Final Project

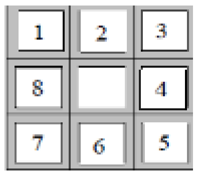
By: Dyllan Britz, Darien Henrie, Alexandre Lopes F. Da Silva

Introduction

The following report is written in plans of using for the submission to the final report for a project in the Analysis of Algorithms at the University of South Florida. The purpose is to represent the input of users 8-puzzle diagram modeled by:

```
1 3 4
8 0 2
7 6 5
```

The program will execute Breadth First Search (BFS), Depth First Search (DFS), and Dijkstra's algorithm on the given graph, and will output the most optimal choice, and the weight of that choice is given the inputs of the user. Overall the purpose is to take the graph and organize it in numerical order after the user's inputs are finalized. This would make the graph output always modeled as:



Background

Before we begin analyzing the output of the program, it is essential to understand what each type of search algorithm is meant to do, and what any assumptions within the projects are. There are four main parts of the projects we can consider for assumptions during this project. The first one is the definition of BFS, the second is the definition of DFS, the definition of Dijkstra's algorithm, and the final one is the assumption of what the weight of a graph is.

BFS is a sorting algorithm in which a tree or graph is sorted starting at the root of the tree and branches to every node of a tree or graph. From the node, the algorithm will consider all nodes at the same depth as the current node and/or the root node, before moving to a node at a different depth and continuing from that depth to the next layer of the tree or graph. The easiest way to view this implementation is to view each node as visited or nonvisited and store each part of the graph adjacent to the starting node in a queue and operate basic FIFO principles and use the next part of the algorithm to use the next part of the queue as the current node and repeat the process over and over again until there are no other portions that can be executed.[1]

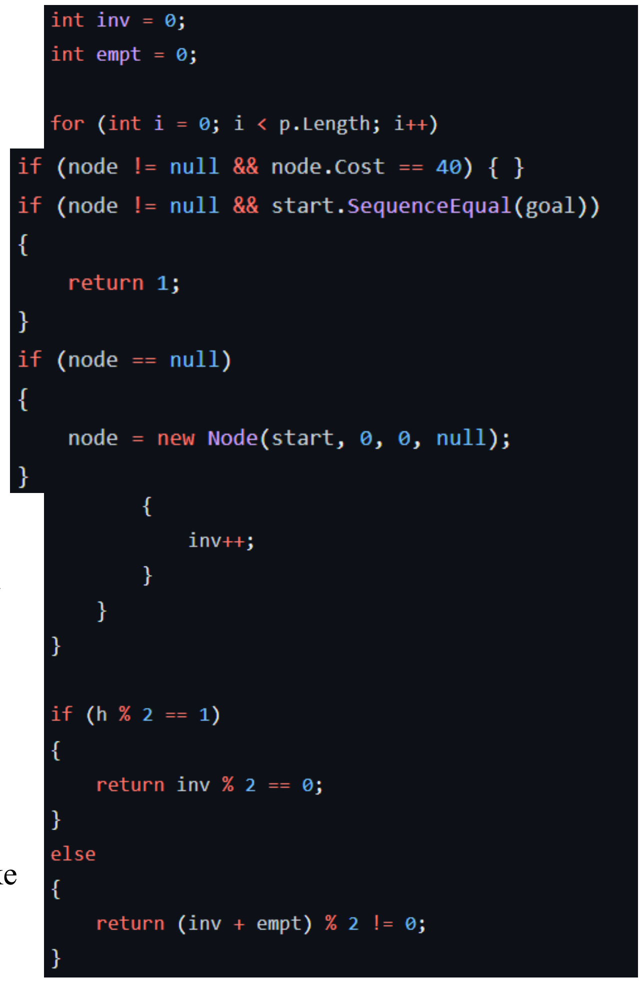
DFS is also a searching algorithm that's purpose is to travel as far as possible from the root node before backtracking. In other words, the purpose is to start from the root node or whatever node is deemed the starting node and go until all nodes within the graph or tree are marked as visited. Finally, the best thing to do is backtrack and check that all unmarked nodes were transversed, and show the path that was followed for the algorithm. [2]

Dijkstra's algorithm is the next assumption we made during the process of making this project. Disktra's algorithm is meant for the purpose of finding the minimum spanning tree or path depending on its implementation. In other words sort through the data and find the shortest path possible from the initial node to the final node. [3]

The final assumption is the calculation of weight within a given graph or tree, and the best way to calculate the overall cost of implementing each given search algorithm. Overall for this circumstance, we will consider a 1 in the graph as purchasing something at the store and this would mean that if a 1 is $1 then an 8 in the graph would be $8 with each number being equivalent to its dollar amount. This weight is the core principle of how the efficiency of the graph search algorithm will operate. [4]

Implementation

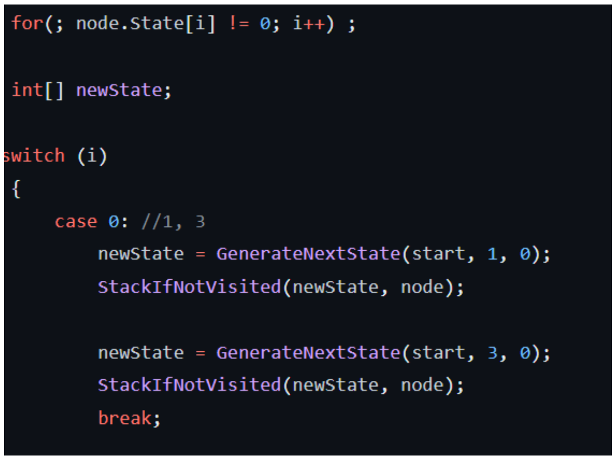
The implementation we deemed the best way to implement a separate class for each algorithm within their own separate files. This was done using either a linked list or an array to represent the graph and to create different transversals. Each element of the list and array was given a weight, which was added to the total weight of the search type. The first part of the program is meant to test if the graph is solvable, which was done through the structure seen on the right, where it tests to make sure the length, does not equal 0, and to make



```
int inv = 0;
int empt = 0;

for (int i = 0; i < p.Length; i++)
if (node != null && node.Cost == 40) { }
if (node != null && start.SequenceEqual(goal))
{
   return 1;
}
if (node == null)
{
   node = new Node(start, 0, 0, null);
}
        {
            inv++;
        }
    }
}

if (h % 2 == 1)
{
    return inv % 2 == 0;
}
else
{
    return (inv + empt) % 2 != 0;
}
```

