Sheep & Wolves: Mini-Project 1 Report

Youssef Biaz
biaz@gatech.edu

1 HOW MY AGENT WORKS

My agent starts at the initial state provided and then conducts a breadth-first search through the entire state space until it arrives at the goal state. If no goal state is found and no more unique states can be explored, it returns an empty list.

1.1 Generating States

A state is defined as the number of sheep and wolves on either riverbank and the placement of the boat. There are five possible successor states: taking one sheep, two sheep, one wolf, two wolves, or a sheep and a wolf.

I decided that my generator should be conservative in generating states, since I didn't want the efficiency of my implementation to suffer from combinatorial explosion. Consequently, a state is never generated if it's illegal, unproductive,[1] or previously visited.

1.2 Testing States

Since my generator does most of the heavy lifting, my agent's tester is quite simple–it simply checks the state to see if all of the sheep and wolves made it to the final riverbank. If so, the state is valid and we can return the path to it.

2 RESULTS

The agent can solve all combinations of sheep and wolves up to 40 sheep, and can find an optimal path for any solvable combination within that space.

The optimality of the agent is guaranteed by the fact that it relies on breadth-first search and does not duplicate states. Since it visits every state possible after each consecutive move, it always finds the goal state in the smallest number of moves–because if the goal state could have been found in fewer moves, the agent

[1] An "unproductive" state is a state where there's only one animal on a riverbank with the boat. The only possible successor is for the animal to come back with the boat, bringing us back to a duplicate state.

would have already visited that state and would have returned the more optimal path.

3 EFFICIENCY

Time complexity is O(S), where S represents the number of possible states for a given input. For investigated combinations of sheep and wolves, the number of states grows linearly, and thus my agent's efficiency is expected to grow linearly. (See the *Appendix* for a deeper dive on this)

On my machine, the agent can solve any combination up to 40 sheep in less than a tenth of a second. Here's what its performance looks like (averaged over 100 runs) plotted against the number of sheep and wolves provided in the initial state:

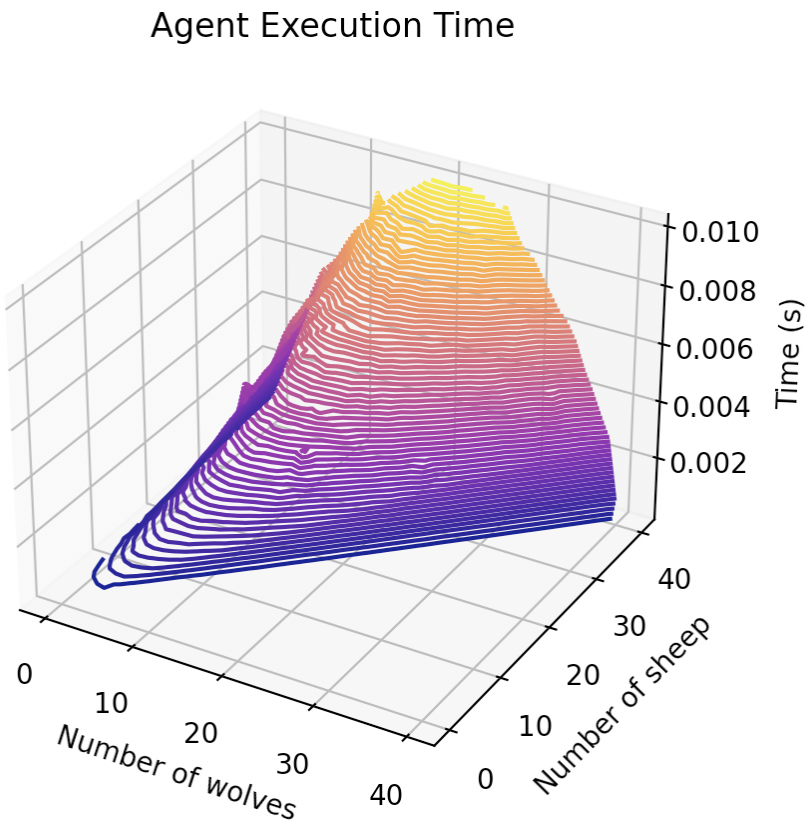

Figure 1—Plot of program execution time against combinations of sheep and wolves (averaged over 100 attempts per combination).

