report.md

Chickens and Foxes

Assignment 1
CS 276, Artificial Intelligence
Fall 2021, Dartmouth College
Spencer Bertsch

This report contains three sections, as seen below:

- Description
- Evaluation
- Responses to Discussion Questions

Description

The problem definition was adapted from the assignment on gradescope.

There are three chickens and three foxes that have found themselves on one side of a river with a single boat. Our job is to model this situation as a graph, then use uninformed search algorithms to find a solution that gets all chickens and foxes safely to the other side of the river.

Starting State:

- (3, 3, 1) - (3 chickens, 3 foxes, and 1 boat)

Constraints:

- Only two animals can fit in the boat at one time.
- In order for the boat to move from bank to bank, one animal must be in the boat to row.
- If there are ever more foxes than chickens on either bank of the river, then the chickens will get eaten.

How do the implemented algorithms work?

The general ideas for the descriptions of these algorithms come from our textbook: Artificial Intelligence, a Modern Approach, and from the lecture notes.

Breadth-First Search

Our implementation of the breadth-first search algorithm begins by checking if the starting node is the goal node. It then examines the top "layer" of the graph or tree, testing each child of the source node to determine if it is the goal node. It then repeats this pattern for the next layer, and so on. This top-down approach is great for finding the optimal solution as long as the path cost doesn't increase as the depth increases. We learned in class that the memory required from vanilla BFS is very large; with a space complexity of O(b^d), the memory explodes for large b or large d.

Depth-First Search

Depth first search starts the same way as breadth first search, checking if the start node is the goal node. It then checks if the first child of the start node is the goal node, but unlike BFS it then continues to check the first children of all current nodes until it either reaches the goal node, or it reaches a terminal or leaf node. Our implementation of DFS uses backtracking in order to save memory, so at this point the DFS algorithm will travel back up the chain, checking each node along the way if it has any unexplored (nodes not currently on the path from start to current state) children. The memory used by DFS is generally much smaller than BFS, but there is a chance that the algorithm will find a non-optimal solution.

Iterative Deepening Search

Iterative deepening search combines a memory usage similar to DFS O(b*d) with the completeness of BFS as long as the number of branches coming from each node (b) is finite. The implementation of this algorithm was pretty straightforward - we just make a function call to DFS and put the call in a loop until we find a solution, then break and return the solution containing the best path.

What design decisions were made?

The design decisions that I made were relatively limited in this assignment because of the extensive skeleton code that was provided, but I will say that I added a few things in order to get the system to work. One thing I added was a new instance variable called 'solved' to the SearchSolution class so that I could always run a test on the returned solution outside the DFS and inside the ids_search function. It took me a while to think of a way to relay information back from the recursive DFS function to an outside function that was making iterative DFS function calls, but by returning an instance of a solution object that had been solved already, I could simply add a boolean instance variable that would tell ids_search if DFS had found the solution or not.

Another design decision that I made that went against one of the guidelines I saw in the comments was to not use an explicit goal_test method in the FoxProblem object. I initially created a method to test whether the current state is the goal state, but the goal test logic can be fit into a single equality test on one line, so I found myself ignoring the goal_test method and eventually deleted it to clean up the code.

Another design decision that I made was to include a print statement in ids_search whenever the depth limit was reached printing the path that was used to reach that depth. Including print statements in loops or recursive functions is generally frowned upon and if I did this at my old job I'm sure some coworkers would wonder why, but I think it's fun to see the exploration of ids_search as it uses DFS to try to find the goal node while being iteratively restricted to larger and larger depths. If anyone grading this assignment can't stand all those prints in the output when you test the ids_search, feel free to comment line line 111 of uninformed search and run again.

How were the problems laid out?

The assignment was presented to use with a strong framework of skeleton code that we were instructed to fill in. I used the file structure that existed in the assignment, and although I made some additions and substitutions for class methods, functions, variables, etc., I still used the same overall structure in the assignment framework. The problem is defined in a FoxProblem class in the FoxProblem.py file. This class is used to dynamically generate the graph which is explored throughout the search.

The actual search algorithms themselves are defined in the uninformed_search.py file. A class representing a SearchSolution object can be found in the SearchSolution.py file; this object is designed to hold the current state, previous state, and other metadata regarding the state of the search. There is driver code in the file foxes.py in which the different search algorithms are called on different FoxProblem objects. In order to see more or less printed output to the console, just comment or uncomment the print statements in foxes.py. I have also created a README which provides instructions on how to pull down this project code locally, install python if users don't already have it, and run the driver code in foxes.py.

