Mini-Project 1: Sheep & Wolves

David Adamashvili
dadamashvili3@gatech.edu

***Abstract*— The Sheep and Wolves problem is a variation of the old folk math problem about crossing a river with a wolf, a goat and a cabbage. In the given problem, we have a river with two banks labeled as **left** and **right**. At first, there are N sheep and M wolves on the left bank (N ≥ M), the right bank is empty. We have a boat on the left bank that can transport either one or two animals at a time. Our goal is to move all the animals to the right bank. However, if at any point the wolves outnumber the sheep on either side of the river, they overpower and eat them. In this paper, I will outline a **Breadth First Search** algorithm for finding the shortest number of moves required to solve the problem.**

1 AGENT DESCRIPTION

My agent looks at the state space of the problem as a graph and uses BFS starting from the initial state to reach the end state. It **generates** states and **tests** them. Since BFS always gives the shortest path to the target in an unweighted graph, the solution is optimal. States that were already visited are kept in a **key-value map**. The key, in this case, is a given state. The states, of course, must be hashable data types. The value is the parent state from which the given (key) state was generated. This map corresponds to the so called **BFS tree**. The root of the tree is the initial state, whose parent state is a predefined null value. The tree helps us with reconstructing the root-to-target path by first finding the target, and then repeatedly moving back up (**backtracking**) to the parent until we reach the root.

In addition to solving the problem, I wrote a tester class that simulates the rules of the problem. My agent sends the optimal moves list to the tester so that the answer can be then validated. Because of this, I knew that my solution was correct and could be sure that it would be accepted.

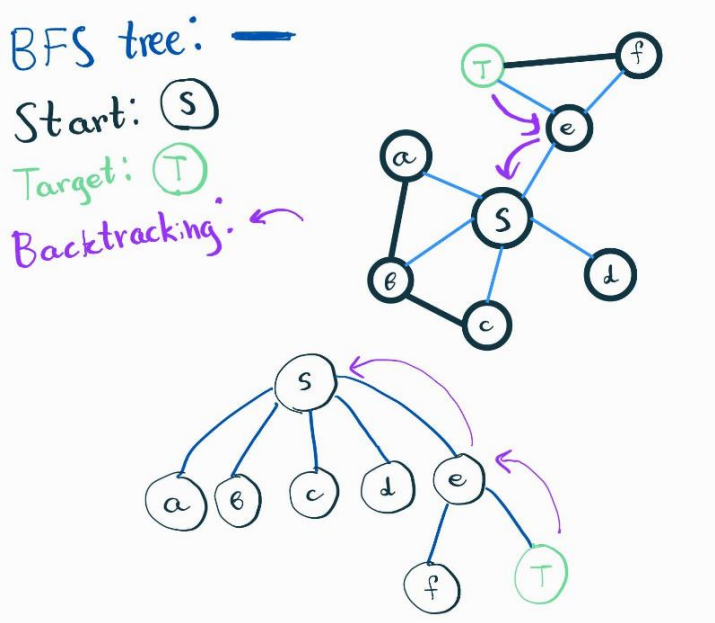
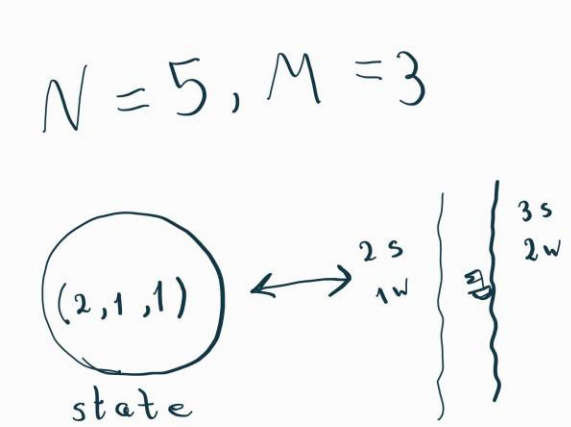


Figure 1 - *BFS tree*

1.1 Generating states

Each state is a 3-tuple of numbers, hence it is hashable. The first two elements of the tuple are the number of sheep on the left bank and the number of wolves on the left bank. The third value corresponds to the location of the boat. If it's -1, the



(left_sheep, left_wolves, boat_side)

right_ sheep == N - left_sheep
right_wolves == M - left_wolves

Figure 2 - *State*

boat is on the left, and if it's 1, the boat is on the right. This information is necessary and sufficient to describe every state, as the remaining animals on the right bank are implicitly known.

Given a state, we can generate 5 new possible states from it. The "branching factor" for this problem is 5. These new states are the following: moving 1 sheep, 1 wolf, 1 sheep and 1 wolf, 2 sheep, and 2 wolves. However, not all of these moves will be valid. First of all, there might not be enough animals left to move, due to which the new state tuple (x, y, 1) might have negative x and y values. And a possible (x, y, -1) tuple might have $x \geq N$ or $y \geq M$ values.

