Lauren Kidman
COSC76: Artificial Intelligence 24F
4 October 2024

Report for PA1: Chickens and Foxes

Table of Contents:

The Assignment....1
Model Implementation:....1
Search Algorithms:....4
BFS....4
DFS....5
IDS....7
Extra – Lossy Chickens and Foxes:....8

The Assignment + *README

The goal of this first programming assignment is to apply knowledge of various search algorithms discussed in class to a specific game-like environment. The environment is as follows:

"Three chickens and three foxes come to the bank of a river as shown in the figure above. They would like to cross to the other side of the river. However, there is one boat. The boat can carry up to two animals at one time, but doesn't row itself -- at least one animal must be in the boat for the boat to move. If at any time there are more foxes than chickens on either side of the river, then those chickens get eaten."

*To run the Chickens and Foxes PA in Python, make sure all four files are in the same folder; run foxes.py to obtain results for various chicken and fox combinations.

Model Implementation:

---

1. Description

The model is implemented through the FoxProblem class, which contains both the start and goal states. It also contains: get_successors(), is_safe(), and goal_test().

At the heart of this programming assignment is the **get_successors()** function–it is used to find out what you can possibly do next from the current state, or, in more graphical language, which nodes you can possibly travel to, given the capabilities of the boat. So, this is the function I decided to complete first as part of the assignment.

I know from the problem statement that the boat can carry up to two animals at a time, and must always have at least one to move; so with that I made a list of possible actions shown in Figure 1 below:

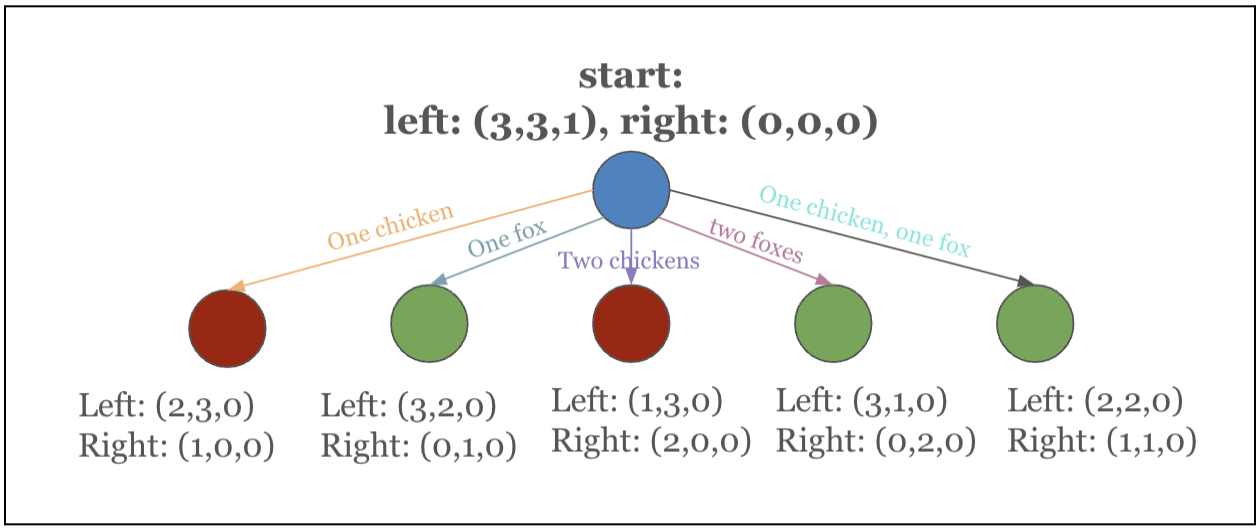

Figure 1: One iteration of next states with the given actions of the problem

The combinations show how one chicken or one fox, two chickens or foxes, or one of each can be moved every rotation of the boat. Based on this alone, the algorithm would be quite simple–you just keep subtracting the number of whatever animal you are bringing over until you reach your goal state. However, what makes it somewhat more tricky is the fact that animals must return BACK from the right side to the left in order for the boat to move, which is shown in Figure 2 below:

