Austin Friedrich

933966744

friedrau@oregonstate.edu

Artificial Intelligence

Project 1

Methodology:

The idea behind my program is to create every possible outcome for every possible valid move to generate a graph. This graph completes when one of the states in the graph is equivalent to the Goal.txt file. It loops until a solution is found. The graph takes care of game rules, meaning that every option generated is a valid move. Graphmaker.cpp creates an unweighted graph. If a node in that graph is equivalent to another node, that nodes priority becomes the same as the equivalent node. I.E a loop is formed.

```
Tree Complete, Choose Search Algorithm.

***SOLUTION FOUND BFS***
Nodes Expanded in search 15
: BFS SOLUTION :
**************************
Leftshore       : 3,3,1
Rightshore      : 0,0,0
Prority number : 69
Parent number   : 56
Array Size      : 1
**************************

CHILD ABOVE |*| PARENT BELOW
**************************
Leftshore       : 2,2,0
Rightshore      : 1,1,1
Prority number : 56
Parent number   : 46
**************************
```

Figure 1 Example of BFS output

Figure 1 on the right shows two nodes, When Graph maker creates the graph, it creates a BFS Heap (figure2). The node number is the priority, a nodes parent number is that nodes connection to the previous node. By doing this method I create an even playing field for all the algorithms. Since they don't have to make anything, I just plug them into the graph to run. The graph also turns in on itself when it gets a duplicate Right Shore, Left Shore value. This way it avoids turning into a Tree graph.

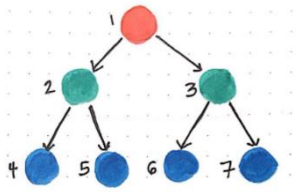

Figure 2 How Graph maker creates the starting graph Each number is a priority; each priority has their parents' priority

The complexity of the Graphmaker is O(n!) so this solution would not work if I don't cut down the number of choices. While generating valid paths I exclude repeated moves and returning to the root node. In figure 1, priority 69 could not be equal to priority 46 because that would be a repeated move. I also apply something I call a Value Engine. It works a little like ASTAR. It allows for a tolerance of 100 nodes before it increments minimum score value by 1. If a node has a calculated Value Engine score, then the minimum score value, that node is rejected from the graph.

BFS: The graph from graphmaker.cpp is generated as a priority queue that's a solution for BFS. It goes left to right one level at a time. Not a whole lot to say on this one. I could have just run a for loop and gotten the same result as the BFS. However, my BFS assumes that the graph is not in order. It creates a queue of my custom struct and pops it every time it looks for a new node. it then adds the pop nodes children to the back of the queue.

DFS: Only needed around 2-3 lines of code to change from BFS for DFS to work, instead of loading all the structs to the back of queue, we instead send them to the front. Now the left-hand side of the array is completed before the right.

IDDFS: The way this works in my program, is a DFS search with a limit of depth 5 (5 nodes away from the root node). During my search if a parent enters the queue with a depth of over 5 it gets added

to another history queue. When the current queue is zero then I append history queue to the current queue. Then Increment the depth limit by 5, this results in checking 5 levels every time.

ASTAR: This one was fun, to computer ASTAR heuristic I added up right shores wolfs and sheep. The more animals that ASTAR had on Right Shore the larger the heuristic, The movement cost was the nodes away from the root. The final cost of ASTAR was f = (animals – nodes from Root.) ASTAR generates this data for all items in the queue then goes down the best path, generates the numbers again and repeats.

Results: My best preformer was DFS, and IDDFS this was probably coicidence with a HEAP Graph. The shortest path is the same for every algorithm, since there is only one solution for the graph is ever

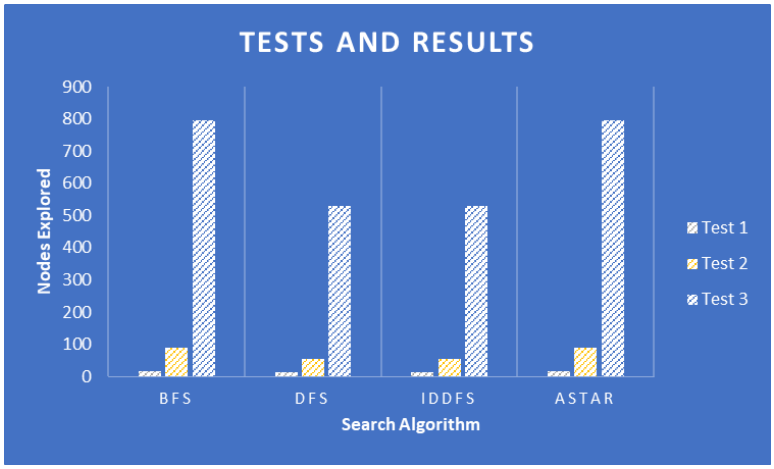

Figure 3 DFS and IDDFS preformed the best in testing

presented, the path to the root node is always the same. Even though Astar and BFS got the same results, im still really excited that my solution to the problem was more or less as effective as BFS.

Discussion:

It took more work then I inteded but with some cleaver solution making in graphmaker.cpp I was able to cut down the number of nodes required to generate a graph. Once the graph was generated using the Value Engine plugging the Algorithms in was the easy part. Going forward I would never use this solution again, it makes results too uniform.One solution exsists because that's the endstate for graphmaker.cpp that excludes any other possible solutions. The only varience we can observe from my solution is the nodes expanded to get to the result.

Conclusion:

The Results where not what I had expected. I wish I could have gone back to day one and tried again using a different method to achieve this result. What I can say is that this was a incredible hard programing feat for me, but the learning experience was good. DFS and IDDFS were the best preformers, and that makes sence if the solution is in the left half of the graph. Though my solution didn't work perfectly it was still an intresting experiment. I think its pretty cool that Astar was not the best pick, it shows that in small scale problems sometimes its better to keep it simple.