CS76, Fall 2021, Assignment 1, Tracey Mills

Description

Breadth First Search

The BFS algorithm works by considering one node at a time, and appending its unexplored successor nodes to the frontier. It explores all nodes in this manner, stopping when the frontier is empty.
I implemented the frontier with a deque, using append and popleft to add and remove items. This means that the frontier is first in first out, so that the state graph is explored one layer of depth at a time. I used a set for the explored set, so that checking if a node was already explored would be efficient.
I also used a helper function to do backchaining once a goal node was found. This function worked by iteratively recording the parent node of a given node until it reached the root node.

Depth First Search

The DFS algorithm works by considering one node at a time, and recursing on each of its successors. This recursion stops either when a goal node is found or when the specified depth limit is reached. For this problem, I used a depth limit of 100. I implemented path checking by backchaining from the currently considered node, usign the helper backchain function. I kept track of the solution by adding nodes to the path once a successful result was returned, indicating that a goal node had been found. At depth 0, I reversed the path before returning the solution so that the path would be in the correct rather than reverse order.

Iterative Deepening Search

The IDS algorithm works by repeatedly calling DFS with an increasingly larger depth limit. It starts with depth limit zero and goes up to 100. While this iterative deepening expores the early possible states many times, it has an advantage over DFS in that it is optimal because it checks each depth for a solution iteratively, and it has an advantage over BFS in that it does not require memoization and thus uses much less memory.

Evaluation

By printing the visited node states each algorithm is run on the test cases, it can be verified that each search is performed in the correct order. The path to the solution is correctly found in test cases 1 and 3, with no solution for test case 2. By exploring successor states in the order of how many animals are on the end side of the river (how close the state is to the goal), I was able to decrease the number of states visited in the test cases, with the largest effect being for depth first search. Since there are at most 5 successors due to the boat constraints, this sorting takes constant time. For the first test case, BFS explored 13 nodes before finding a 12 node path, while DFS explored 12 nodes to find the same path. IDS explored 161 nodes to find the same path. In test case 2, BFS found there was no solution after 13 nodes, while DFS stopped its search after 24 nodes, and IDS after 2299 nodes. In test case 3, BFS explored 28 nodes to find a solution of length 16, while DFS explored just 16 nodes to find the same solution. IDS explored 1079 nodes to find the same solution.

Discussion

States

For the problem instance 331 in which 3 chickens, 3 foxes, and 1 boat begin on the starting river bank, there are at most 32 possible states, not considering legality. This is because there are between 0 and 3 chickens, between 0 and 3 foxes, and between 0 and 1 boats on the starting river bank at all times. Then, 4 choices x 4 choices x 2 choices gives us 32 states.

Memory Management

Pathchecking DFS saves significant memory over BFS because while BFS stores every node that has been explored, pathchecking stores just those nodes on the current path. A tradeoff is that on some graphs, DFS will take much more time than BFS because many of the first paths fully explored do not contain the goal node. Consider the example below.

In this graph, DFS will traverse each of the long downward pathways, left to right, before finally arriving at the goal node at depth 1 on the far right. In addition, interconnections between these downward paths mean that the same nodes will be visited multiple times, since nodes lie on multiple paths, and pathchecking only weeds out nodes that have been seen on the current path. In contrast, BFS or IDS would find the goal node on the first iteration, and BFS would visit each node only once.
If DFS is instead implemented with memoization, it would also visit each node only once, solving the problem of nodes lying on multiple paths. However, this would then require the algorithm to use at least as much space as BFS, since each node that is explored must be stored, just like in BFS.
When traversing a graph with IDS, it therefore makes sense to use pathchecking DFS rather than memoization. Pathchecking takes more time, since one must compute the path from the node up to the root before checking if it has been seen before, but it uses less memory than memoization. The only reason to do IDS instead of memoization is to decrease memory usage, since it will take at least as long as BFS (both explore one depth at a time, but IDS with recalculations). If memoization is used, IDS will use the same amount of memory as BFS. Thus, it only makes sense to use IDS with pathchecking DFS, since this is what will save memory over BFS.