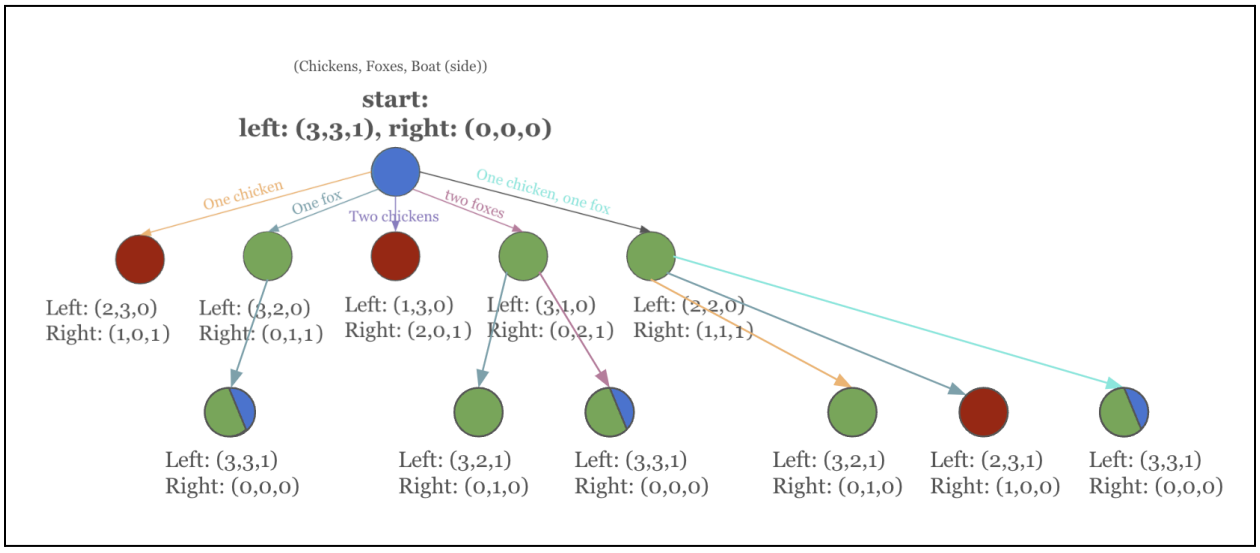

Figure 2: Two iterations of next states with the given actions of the problem

Thus, for my get_successors function, since the calculation for the next state depends on the side of the boat, I split it into two parts with an `if` statement. For each action, if the boat is on the left side, simply subtract the number of animals detailed by the action; if the boat is on the right side, add the animals back to the state, since they must return back to the left side for the boat to move. Once that action is complete, you are left with the new state, which details how many chickens and foxes are now on the left side.

The **is_safe()** function is a helper function to accompany the get_successors() function. Once the calculations in get_successors() are done, you are left with a new state; however, you have to confirm that this new state still abides by the rules of the game. For stylistic purposes/to make the if statement less cluttered with `and` and `or` statements, I split is_safe() into two `if` statements.

> *Check 1*: make sure there are not less chickens than foxes ON EITHER SIDE (**unless there are zero chickens)
>
> *Check 2*: cannot have at any point more chickens or foxes than the game started with, and conversely, cannot have negative numbers of chickens or foxes

The first check is pretty intuitive, but the second check I learned to implement after a few failed tests. Without the second check, since in get_successors() you are adding and subtracting from states without any sort of limit, you could end up with infinitely large numbers of animals or negative numbers. Furthermore, another aspect of the check(which I also forgot to include initially) is the fact that you CAN have less chickens than foxes on either side, as long as the number of chickens equals 0. If I were to forget this step, any moves which resulted in 0 chickens but X number of foxes on either side, which is legal (as there are no chickens to eat), would be flagged as illegal. After the checks, if the state passes is_safe(), it can finally be added onto successors, a storage array.

Finally, **goal_test()** is a simple function which checks if the current_state is the goal, (0,0,0).

