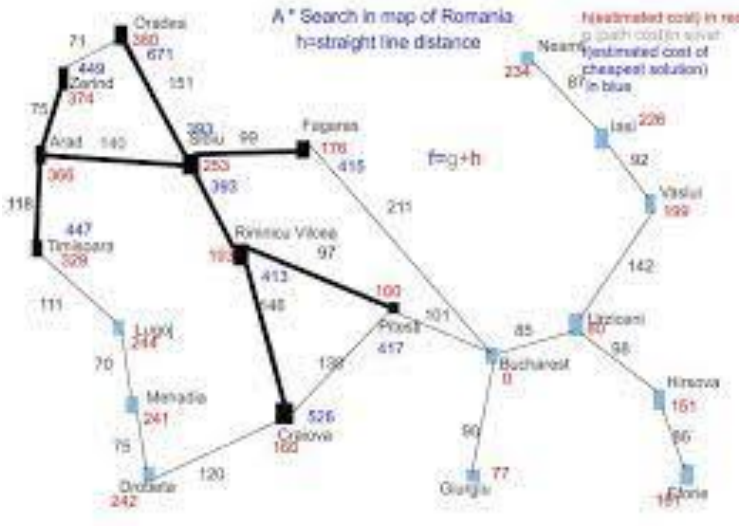
in the equation

f (n) = heuristic function that does the calculation.

g (n) = Cost from origin node to current node

h (n) = Estimated distance from current node to destination node.

As will be noted, the reason f (n) is intuitive is the heuristic function h (n), which is included in this function and is based on predictions.



The algorithm works:

The algorithm has a fairly simple structure using the addition process above. In the algorithm that uses a priority queue as its data structure, the node with the highest priority is the node with the lowest f (n) value.

A. At each step the algorithm takes the lowest value (And hence the most important) node (i.e. goes to this node) and removes the node from the queue.
B. The values of all the neighboring nodes according to this node are updated (now there is a cost to come to this node and this value is included in the function f (n).
C. The algorithm repeats the above steps until it reaches the target (i.e., the target node comes first in the priority queue) or until there is no node in the queue.

Implementation Details

BFS:

```
def run(self, target):
    """ YOUR CODE HERE """
    #control = target.is_equal(self.queue[0])

    while len(self.queue):
        control = self.queue.pop(0)
        self.visited[control.UID] = control
        self.counter += 1

        down = self.graph.reveal_neighbors(control)

        for i in down:
            if i.UID not in self.visited.keys():
                self.queue.append(i)
        if control.UID == target.UID:
            return True, self.counter, control.step

    # return 3 items
    # a: bool (match found or not)
    # b: int (counter)
    # c: int (depth)
```

- I repeat the while loop for the size of the Queue, then I get the first element "pop". Later, I compare it with the nodes I have visited but do not want to visit again. In each loop, the number of counters and the number of nodes we visit increases to print.

DFS:

```
def run(self, target):
    """ YOUR CODE HERE """
    boardVisited = set()
    global control
    while len(self.stack):
        control = self.stack.pop(-1)
        self.visited[control.UID] = control
        boardVisited.add(control.UID)
        self.counter += 1

        if control.UID == target.UID:
            return True, self.counter, control.step

        possiblePaths = reversed(self.graph.reveal_neighbors(control))

        for path in possiblePaths:
            if path.UID not in self.visited.keys():
                nearby = [path]
                self.stack.append(path)
                self.stack.extend(nearby)
                boardVisited.add(path.UID)
    # return 3 items
    # a: bool (match found or not)
    # b: int (counter)
    # c: int (depth)
    return False, 0, 0
```

- I repeat the DFS algorithm for the size of the stack, I define it as the first element of the variable stack that I call "control". If our control and goal state are the same, I return. Depth is defined with the step of the node, of course, if the match is completed. I created "possiblePaths" by taking the reverse to examine it with the LIFO concept.

UCS:

```
def run(self, target):
    """ YOUR CODE HERE """
    while not self.queue.empty():
        point = self.queue.get(False, None)[1]
        self.counter += 1
        if point.UID == target.UID:
            return True, self.counter, point.step
        else:
            list_neighbour = self.graph.reveal_neighbors(point)
            for nearby in list_neighbour:
                if nearby.UID not in self.visited.keys():
                    self.queue.put((nearby.step + target.step, nearby, int(nearby.UID)))
        if point.UID == target.UID:
            return True, self.counter, point.step

    # return 3 items
    # a: bool (match found or not)
    # b: int (counter)
    # c: int (depth)
    return False, 0, 0
```

