Collaboration Statement

Author: Andrew Koulogeorge Ideas contained in this report were discussed with Alan Sun and Eric Richardson

Initial Discussion and Introduction

Each state in the problem is defined as an ordered triplet (f,c,b) where:

- $f :=$ Number of Foxes on the left side of the river
- $c :=$ Number of Chickens on the left side of the river
- $b :=$ -1 if the boat is on the left side and 1 if the boat is on the right side

Number of States: Let F be the total number of foxes, C be the total number of chickens and n be the number of states in our game. * $f \in \{0, 1, ..., F\}$ * $c \in \{0, 1, ..., C\}$ * $b \in \{-1, 1\}$

In the standard problem where $F = C = 3$, $n \leq (4)(4)(2) = 32$. More generally, $n \leq (F + 1)(C + 1)(2)$. I got this number using the general product principle- counting the number of possible states is the same as counting the number of valid sequences where we have 4 options for the first entry, 4 options for the second entry, and 2 options for the third.

Clearly, not all of these states will be legal. For example:

- $(3, 2, -1)$ is not a legal state since there are more foxes than chickens on the left side
- $(3, 3, 1)$ is not legal state since the boat cannot drive itself

In the diagram **states.pdf**, nodes that are colored green are legal states and nodes colored red are illegal. Note our state graph is undirected since we can always repeat an action and return to the orginal state. The edges are labeled with what animals are moving across the river between states. From the diagram, we see that: * For the inital state $(3, 3 - 1)$, there are a total of 5 actions that can be taken but only 3 of them are legal. * Moving just a chicken or chickens to start the game automatically loses since the chickens we left behind would get eaten! * Moving 1 fox across to start the game is a bad move! Since the boat cannot drive itself, returning to the start state would be the only legal next move.

Code Design

Building The Model

The *FoxProblem* class has been altered to accept different boat sizes, custom values for F and C, as well as different start states besides just (F,C,-1).

- **get_successors**: Returns a set *valid_out_neighbors* of possible valid next states by considering all pairs of foxes and chickens such that at least one fox or chicken is in the boat and there is no more than *max_boat_size* animals in the boat at one time. I modeled the boat as -1 when on the left side since we are subtracting from the state vector when the boat moves from left to right and adding to the state vector when it moves from right to left. A cleaner calculation can be made that does not require an extra if statement by multipling *delta_fox* and *delta_chicken* by the value of *boat* and toggling the sign of *boat* for the next state.
- **is_valid**: Before adding this new state to our *valid_out_neighbors* set, we call **is_valid** to ensure the new number of foxes and chickens is not outside the boundaries and that no chicken got eaten!
- **is_goal** returns True if the input state is the goal state and False otherwise.

BFS

It would not be a good idea to store the nodes we have visited during BFS in a Linked List because of how slow it is to check memership of an element in a Linked List. It could take up to O(n) time looking over each element of the Linked List. With a HashSet, we can check membership in constant time. When we are traversing the graph and we come to a new node, we want to be able to check quickly if we have seen this node before or not!

Given an instance of FoxProblem: * Create an instance of SearchSolution that will hold the set of nodes *current_seen* that the search visits during BFS. * Use a deque to keep the nodes we visit arranged in a FIFO manner and begin the search with the *start_state* wrapped in a *SearchNode* object. * If the start node we were given is not a legal state, return the empty SearchSolution object. * Otherwise, we perform a standard BFS: pop the node from the front of the deque. For the neighbors we have not visited, increment the count of nodes we have seen and add new nodes to the deque. * When we create a new SearchNode to be added to the deque, we pass parent node as an instance variable to the new node to keep track of which node found it. * When we find the goal state, we call **backchain** on the goal state node to backtrack through its parent nodes and return a list containing them in order starting from the start node. * If this endstate is never found, we keep popping from the deque. Eventually the queue will become empty when we visit all the nodes in the graph, the while loop will terminate, and we will return an empty SearchSolution object.

Memoizing DFS

Memoizing DFS does not save significant amounts of memory compared to BFS. Given a state space, let d equal the depth of an optimal solution and b equal the number of children/sucsessors of a given node. BFS holds every node that it

visits in memory. If every node of the graph is visited by BFS, then its space complexity = $O(b^d)$. If the state space has a depth of d, then DFS will visit and store each node in the graph and thus will also have space complexity = $O(b^d)$. If the search space is infinite, then memoizing DFS is not guarenteed to termiante at all (it is incomplete) and it could overflow the memory storage of your computer!

Path-Checking DFS

In the SearchSolution object, we store several instance variables that help us implement DFS recursively. *dfs_found* is a boolean variable that is set to True when we find the solution and helps the recursive function return the SearchSolution object. We store the nodes on our current path in *path* and also store the elements of path in the set *current_seen* to enable fast lookup time for membership checking if another node is already in our current path (helps us prevent cycles!). *nodes_visited* is a counter for the number of nodes we visit during the search.