2. Evaluation

I feel my implementation of the FoxProblem model works well. It is functional and abides by all rules of the game with minimal state representations, as emphasized in the assignment details. Stylistically, I chose to break up long if statements into multiple checks, which I feel makes the code flow better and more readable. In terms of complexity, it does not waste space keeping track of irrelevant details about the state.

3. Discussion Questions

- *Discussion question: States are either legal, or not legal. First, give an upper bound on the number of states, without considering legality of states. (Hint -- 331 is one state. 231 another, although illegal. Keep counting.) Describe how you got this number.*

This question relates back to the mathematical principle of combinations. We can find the maximum number of states by multiplying the maximum number of possibilities for each individual component of the state (i..e the chickens, foxes, and the boat). For both chickens and foxes, if the starting state is (3,3,1), your options are 0, 1, 2, or 3, i.e. 4 options each. For the boat, since it can either be on the right or left, it is 0 or 1, i.e. 2 options. Thus, **maximum number of states = 4*4*2 = 32.** Following this same logic, for any general start state (x,y,z), the maximum number of states would = (x+1,y+1,z+1).

Search Algorithms:

BFS

1. Description

My BFS follows the pseudocode outlines in Lecture 4 of class. BFS uses a First In, First Out (FIFO) queue data structure, so the frontier is initialized as a queue containing the root node. I also initialized a visited set to keep track of previously-explored nodes and make sure states are not repeated. The algorithm then expands all subsequent *levels* of nodes (i.e. at each level it expands, it adds onto the queue all possible successors), adding these child nodes to the frontier. As long as the frontier exists, each node is popped; when popped, the goal_test() function checks if the current node's state is the goal state. If so, a backchaining helper function is called that traverses through the parents of the goal node to return the final path.

2. Evaluation

Even though the actions of FoxProblem may involve different numbers of animals (e.g., moving 1 chicken, 2 chickens, 1 fox, etc.), each action can be treated as having the same uniform cost–it does not cost any different to move one chicken than to move two foxes. Thus, BFS returns the shortest, and most optimal, path. Looking at the results of foxes.py, BFS is, as expected, more optimal than DFS–while a smaller start state like (3,3,1) returns a solution length of 12 for both algorithms, with start state (5,4,1) BFS

returns a solution length of 16 as opposed to DFS' 18. In general, BFS will always outperform DFS when you have shallow solutions. The only drawback of BFS is that space is defined by your fringe, making it much less memory efficient.

3. Discussion Questions

- *Using a linked list to keep track of which states have been visited would be a poor choice. Why?*

Using a linked list to keep track of visited states is **a poor choice largely due to its searching time complexity**. In a linked list, time complexity is O(n)--you have to traverse through, in a worst case scenario, the entire list to find the goal state. Meanwhile, with a set for example (which is what I used in my code), looking up a state is only O(1). Also, linked lists are generally used because they are quite effective for insertion and deletion; when using search algorithms like BFS and DFS, **we are not "deleting" states from the visited set, so that advantage is essentially worthless**.

DFS

1. Description

My DFS algorithm also is based on the pseudocode outlined in Lecture 4. If there is no starting node parameter, initialize the root node. The search begins, and continues as long as the specified depth limit has not been reached. The algorithm first checks if the current state is the goal state; if so, it returns the solution path. If not, it retrieves the successor states of the current state. Instead of using an explored set like in BFS to assess if a child node has already been visited (memoization), path-checking DFS checks if each successor has been explored by backchaining from the current node. If not, we then recursively call the DFS function, treating this successor as the new current state.

Finally, a crucial part of this implementation is the next_solution.path check after the recursive call. This check is what allows the recursion to "break" after the first valid solution (though not necessarily the most optimal) is found, preventing the exploration of other tree branches and thus making it more memory efficient.