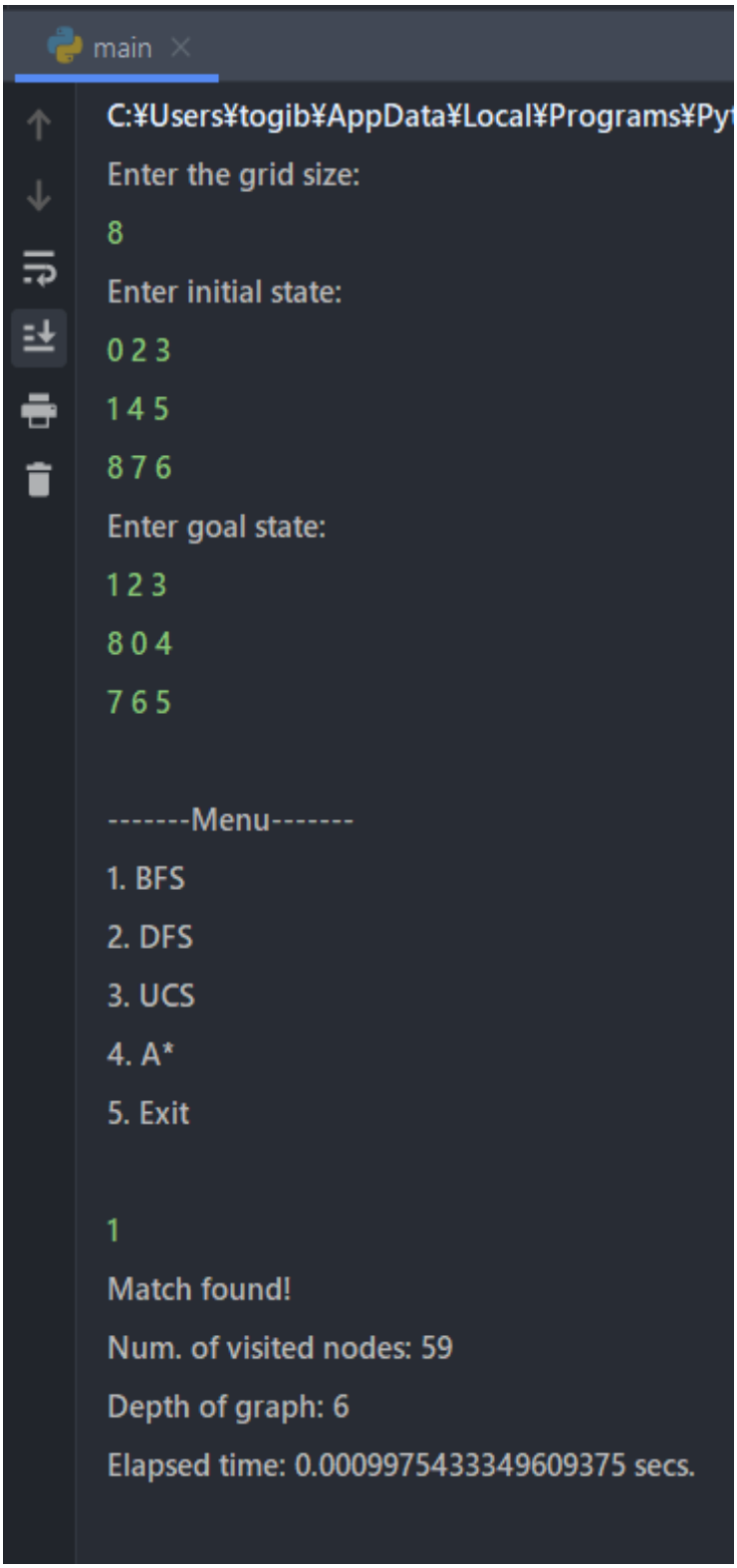
I repeat the UCS algorithm as long as the queue is not empty. If our node matches the target.UID, we return True, but otherwise we roam the neighbors until it is True. Except for append or extend, I used put () function here, when I get an element from the queue, it returns the node with the least distance to me.