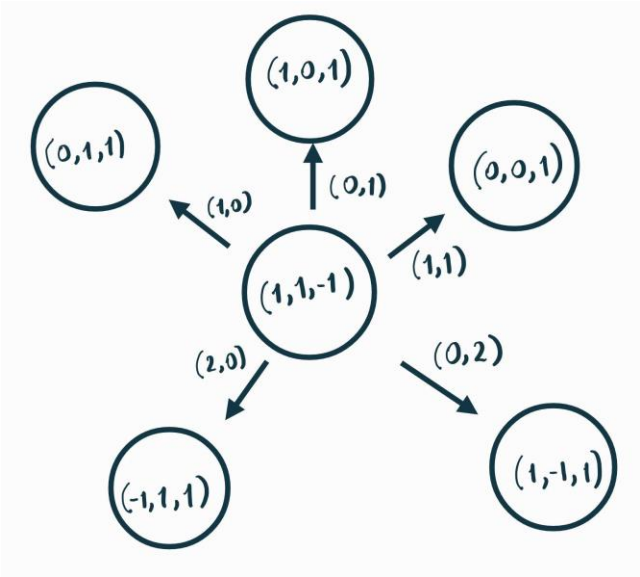


Figure 3 – *Generating States*

1.2 Testing states

The states are tested for **validity** and **redundancy**. As mentioned before, states that have already been visited are kept in a map, and so we can check in O(1) time whether a generated state is useful or not. As for validity, the agent checks that the number of animals on either side of the river is greater than zero (meaning there were enough animals left to move) an5d that the wolves do not outnumber the sheep on both sides.

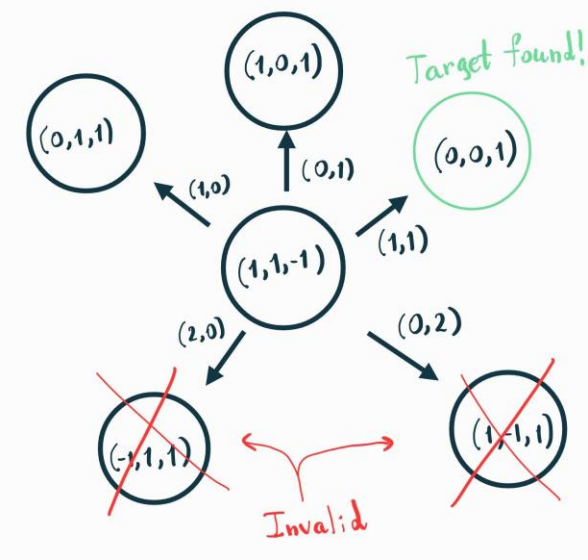


Figure 4 – *Testing states*

2 PERFORMANCE

2.1 Test-case performance

Since my agent always returns the optimal solution, it passed all of the test cases and received a perfect 40/40 score. It should also be noted that I cut down on the amount of memory used by only saving the animals on the left bank in the states. Moreover, the testing done on the states is smart in the sense that invalid and redundant states are not added to the BFS queue. They are tested beforehand.

2.2 Time and space analysis (efficiency)

The following performance analysis might not be the most accurate, it will only give a rough upper bound for the asymptotic performance of the agent. In reality, the graph produced by this problem should be analyzed more thoroughly, and the average depth of the target state in the BFS tree should be calculated. Only the nodes at this depth and above will be used by the agent, and more tight bounds can be produced. However, my intuition tells me that this answer will still be equivalent to the rough estimate up to a constant multiplier.

We know that, for a general graph, the time and space complexities are both O(|V|), where V is the set of nodes in the graph. This is because at worst all the nodes need to be considered, and every level of the BFS tree will be in the BFS queue at the same time. We know that the number of leaf nodes of a complete tree (the worst case) are of the order of the number of nodes.

So, we just need to calculate an upper bound for the number of states for our problem. This is very easy to do: there are N+1 choices for the first tuple element in our state, M+1 choices for the second, and 2 for the third. All in all, $|V| \leq 2(N+1)(M+1)$.

So, all in all, both the time and space complexity of this solution will be **O(N*M).**

3 HUMAN COMPARISON

I, as a human, personally approached this problem a bit differently. Instead of BFS, I used DFS (Depth First Search). What I basically did was, keep making moves until I failed, backtracked, and tried a different move. Sometimes I forgot to check if I had already visited a state and redid my work. And most of the times, the answer I found was not optimal.

To find the optimal path using DFS, we need visit all states. We cannot stop the algorithm whenever we find a solution. Humans, however, instinctively think that their solution must be the best. In fact, I asked my father to solve the (N,M)=(3,3) problem. He found a suboptimal solution, but was convinced that it was impossible to do better. The hubris of a person might lead them down a wrong path. However, a tried and tested algorithm will always return the correct answer. Unless the programmer also made a mistake due to his or her overconfidence and lack of awareness.