2. Evaluation

My recursive DFS function worked well. It always completed, outputting an answer when there was a path to be found and not getting trapped in an infinite loop (which can sometimes happen to recursive functions). As expected, it did not always return the most optimal solution as DFS returns the "left-most", i.e. first, solution instead of the shortest. Similarly mentioned in the BFS evaluation, for test (5,4,1) its solution length

was 18 compared to BFS' 16. However, the nodes visited output for the DFS was significantly less than the BFS, showcasing how DFS is much more memory efficient–in terms of O complexity, BFS has a space complexity of O(b^d), while DFS space complexity is O(bm).

3. Discussion Questions

- *Does path-checking depth-first search save significant memory with respect to breadth-first search? Draw an example of a graph where path-checking DFS takes much more run-time than breadth-first search; include in your report and discuss.*

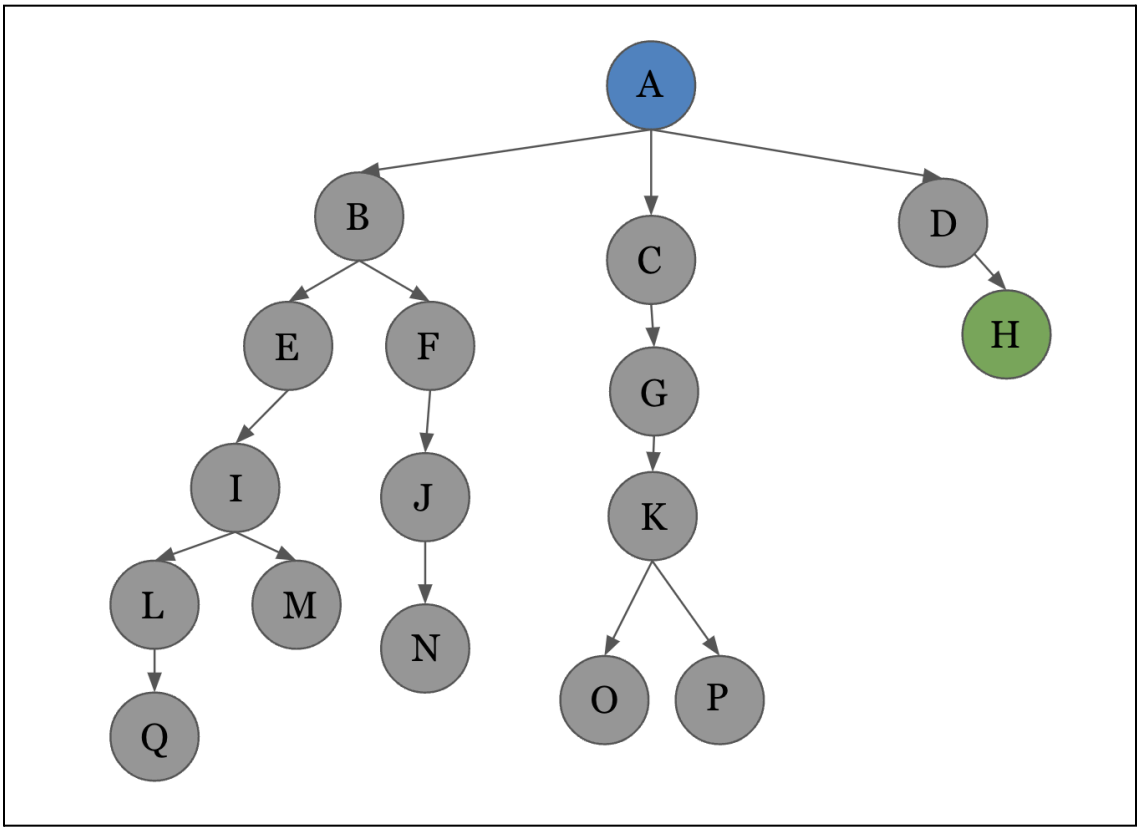

Figure 3: An example graph of when BFS will outperform DFS

An example of when DFS takes much more run-time than BFS is **when you have shallow solutions**. Because of DFS' nature of finding the first/leftmost solution it encounters, it will continue down a large tree of nodes to find a solution, whereas BFS would find the solution in a few simple iterations. **Consider that the run-time of BFS and DFS, respectively, are O(b^d) (where d is the depth of the most optimal solution) and O(b^m) (where m is the max depth of the graph)--in this scenario, d will always be smaller or equal to m.**