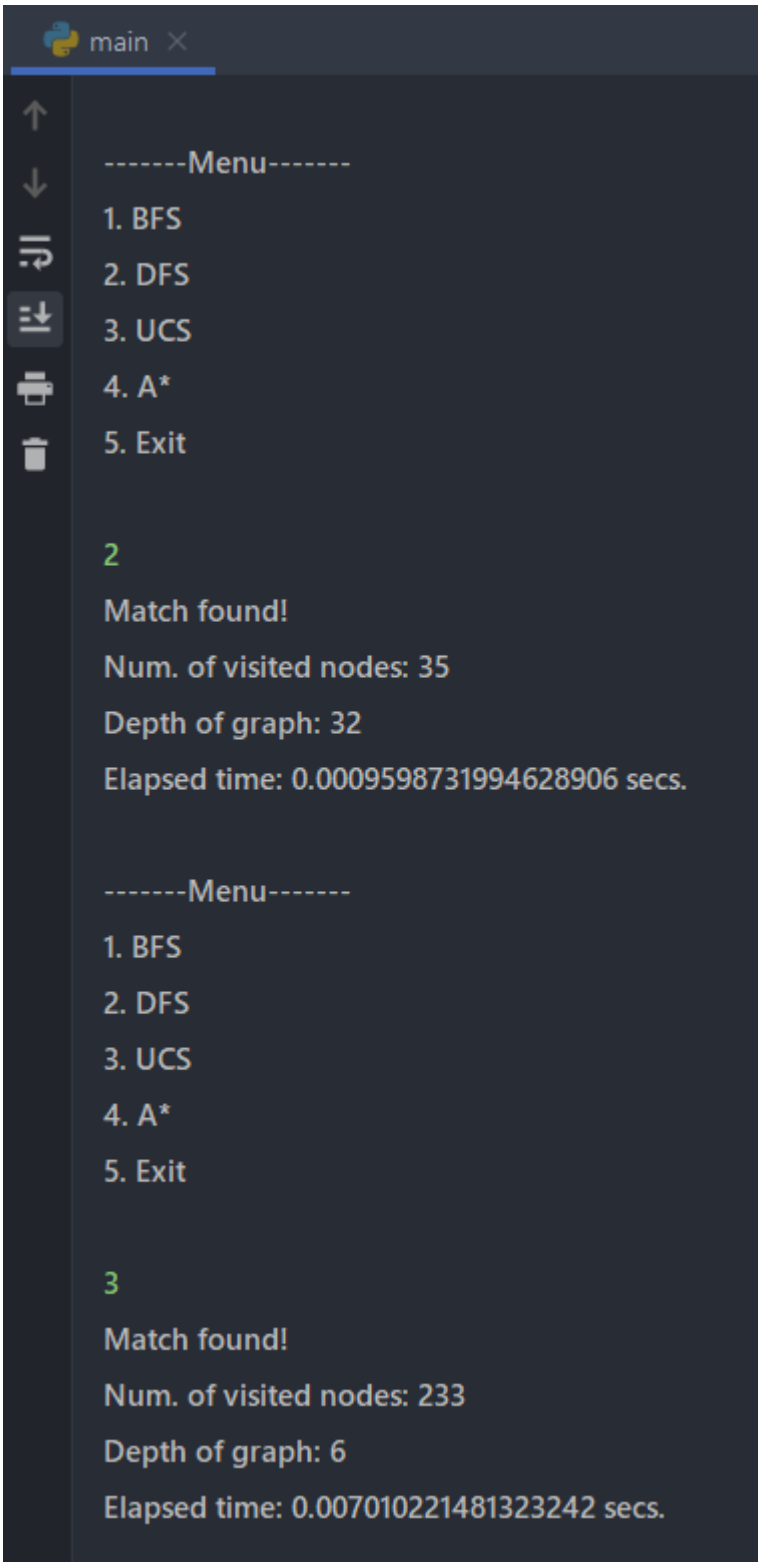
A*

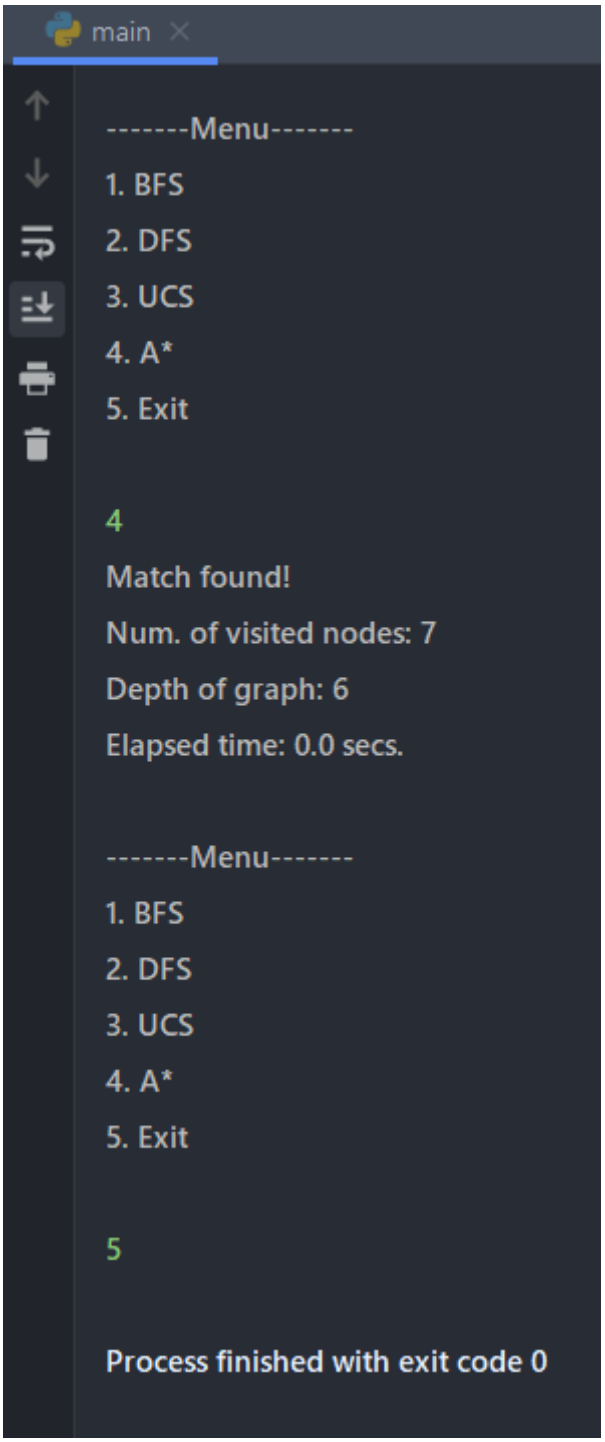
```
def run(self, target):
    """ YOUR CODE HERE """
    # I would create counter as 0 but we already assigned at the top
    self.queue.put((self.manhattan_distance(self.root, target),self.root,int(self.root.UID)))
    already_been = {}
    in_line = Q.PriorityQueue()
    in_line.put((1,self.root))
    already_been[self.root.UID] = self.root

    while not self.queue.empty():
        self.counter +=1
        last_one = self.queue.get(False, None)[1]

        check = bool(last_one.UID == target.UID)

        if (check):
            return True, self.counter, last_one.step

        close = self.graph.reveal_neighbors(last_one)

        for i in range(len(close)):
            if close[i].UID not in already_been:
                length = self.manhattan_distance(close[i],target)
                self.queue.put((length, close[i], int(close[i].UID)))

    # return 3 items
    # a: bool (match found or not)
    # b: int (counter)
    # c: int (depth)
    return False, 0, 0
```

Just like in the UCS algorithm, we use priority queue here, I rotate it as long as the queue is not empty, one of the reasons why this algorithm works faster and optimally than other algorithms is that it is a heuristic algorithm. We had to use the "manhattan_distance" function to use this feature. Thanks to this function, we get the distance between two nodes

Results


main
C:¥Users¥togib¥AppData¥Local¥Programs¥Pyt