sure that the total graph size is even or not, where w is the width of the graph, and h is the height of the graph. If it returns with anything but the height being divisible by 2 with a remainder of 1 then the graph will be unsolvable. Because the purpose is to have a 3x3 graph that is able to be solved through graph manipulation. This is essential to be analyzed because as mentioned previously the purpose is to get the graph into numerical order from top left to bottom right.

Following this structure, there are two other major universal structures used for all searching algorithms. The first is a check for if the nodes are null, meaning a graph in the proper format was stored, but the graph has no real values. The first check is to test if the node is null, and the second one is to check if the initial graph is already in the format of the goal state. This means it would only cost 1 to check if the graph is in its final state, or if the graph is null. After is the initialization as nodes for later use.

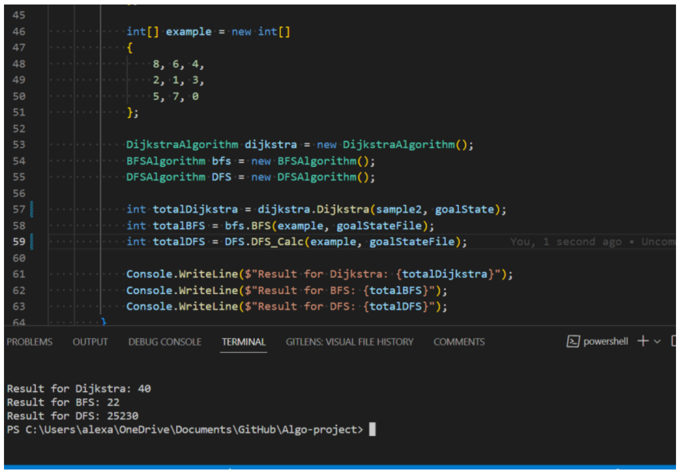


```
for(; node.State[i] != 0; i++) ;

int[] newState;

switch (i)
{
    case 0: //1, 3
        newState = GenerateNextState(start, 1, 0);
        StackIfNotVisited(newState, node);

        newState = GenerateNextState(start, 3, 0);
        StackIfNotVisited(newState, node);
        break;
```

Finally, all three searching algorithms use a switch case to analyze each number, starting from the front. From the image of DFS, the structure for the switch case that will loop until the final value is determined is seen on the left. The only main variance is the type of structure used to transverse the new state with the current node in the position. In DFS the implementation is a stack, that will add the new position to a stack, and will export them in the order that they are added to the stack. For BFS the nodes are added to a Queue and then Dequeued based on the order in which they arrived in. Finally, Dijkstra operates an enqueue as well while comparing both moving options and calculating which will be optimal instead of just iterating through the data in an ambient order.

Conclusions

The results of the graphs were tested by pushing different test graphs through each implementation and comparing the results printed by the respective algorithms. The code for how each respective function was called and how they were output for the comparison is on the right. The output of the program will be situated as shown in the image by displaying all of the outputs in order to show the user how it was simulated. Overall our group found that BFS and Dijkstra often competed with one another in order to test which had the faster result speed. We calculated it based on which desired output for the two was closer to the input. Dijkstra was found to have the fastest run-time efficiency with the combination of all the algorithms taking about 2 minutes to compile and output the results due to the sorting of DFS and BFS being in a much more random manner. That is also why the DFS has a result in the 10's of thousands and given some other inputs this could also be true for BFS. We were surprised that BFS could compete with Dijkstra given the right inputs, but in the end, Dijkstra was proven to not only be a more optimal choice in terms of cost, but also a faster-running choice given the same structure from algorithm to algorithm, and overall Dijkstra is the most practical for real-time usage of solving problems in a similar manner.



```
            int[] example = new int[]
            {
                8, 6, 4,
                2, 1, 3,
                5, 7, 0
            };

            DijkstraAlgorithm dijkstra = new DijkstraAlgorithm();
            BFSAlgorithm bfs = new BFSAlgorithm();
            DFSAlgorithm DFS = new DFSAlgorithm();

            int totalDijkstra = dijkstra.Dijkstra(sample2, goalState);
            int totalBFS = bfs.BFS(example, goalStateFile);
            int totalDFS = DFS.DFS_Calc(example, goalStateFile);        You, 1 second ago • Uncom

            Console.WriteLine($"Result for Dijkstra: {totalDijkstra}");
            Console.WriteLine($"Result for BFS: {totalBFS}");
            Console.WriteLine($"Result for DFS: {totalDFS}");
        }
```

```
Result for Dijkstra: 40
Result for BFS: 22
Result for DFS: 25230
PS C:\Users\alexa\OneDrive\Documents\GitHub\Algo-project>
```

References:

1. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
2. https://www.dotnetforall.com/depth-first-search-algorithm-dfs-csharp/
3. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
4. https://iq.opengenus.org/weight-balanced-binary-tree/