- *Does memoizing DFS save significant memory with respect to breadth-first search? Why or why not? As a reminder, there are two styles of depth-first search on a graph. One style, memoizing keeps track of all states that have been visited in some sort of data structure, and makes sure the DFS never visits the same state twice.*

Even though the addition of an explored set (memoizing) increases memory usage of DFS, the nature of DFS as a search algorithm–only tracking the current path and visited nodes–still allows it to save more memory than BFS, which keeps track of all nodes at a certain depth. This is especially true with wide trees, which causes BFS to store a lot of unnecessary data.

IDS

1. Description

In simple terms, IDS (called iterative deepening search) is a built off of DFS. Within a for loop, IDS calls the DFS function–if DFS returns an empty list, that means the goal state has not been found, and the depth limit is increased by 1. This continues until a non-zero length list is returned (i.e. a solution is found), or the maximum depth limit is reached, after which it returns an empty list to signify failure.

2. Evaluation

My IDS function worked as expected–like BFS, it found the shallowest solution in all cases when there was an actual solution to be found (length 12 for (3,3,1) and length 16 for (5,4,1)). Something that initially stuck out was how many more visited nodes appeared in the IDS results; however, this is due to the fact that every iteration calls DFS over again, restarting from the root for each new depth level. From a memory perspective, DFS is still more memory efficient than BFS because it still only stores the children of the branch you are exploring, while BFS stores all expanded children. BFS space complexity is O(b^d) as mentioned, while IDS space complexity is O(b*l), which showcases IDS' memory efficiency. However, the one drawback of IDS is that it takes more time than BFS.

3. Discussion Questions

Discussion questions: 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. (Hint. If it's not better than BFS, just use BFS.)

From a memory perspective, it only makes sense to use path-checking DFS as opposed to memoizing. The whole benefit of using IDS over BFS is the fact that IDS utilizes less memory. Since memoizing causes DFS to use more memory (due to the fact that it needs to keep track of all the visited states), implementing memoizing over path-checking essentially eliminates that space advantage. However, from a time perspective, BFS will always be better than both DFS and IDS, regardless of memoizing or path-checking.

Extra – Lossy Chickens and Foxes:

Every fox knows the saying that you can't make an omelet without breaking a few eggs. What if, in the service of their faith, some chickens were willing to be made into lunch? Let us design a problem where no more than E chickens could be eaten, where E is some constant. What would the state for this problem be? What changes would you have to make to your code to implement a solution? Give an upper bound on the number of possible states for this problem. (You need not implement anything here.)

To implement this additional "chickens eaten" rule, you would not need to make super significant changes to the entirety of the current code. First, you would need to **modify how states are represented**, so you are able at any point to calculate how many chickens left can be eaten. Instead of (chickens, foxes, boat location), now it would be (chickens, foxes, chickens allowed to be eaten (E), boat location). As the PA details emphasize state minimization, this still does that–you can easily calculate how many chickens are left that can be eaten based on the initial number of chickens and E.

Beyond state modification, the primary change lies in calculating the successors. **Is_safe would be modified** so that certain moves on either side where chickens get eaten, depending on E, how many chickens would be eaten in that particular move, and how many chickens have already been eaten, are now legal. I would probably implement this as two additional checks: 1. First check if the number to be eaten is <= E, then 2. check at that particular state how many chickens have already been eaten prior, and if the total eaten value is still <= E after incorporating the state.

Finally, in order to find the maximum number of states, we can use the same logic as in the Model Implementation discussion question. Since the maximum number of states is simply the multiplication of the maximum number of possibilities for each individual attribute, it would look quite similar–now just with a fourth attribute. Instead of (x+1,y+1,z+1) → (x+1)*(y+1)*(z+1), now it would be **(x+1,y+1,E+1, z+1) → (x+1)*(y+1)*(E+1)*(z+1)**