Evaluation

Did the implemented algorithms work? How well did they work?

Yes, the implemented algorithms all work as expected.
All algorithms show that there is a solution the problems with starting states of (3,3,1) and (5,4,1), but not to the problem with a starting state of (5,5,1).

As discussed above, I took a few liberties with the print statement in the ids_search, but if you comment all print statements in foxes.py except for a single print, you can see that the outputs of each algorithm are correct. Now onto how well each algorithm performed!

For the problem with a starting state of (5,4,1), the BFS returned a solution with a path length of 16, while the DFS and isd_search returned a solution with a path length of 21. Similarly, for the problem with a start state of (3,3,1), DFS search returned a solution with a path length of 12, while DFS and ids_search returned a solution with a path length of 13. Ignoring time and space complexity and only considering optimality, I would say in this case BFS is more optimal.

Discussions on the time and space complexity of these algorithms and their different flavors (memoizing vs. path checking, etc.) can be found in above and below discussions.

Responses to Discussion Questions

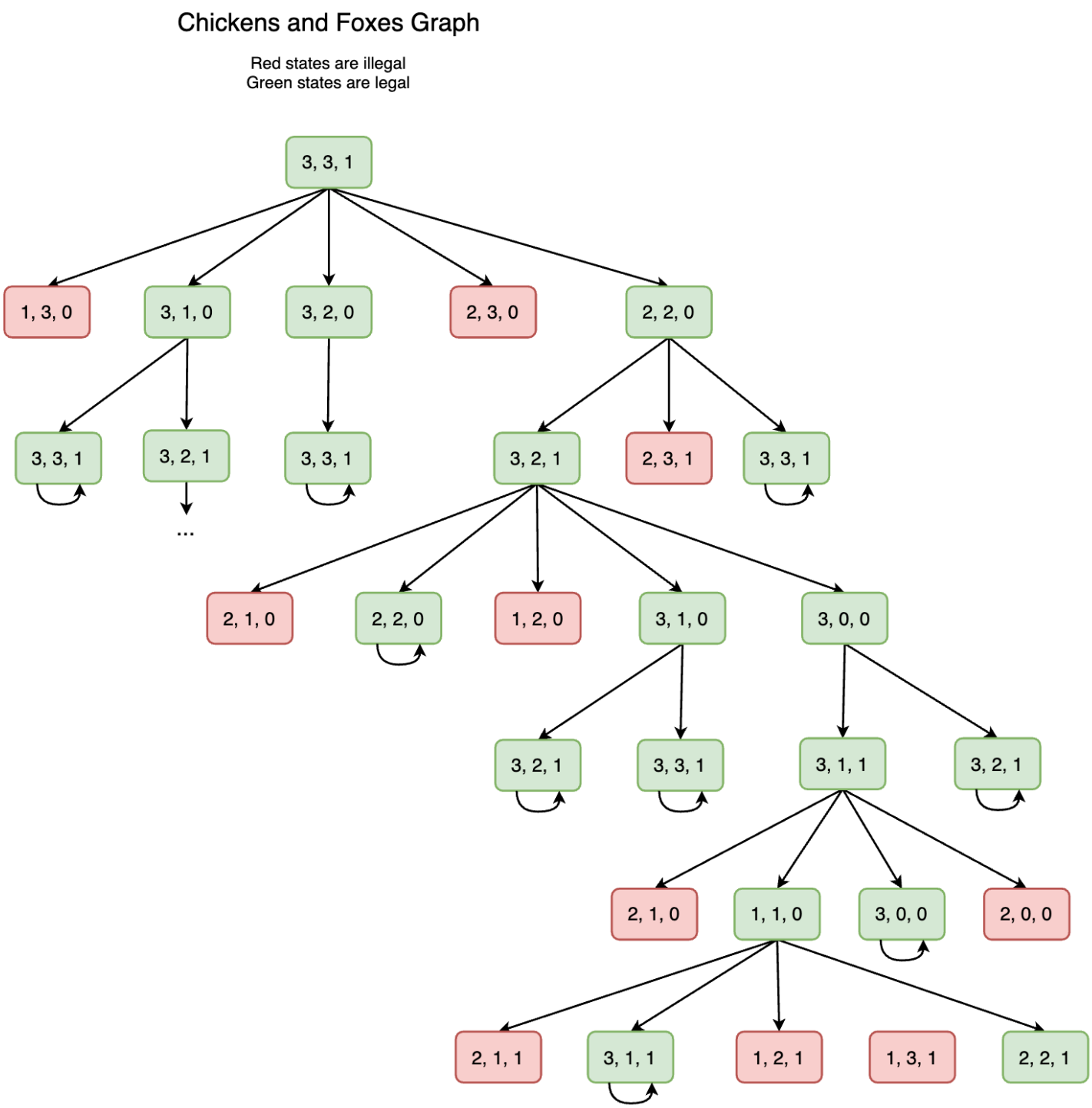
Upper Bound on States

The upper bound on the number of states, given that the starting state is (3, 3, 1) is 32 states including the start state. The number of all possible states -- legal or illegal -- can be thought of as the combinatorial set of all integers with maximum values of 3, 3, and 1. Ignoring the boat/not boat and simply examining the chickens and foxes, we can see that the set of available states is: {(3,3), (3, 2), (3, 1), (3, 0), (2, 3), (2, 2), (2, 1), (2, 0), (1, 3), (1, 2), (1, 1), (1, 0), (0, 3), (0, 2), (0, 1), (0, 0)}.

From this we can conclude that adding a binary variable to the state definition (boat or not boat) would double the number of possible states. See the below figure for a look at each of the 32 unique states.

