Mini-Project 1

June Wang
jwang3213@gatech.edu

Abstract—This report will describe my Sheep and Wolves agent's design and performance in Mini-Project 1.

1 MINI-PROJECT 1 REPORT

1.1 How the agent generates and tests

My agent works by recursively using depth-first search and semantic networks to step through each of the newly generated, validated moves. The semantic network is connected through nodes, and each note is a state representing the number of sheep and wolves on each side of the river. In python, the states are represented as a list designed to visually mimic the problem:

[[number of sheep on the left, number of sheep on the right],

[number of wolves on the left, number of wolves on the right],

direction (Boolean variable for moving right)]

When listing all the possible moves from a state, I first consider the current state and direction. From this, I generate 5 new states from all 5 possible moves, disregarding the move that moves 0 animals. These moves are as follows: 1 sheep 1 wolf, 2 sheep, 2 wolves, 1 sheep, and 1 wolf. Depending on the direction (left or right), I compute the new state by adding or subtracting numbers from the state. After that, I validate all the generated states with a function that checks whether the move is physically possible (does not end up with a negative number of animals on a side) and is within the realms of the problem (number of wolves does not exceed sheep if there is at least one sheep on that side; if there are no sheep on a side, it doesn't matter how many wolves there are on that side).

Along with checking if a new state is valid, I also check to see if that state has been visited before; if we didn't check this, we would get into an infinite loop. I did this by creating a variable called that stored all the processed states. If a state is already in the list, then we don't want to check it. However, one caveat is that we need to check the direction as well. If we have visited a state while moving to the right, that state is still valid if we are now moving to the left.

The agent will then use DFS to recursively step through the validated new states. For each state that it processes, it adds the move it makes to a list called final_moves, which is the list that is ultimately returned in the solve function.

It will continue to do that until it receives an empty list of validated new states. This means that for that path, there is no solution. Thus, the agent will backtrack, removing the most recently added move in the final_moves list, and step through the next state on the list – finally, if the agent exhausts all the options, it will return an empty list, meaning that there is no valid solution.

1.2 Agent performance and efficiency

The agent performed well and efficiently; it did not struggle on any case in the subset that was tested both locally and on Gradescope and always solved the problem in the optimal number of moves.

However, I wanted to see what the upper limit to my agent was. The maximum number of animals tested on this assignment was 25 of each, and the agent solved all the given problems quickly and easily. When I started inputting large numbers into the test function locally, it was able to solve 200 sheep and 100 wolves in under a second. However, I noticed that as the numbers got larger (>400), I ran into a maximum recursion depth exceeded error.

I timed the results of each test case for up to 50 sheep and 50 wolves, and as expected, the greater the number of animals, the longer it took. I created a 3D graph below using matplotlib to indicate the agent performance. To minimize anomalies, I timed each test case 20 times and took the average of each.

As expected, the more animals there are, the longer it takes. Interestingly, there is a clear diagonal line in the 3D plot separating a flat ground (purple) and the contoured map – this is because it is impossible to solve any test case where there are more wolves than sheep, so this will result in a much faster runtime.

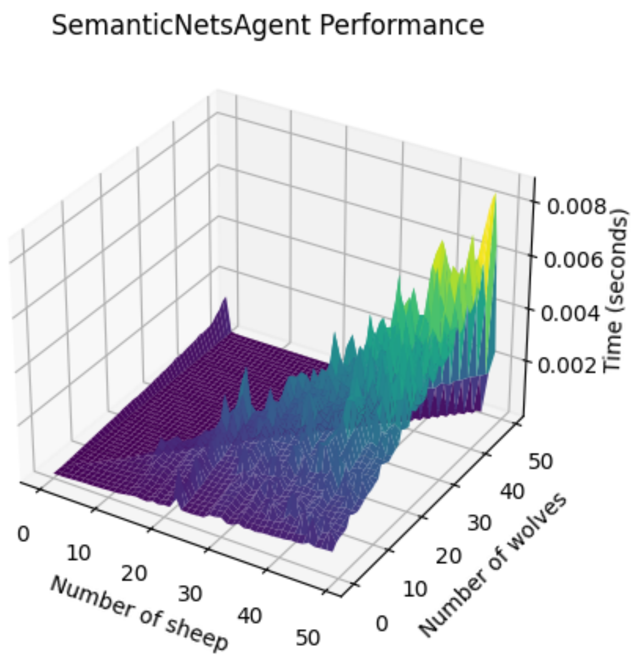

Figure 1—My Semantic Networks Agent Performance up to 50 sheep and 50 wolves.

1.3 Agent cleverness

One thing the agent does cleverly is when generating the new moves. When moving to the right, it will generate these following moves in order: 1 sheep 1 wolf, 2 sheep, 2 wolves, 1 sheep, 1 wolf in order. I thought that this was the most efficient way to generate moves, since it prioritizes moving a greater number of animals each time. Thus, the agent will always try to put 2 animals on the boat before exhausting its' options; then, it will try to put only 1. This works well on the first move, since the first move is always to move right, and will always want to move 2 animals.

However, when moving to the left, it will attempt to generate the moves in the opposite order to the right. I did this because the agent kept trying to move 1 sheep and 1 wolf on move 2, right after moving 1 sheep and 1 wolf at move 1, therefore wasting 2 moves. I'm sure that there is a better way to go about this but making this small change saved me 2 moves each time.

After making this change, I resubmitted and received full points on the coding portion of the assignment. If I wanted to make it even more efficient, I would hardcode a case that would automatically return an empty list if the initial number of sheep and wolves were equal, and if each number was greater than 3 (sheep == wolves > 3). This is because I discovered that the problem is impossible to solve for this state.

1.4 Agent behavior compared to a human

Before getting to the final solution, I spent hours debugging to figure out what was wrong with my recursion logic. Through this, I observed the similarities and differences in the way that I, a human, worked through the problem compared to my agent.

Initially, we went about it the same way. Since I created the agent, I wrote it so that it would make the same first move that I did – which is 1 sheep 1 wolf. I'm not sure why I favored this as a first move, but it seemed the most efficient. Working through the problem on paper helped me discover bugs in my code. For example, it is clear if you write out the problem, that having 2 wolves on the left and 2 sheep on the right is a valid state. However, my code did not initially have that logic – it only checked to see if the number of wolves was less than or equal to number of sheep. It was only after writing it out, that I added in the caveat that this only applied if the number of sheep was greater or equal to 1.

One thing that the agent did better than me was that it kept track of previously visited states. When I was solving the problem, I did not consider keeping track of that because it was too tedious, so I would constantly forget which states I'd already tried to solve the problem with. I found that past 3 sheep and 3 wolves, I just didn't have the willpower to solve the problem on paper anymore. 11 final moves are nothing to the agent – but for a human, it can be quite time consuming to come to the right solution.

I think that the agent would beat out any human, simply due to the fact that humans cannot process that many moves in a short amount of time.