Enter the grid size:
8
Enter initial state:
0 2 3
1 4 5
8 7 6
Enter goal state:
1 2 3
8 0 4
7 6 5
-------Menu-------
1. BFS
2. DFS
3. UCS
4. A*
5. Exit
1
Match found!
Num. of visited nodes: 59
Depth of graph: 6
Elapsed time: 0.0009975433349609375 secs.


main
-------Menu-------
1. BFS
2. DFS
3. UCS
4. A*
5. Exit
2
Match found!
Num. of visited nodes: 35
Depth of graph: 32
Elapsed time: 0.0009598731994628906 secs.
-------Menu-------
1. BFS
2. DFS
3. UCS
4. A*
5. Exit
3
Match found!
Num. of visited nodes: 233
Depth of graph: 6
Elapsed time: 0.007010221481323242 secs.


main
-------Menu-------
1. BFS
2. DFS
3. UCS
4. A*
5. Exit
4
Match found!
Num. of visited nodes: 7
Depth of graph: 6
Elapsed time: 0.0 secs.
-------Menu-------
1. BFS
2. DFS
3. UCS
4. A*
5. Exit
5
Process finished with exit code 0

Conclusion

Iterations: A* has less number of iterations than the other algorithms.

Time: A* is the fastest algorithm. Then comes DFS with higher time. BFS is the slowest.

Depth: The DFS algorithm has the most Depth with the initial and goal states we have used.

References

https://en.wikipedia.org/wiki/15_puzzle

https://en.wikipedia.org/wiki/Breadth-first_search

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

https://en.wikipedia.org/wiki/Depth-first_search

https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/

https://en.wikipedia.org/wiki/A*_search_algorithm

http://bilgisayarkavramlari.com/2009/03/02/a-yildiz-arama-algoritmasi-a-star-search-algorithm-a/