- **Note:** I also have PDF copies of all figures under the `docs` directory in the ChickenAndFoxes assignment. I couldn't figure out how to embed local PDF files into markdown docs, but if the TAs have some example code for that I will surely use that method in the future. For now please refer to the PNG files in the markdown and the PDF copies with the same names in the `docs` directory if more clarity is needed.

There is a distinction between the states that are "illegal" and states that are "impossible." Illegal states can be generated by the get_successors method, but they are illegal because they break the chickens < foxes rule. Impossible states, on the other hand, are truly impossible to reach and the get_successors method will never return them. (3,3,0) is one such state; I thought this was important to note because I will be excluding the "impossible" states from my diagrams and I will not consider them in my code. (Because they are impossible to reach.)



Above we can see the Chicken & Fox graph up to layer 6. We can see that there exists a path from the root node to the node (2,2,1). I assume that once we arrive at the node (2,2,1), a logical repetition can occur in the pattern of search.

BFS Discussion

Indeed we can see from the result of the BFS that the path found is:

[(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1), (1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]

Showing that the above graph representation is correct. As far as I can tell, the BFS implementation is working correctly. I implemented the pseudocode from the lecture slides and used a double-ended queue in python as the structure that stores the nodes in the frontier.

We can see from running the test code in foxes.py that there is a solution for a start of (3,3,1) and (5,4,1), but not for (5,5,1).

DFS Discussion

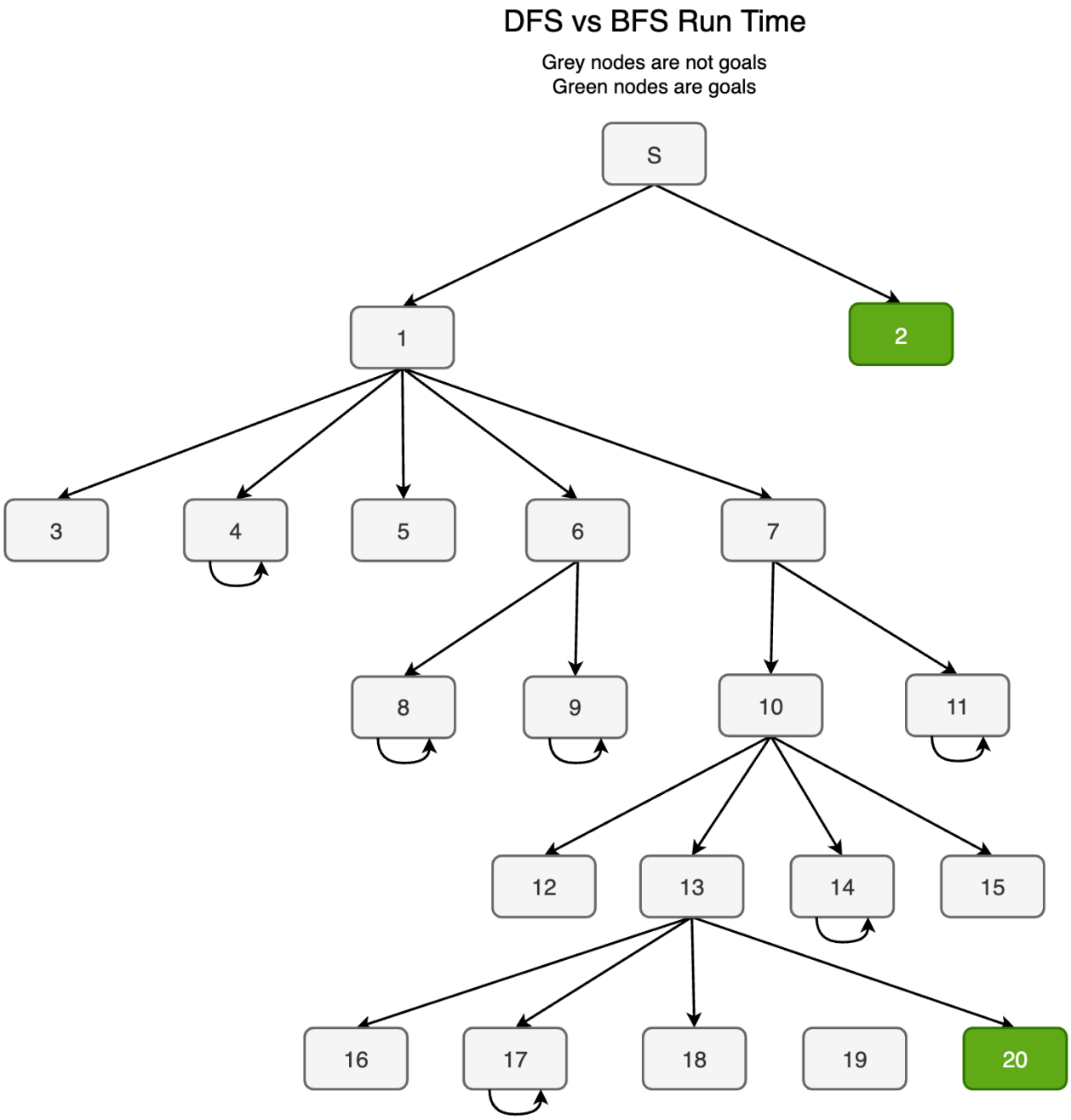
- Q1: Does path-checking depth-first search save significant memory with respect to breadth-first search?