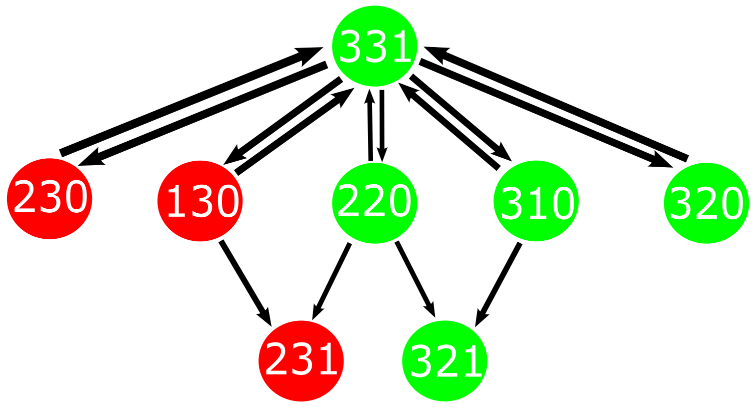

Figure 1: First three iterations of possible states, starting with state 331. Legal states are colored green, illegal states red.

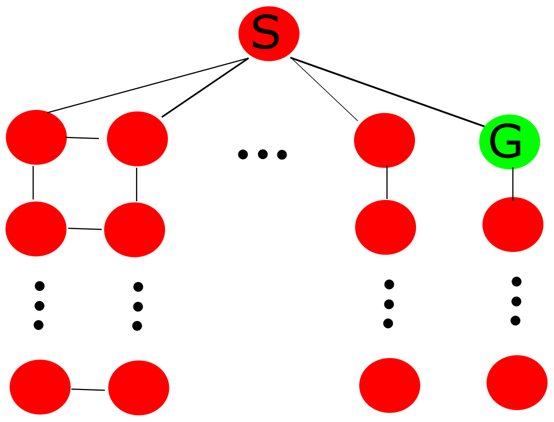

Figure 2: Graph in which successors are explored left to right, with start state S and goal state G.

Lossy Chickens and Foxes

In this problem version, we would add to the state the number of chickens that there are currently. So the 331 problem start state would become 3313. We would also have a class variable E indicating the number of chickens that could be eaten from the start. E would potentially decrease to E' as chickens are eaten over the course of the problem. The search code would remain the same in this version, with just the FoxProblem class having to change. Instead of checking that the number of chickens eaten is 0, the legal moves checker would now check that the number was less than E', and the get successors function would take into account the eating of some chickens, updating the chicken value in the state and the total number of chickens class variable appropriately. Starting with with c chickens, f foxes, and b boats, we can find the total number of states as follows. When E'=E, we have (c+1)(f+1)(b+1) possible states, since the number of chickens, foxes, and boats on the starting side can be 0 to c, f, or b respectively. When E'=E-1, we have (c)(f+1)(b+1) possible states, since the number of the chickens on the starting side can now be at most c-1.
This process continues until E chickens have been eaten, so that E'=0, and there are ((c+1)-E)(f+1)(b+1) possible states. Adding the possible states at each E' value, we get
(c+1)(f+1)(b+1) + (c+1-1)(f+1)(b+1) + (c+1-2)(f+1)(b+1) + ... + (c+1-E)(f+1)(b+1) =
(c+1)(f+1)(b+1) - (0)(f+1)(b+1) + (c+1)(f+1)(b+1) - (1)(f+1)(b+1) + ... + (c+1)(f+1)(b+1) - (E)(f+1)(b+1) =
(c+1)(E+1)(f+1)(b+1) - (0+1+...+E)(f+1)(b+1) =
(c+1)(E+1)(f+1)(b+1) - (E(E+1)/2)(f+1)(b+1)
Given E that is at most c, since it is meaningless to be able to eat more chickens than are present, we get an upper bound of
(c+1)(c+1)(f+1)(b+1) - ((c)(c+1)(f+1)(b+1)/2)
which is in the order of ((c^2)fb)/2 states.

Lossy Implementation

I implemented the lossy version, commented out, in FoxProblem.py.