Samuel Cooney
CS 331

Wolves and Chickens River Problem

Methodology

I started the algorithms by focusing on the pseudocode provided by *A Modern Approach* and as well as the slides and developed the backbone pseudocode around that. I spent a while struggling while developing the bfs with my interactions between my child nodes and parent nodes. I was having boats not swapping sides in the correct areas and wolves and chickens going negative when my legal function prevented that. However, once I realized this was from a bug I implemented in the successor function, I had a much easier time implementing the rest of the assignment.

For the breadth-first algorithm I chose to use python3's queue.Queue(), which is a built in library providing a FIFO queue. I felt it was unnecessary to make my own queue when it was provided for me. My tests for BFS were promising with the results shown in Figure 1 below.

For the depth-first search algorithm, I realized I could have modularized my BFS code since I reused the almost all of the same pseudo-code with a LIFO queue instead. For the lifo queue I chose to use the queue.LifoQueue() for the same reason as discussed in BFS. Again, results are shown in Figure 1.

This algorithm gave me trouble. I am not sure if my implementation of the pseudo-code was incorrect or if my computer could not handle it. I was only able to receive a portion of results, as the other options just hung for minutes on end. I chose 500 for the depth limit as I felt it was large enough, and considering the program still hung for the larger tests, I don't think this was an issue.

For the astar I ran into the same issue as dfs where I could have modularized much of the code to save space. I had to do a bit of research to find what I should do for the cost to reach the solution and after reading about Manhattan distance, I decided to calculate the score based on how many individuals were on the goal bank. I think another good score would be how close each individual chickens and wolves count is to the goal score.

Results

The following table represents the results for each test. A blank square is one that I stopped after 15-30 minutes (multiple retests proved to be the same).

	Test Start and Test Goal 1	Test Start and Test Goal 2	Test Start and Test Goal 3
BFS	Path Count: 11	Path Count: 39	

Samuel Cooney
CS 331

	Nodes: 46	Nodes: 533	
DFS	Path Count: 11 Nodes: 25	Path Count: 39 Nodes: 124	Path Count: 953 Nodes: 4549
IDDFS	Path Count: 11 Nodes: 18		
ASTAR	Path Count:11 Nodes: 29	Path Count: 39 Nodes: 99	Path Count: 387 Nodes: 2885

Discussion

The results were surprising yet almost expected. The one that surprised me the most was the Path Count for DFS being so high. I also wish I had the ability to see the Tests 2 and 3 for iddfs and bfs, but unfortunately my computer could not handle the size. I am not sure if this is because of a mistake in my programming, but it is expected that these results would take much longer.

Conclusion

From these results, I believe Astar with the priority queue is on average the best to use for this case. Even though IDDFS and DFS had smaller node counts for test 1, as the number of path counts grew, AStar performed the most efficiently, finding the path with the less nodes expanded and even a shorter path.

I also wasn't too disappointed by how BFS performed for the smaller tests. Even though the node count was slightly higher, because the path it found for the crossings was actually a different path from the solution provided in the assignment description and also a different solution than the DFS algorithm found. While both were 11 crossings, it was an interesting find.

If I were to code this again, I think I would modularize a lot of the code so I could reduce repeats. Between the algorithms, since a lot were based on the Graph search, I could have saved a large number of lines. I would like to have also automated some of the testing to run a single command and have it process all the algorithms and test cases.