Sheeps & Wolves Report (Spring 2022)
CS7637

Samantha Ho
sho67@gatech.edu

1 HOW MY AGENT GENERATES NEW STATES AND TESTS THEM

In designing my agent, I heavily leveraged the concepts shown in the Generate & Test lecture. Since I knew the total number of graph states could grow very quickly, especially with higher input numbers of animals, I needed to implement my agent with both a smart generator and a smart tester to avoid traversing through unproductive and illegal states. This would help with increasing runtime efficiency and avoiding the situation where the agent traverses states in a loop, eventually timing out. Therefore I implemented a class function that first generates all 5 possible new children states given the current state (moving 2 sheep, 2 wolves, 1 sheep and wolf, 1 sheep, or 1 wolf). Then the generator only returns the set of states that are legal, where sheep are never outnumbered by wolves, via passing all possible states through a checkpoint if-statement block. Then, after the generator returns all legal states, the solve method tests whether those states are unproductive (have been visited) via cross referencing with a list of visited states, implemented via a set. It is only after having been filtered through the logic of the generator and tester that any remaining new states will be added to the solve method's dictionary, with the parent state tuple as the key and children state tuple(s) as value(s). New children states are added to a queue to generate their children states. The solve method stops calling the generator if the solution state has been generated, and the dictionary is complete. (In the case that there are no more new child states to be generated and the solution is not found, the solve method returns an empty list). My agent then performs a Breadth-First-Search of the dictionary, implemented via a queue. It finds the shortest path from the starter state to the solution state by adding to the queue the whole path that has been traversed thus far to get to each child state (with a list of tuples representing states traversed on each path), and indexing the last element to find the current state, checking if that is the goal state. If the solution state has been found, it returns that path as a tuple list of states before being converted to moves (Semantic Network transformation labels between each state) and returned as the solution.

2 HOW WELL MY AGENT PERFORMS

If a solution exists, my agent successfully returns the most optimal list of moves for every test case, due to the nature of Breadth-First-Search. Given the implementation choices that were made with the smart generator and tester, and the fact that the agent stops visiting or generating any more states after the solution state has been found, runtime is able to be cut shorter with those decisions. My agent did not fail any of the test cases on Gradescope, random test cases during local testing, and finished executing for every possible combination of sheeps and wolves (where sheep is never outnumbered by wolves) up to 25 sheep and 25 wolves (as is shown in the next section). Therefore it does not seem to struggle particularly with any case.

3 AGENT'S EFFICIENCY & HOW PERFORMANCE CHANGES WITH INPUT

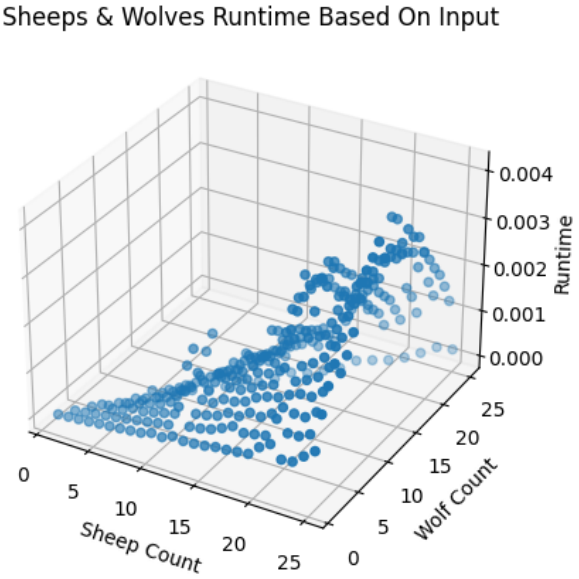

Figure 1—3-D scatterplot of how runtime changes with different inputs of sheeps and wolves, created via the matplotlib and time libraries

I generated this 3-D scatter plot using the matplotlib and time libraries to record the time it takes for the solve method to finish executing for every possible initial state of sheeps and wolves, where sheep are never outnumbered by wolves, up to 25 sheep and 25 wolves. As seen on the plot, in all cases, it takes my agent less

than 0.004 seconds to finish executing, which is very efficient. The plot indicates that my agent takes the longest to solve for a higher number of both animals, the spike in runtime being observed around 15 sheep and 15 wolves onwards. This makes sense, as the more animals to transport, the more moves the agent will naturally have to make to get to the goal state given that the number of animals that can be transported with each move is limited up to 2 for all starting states.