We know from the textbook and from class that in vanilla DFS saves a lot of memory over BFS, especially when b is high (b>10) or d is high (d>10). When either of these (or both) are the case, then DFS will use orders of magnitude less memory than BFS.

A path-checking DFS like the one we implemented for this assignment uses path checking so we are only storing the current path's states in the solution's path instance variable. This saves even more memory than simply keeping a set in memory and filling it with all the nodes that we have seen before. I'm curious if there is an expression that describes the memory saved when using path-checking DFS over vanilla DFS - I will check with the TAs.

- Q2: Draw an example of a graph where path-checking DFS takes much more run-time than breadth-first search.

This is an interesting example. The textbook describes this situation quite well; if there are multiple goals and one goal exists several layers down starting from a left branch of the root node, then DFS will search through many nodes to find a very non-optimal solution. (See below graphic)



This is where Iterative Deepening Depth First Search comes into action! Because vanilla DFS won't find the optimal solution and will often be worse than BFS in terms of time complexity, Iterative Deepening DFS was created to give us the best of both worlds. Using ID-DFS we can begin with a Depth Limited Search with l=0 (pretty boring case), then we can move onto a Depth Limited Search where l=1, and so on. That way we are achieving the space complexity of DFS while also roughly achieving the completeness and optimality of BFS.

- Q3: Does memoizing DFS save significant memory with respect to breadth-first search? Why or why not?

Memoizing DFS will not save significant memory with respect to DFS. From the lecture notes we can see that the space complexity of DFS is O(min(n, b^m)) and the space complexity of graph BFS is O(min(n, b^d)). In this case the max depth of the state space (m) and the depth of the shallowest solution (d) are very similar, so the space complexity of memoizing DFS and graph BFS will be similar.

Iterative Deepening Search Discussion

- Q1: On a graph, would it make sense to use path-checking DFS, or would you prefer memoizing DFS in your iterative deepening search? Consider both time and memory aspects.

The answer to this question depends on what we want to optimize.

If the solution is shallow, and we want to find the solution quickly: Use memoizing DFS for ids_search here because the overall speed will be faster if we store all visited nodes in memory, not just the nodes on the current path.