Note that the agent's latency doesn't peak around (40 sheep, 40 wolves) as one might expect–rather, it peaks around (40s, 20w). That's directly related to the number of available states–there's more than ten times as many visited states in the latter (919) than the former (82).

Most of the work on my implementation revolved around improving its efficiency. Note that my first implementation of the agent took more than 4 seconds to solve the combination (5s, 3w) and could not solve (6s, 3w) in any reasonable timeframe.

3.1 State representation

At its most descriptive, a single state contains all of the information the agent could want: how many sheep and wolves are on either riverbank, and where the boat is. However, I realized that, if necessary, one can represent a state as only two integers plus a single bit: the integers tell you how many sheep or wolves are on the *initial* bank, and the bit tells you which side the boat is on. The reason we can get away with storing only one riverbank's information is because we can always deduce how many sheep and wolves are on the other side based on how many we started with.

Since I realized that my breadth-first search implementation may need to store many states at the same time in its queue, I hesitated to store full-fledged State objects in the queue. I asked myself if I could somehow get away with storing a *single number* to represent a state.

This is when I discovered that mathematical set theory had given us the Cantor Pairing function–an invertible function that can map two natural numbers to a single natural number ("Pairing Function", 2021). The two natural numbers to use as input are how many sheep and wolves are on the initial side of the bank. Additionally, we can represent where the boat is by making the mapped output number either positive (for the initial bank) or negative (for the ending bank).

Once I had this function, I could store a state as a single integer, and then, once I needed to, I could invert the integer to derive the state information. This is the key to the efficiency of the agent's implementation. Instead of storing many large State objects, it's simply storing collections of integers, which significantly reduces overhead for breadth-first search (because the queue is so much smaller).

Note that storing as an integer also makes the task of checking for a duplicated state in the state generator *significantly* easier. The previously-visited states can be stored as a hash set, and we can check for a duplicate state by checking for the existence of the integer inside the set, which takes constant time (on average).

3.2 Path re-creation

Breadth-first search is very good at telling us how many steps the optimal path contains, but it doesn't actually trace that path. To recreate the path, we need to "remember" how one state led to another.

My agent initially created states as lists of integers describing the states in order, so a state always came prepended with all of the states that led to it. This was extremely inefficient, since it duplicated lists for every new successor state. So I changed the object storing "previously visited states" from a hash set to a hash *map*, which would map each previously visited state to *the state that led to it.* This way, I could still quickly check if a state had been visited (by checking for the existence of the key in the map), but I could also use the mappings to recreate a path back from the end state to the initial state (each "move" is the difference between states, and the move list is reversed to return the path).

This optimization took my implementation from inexcusably-slow to lightning-fast. Now the breadth-first search could use a queue of simple integers, without needing to prepend all states that lead to a state–instead, it would simply add a mapping of "previous state"->"this state" every time it added a visited state to the hash map.

4 AGENT VS HUMAN

My agent thinks about this problem pretty similarly to how I thought about it when attempting the combination with 3 sheep and 3 wolves.

As a human, I probably perform a bit more depth-first search with some backtracking–but if I truly wanted the *optimal* path, I would go for an approach almost identical to the one outlined. The problem is that this becomes very unmanageable for me as the number of sheep and wolves grows.

4 REFERENCES

1. Pairing Function. (2021, November 27). In Wikipedia. https://en.wikipedia.org/wiki/Pairing_function

5 APPENDIX

We know that my agent's implementation grows linearly in proportion to the number of visited states, because breadth-first search grows linearly in relation to the graph that it operates on. However, the relationship between the number of states and the combinations of sheep and wolves is less clear.

I did some investigation to try and induce a mathematical relationship between the combination of sheep and wolves and the number of valid states that my implementation visits. Here's a summary:

Table 1—Number of Visited States as a function of the number of sheep and wolves as input.

Number of Sheep	Number of Wolves	Number of Visited States
n	n	$2 * n + 2$
n	$n - 1$	$6 * n - 3$
n	0	$2 * n - 3$
n	1^+	$4 * n - 4$
n	$n / 2$	?

For all the investigated relations, the number of states grows linearly. Unfortunately, I was unable to come up with a relation for when the number of wolves is half the number of sheep. I was most interested in this case because that's the combination that seems to lead to the slowest times and the largest number of states (40 sheep and 20 wolves). I conjecture that this relation is linear based on the current investigation and the actual performance on my machine, but I hope to continue investigating and to find out more.