4 AGENT'S ABILITY TO ARRIVE AT ANSWER MORE EFFICIENTLY

I equipped within the smart generator function a hierarchy in generating states that I believed helped my agent arrive at the solution much faster. Since the goal of the problem is to move all animals from the left side to the right side, when generating all 5 new possible states, the generator prioritizes more productive moves over less productive moves. When generating a move from the left side to the right side, it first generates states that will move 2 animals (moving two sheep, two wolves, or one sheep and one wolf) before it generates states that move only one animal (one sheep or one wolf). When moving from the left side to the right side, the generator will first generate possible moves that move only one animal back (one sheep or one wolf) and then those that move two animals back (two sheep, two solves, or one sheep and one wolf). Then these possible new states are appended one by one to a list of all possible new states. The order in which they were appended is hierarchical based on the side that the boat is on. When passing the list of all possible states to the legal checkpoint if-statement block, they will be passed by priority and only appended to the list of legal moves if legal, so the list of legal moves returned from the generator will always list the most productive moves first. Then this list of new move states are cross checked by the testing mechanism in the solve method with the set of visited states to filter out unproductive moves. A set was used to achieve runtime of O(1) when cross referencing, which made the agent run faster as well, ridding the need to traverse through a list container, which would be a big-O complexity of O(n) each time the cross-check is performed, which is computationally expensive as this check must be performed every time the solution state is not found and more children states need to be generated, for every child state before the solution state. Once the cross-check has been performed, the agent will place the resulting priority-based new state(s) into the queue to be explored next. As the more productive states are explored first and their children are generated first, it is more likely that the agent will find the solution state faster. The solve method also stops exploring/generating any more children states in the queue once the

solution state has been found, so by not exhausting all states in the queue, this cuts down on runtime as well. Both the generation of new states and searching for moves from starting to solution state were performed breadth-first.

5 HOW AGENT OPERATES IN COMPARISON TO A HUMAN & MYSELF

The agent is able to solve the problem with much better efficiency and cognitive resources than I could. As a human, it is impossible for me to solve any sets of sheeps and wolves within a few seconds. Cognitively, I do not have perfect working memory as the agent does to perform the unproductive-state filtering efficiently. It is impossible for me to cross-check for unproductive states instantaneously, I would have to refer to the hand drawn graph of states previously visited every time I generate a new state. I am also prone to mathematical, legal, and unproductive state errors while the agent is not. Therefore the agent is much more efficient and error resistant than I am as a human (I used the results generated by the agent during testing to catch errors that I myself made while verifying the results by hand).

In terms of approach, there are similarities between how I would reason through the problem and how my agent reasons through it. Firstly, to make sure that I am not erroneously leaving out legal states, I too tend to draw out all 5 possible new states before I sift through them to determine their legality. It is with this reasoning approach that I designed my smart generator. I too then cross-check these legal states with states that have been traversed previously to determine their productivity before deciding whether to include them in the graph. Cognitively, I instinctively perform a Breadth-First-Search while attempting to solve the problem by hand - I also take a look at all newest children states in the graph to scan for the solution state before making any new children states. When I generate new children states, I also generate for all of the previous parent states before moving on (not a depth-first approach). If one of the states generates the solution as a child state, I also stop generating children states for any of the other states because it would be a waste of time. There are also differences in reasoning approaches between myself and the agent. I did not realize before designing my agent that more productive states should be created and traversed first and will likely yield the solution state sooner - so the agent is equipped with a more optimal cognitive reasoning approach than I am in this sense.