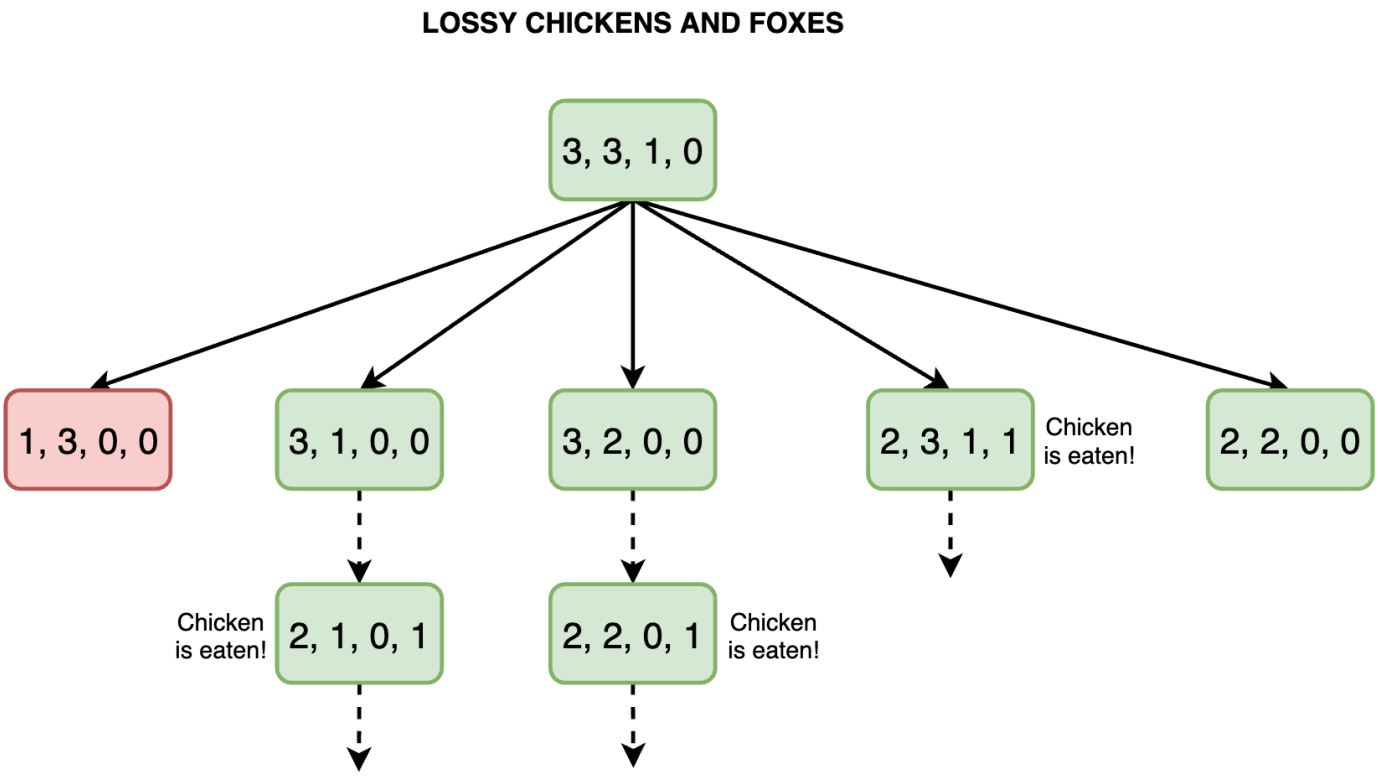
If the solution is deep and you want to save memory: Use path-checkinig DFS for the ids_search because path-checking DFS will save the most memory, especially if the solution is very deep and the ids_search needs to iterate a large number of times to get to the solution.

Lossy chickens and foxes

This is an interesting change to the problem definition! Intuitively we can re-define the starting state from (3,3,1) to (3,3,1,0) where the last number represents the number of chickens that have been eaten. A number E would act as a maximum such that the fourth int in the tuple representing each state could not exceed the maximum (E) number of chickens that can be consumed.

Note: If I were actually programming this, I would probably create actions that could be taken that represented eating the chickens. (See the graph below). This would mean that the graph would start with the maximum number of chickens being consumed right away, then the traditional search would start from there. I'm sure there are many ways to implement this problem, but if I were actually implementing the solution this is how I would do it.

Let's look at a graph representing the first few layers of the Chicken & Fox problem in which E = 1:



In order to update our code to allow this, we would change the way that the state itself is represented by the tuple. Instead of a tuple of length 3, we can use a tuple of length 4 (chickens, foxes, boat, chickens_consumed). If E=1, then the last number can be either 0 or 1, representing the number of chickens that have been eaten. The code changes would need to reflect the new state, so the SearchSolution class would have to change, the method that determines potential new states would have to change, and all checks to see if we are at a goal state would have to change. I believe the actual search algorithms themselves will still work the same way on the new graphs created by adding the extra (chickens_consumed) to the problem.