Given an instance of FoxProblem, a SearchNode, a depth limit and a SearchSolution: * Check if we are at the start of the search by checking if node is None. If we are, create the root node and a solution object. * Add this node to our working path (represented as both a set for quick lookup time and as an array so that we can return the path) and increment the number of nodes we have seen in our search. * Base Case: Check if we are at the goal state. If we are, we set the *dfs_found* variable to be True and we return the instance of SearchSolution. * Recursive Case: If the length of our path is less than or equal to our depth limit, we loop over all legal neighbors not currently in our path. For each of these states, we construct a new SearchNode and call DFS on this new search node. If we ever find the goal state, the instance variable *dfs_found* of *solution* will be True and we will return the solution object. Checking for *dfs_found* after recursing forces all of the nodes on the path containing the goal node to return *solution* and the algorithm will terminate. * When we get to a node that has no neighbors that are not in our path or we have exceeded our depth limit, we remove that node from our current path and return the *solution* instance to its parent node. This is the bubble up part of the algorithm- we return back the instance of *solution* that no longer holds the child node in its path.

Path-checking DFS does save significant amounts of memory with respect to BFS. For path-checking DFS, we only need to keep a set of the nodes that are currently on our path which takes O(l) space where l is the max depth of the tree. In our implemation we store 2 copies of these nodes since we keep track of a list and a set but this does not change the memory usage asymptotically –> O(l + l) = O(l) which is much better than O(bˆd)! Still note that if the state space was infinite it is not guarantee that this algorithm will ever termiante so in an infinite state space the memory usage could overlow.

dfs_vs_bfs.pdf shows a graph with a complete subgraph but a very shallow goal state. For this graph, BFS would find the goal node very quickly because the goal state is only 2 edges away from the start state. Path-checking DFS, on the otherhand, would get tangled up in the complete subgraph on the left side for a very long time. Recall that while keeping track of the current path prevents the search from going in a cycle, it does not remember globally which nodes it has been to. As a result, it will explore each of the nodes in the complete subgraph several times as it will consider all paths within the complete subgraph.

Iterative Deepening Search

It is true that for a tree, DFS does not require as much memory as BFS since DFS will only ever hold in memory the length of the longest path in the tree. This memory usage, as we have said, is O(l) where l is the depth of the tree. In a graph, however, its possible that the longest path will contain all of the nodes of the graph and thus DFS could take O(n) memory just like BFS.

In my implementation of IDS: * Create an instance of SearchSolution and the start node to pass along to DFS * If the starting node isnt a valid state return an empty SearchSolution object * Otherwise, loop over all values between [0,depth_limit] and run DFS from the start node with a max search depth of *depth*. In my implementation, I consider the root node to be depth 0. IDS for a depth limit of k will run DFS for depths up to and including depth k ($k+1$ levels). * The return type of DFS is the SearchSolution Class. There are two cases to consider: * If no goal state is found with the given depth limit, all nodes in the path and current_seen instance variables of SearchSolution will be empty. The check for *solution.path* will return False and the loop will call DFS again with a larger depth limit. Note that we send in the same instance of SearchSolution back into the DFS call. This enables us to keep track of the total *nodes_visited* since this value is never reset. * If a goal state is found, the *path* instance variable in SearchSolution will contain the path from the start node to the goal node and thus will not be None and we will return the solution before any more iterations of the loop is called. This solution object will hold the total number of nodes we visited during the various DFS searches

Memoizing DFS has the benefit of preventing our search from revisiting a node. It trades off extra memory storage for a faster algorithm than PC DFS. For PC DFS, it trades off a longer run time for less required memory as it only stores the nodes in its current path (as opposed to all of the nodes it visits). If there are multiple ways to get to a partiuclar node, PC DFS will visit this node several times. Since we can revisit a node several times, PC DFS is much slower than both BFS and Memoizing DFS but it does same memory.

If we used Memoizing DFS for our IDS algorithm, asymptotically it would be the same as BFS- all nodes would be visited and stored in memory resulting in O(bˆd) space and time complexity where b is the branching factor and d is the

depth of the goal. In pracitce, however, BFS will be faster than IDS since it will never do the repeated work on the earlier level nodes that IDS does. Thus, I would go with BFS!

Lossy Chickens and Foxes

- Problem State: Now that its possible for the number of living chickens to decrease during the game, we no longer can reconstruct how many chickens are on the right side from the starting number of chickens and the number of chickens on the left side. To fix this, we could add another variable *chickens eaten* to our state that represents the number of chickens that have been eaten so far. We will be able to construct the number of chickens on the right side by right_chickens = starting_chickens - left_chickens - chickens_eaten. Our state is now represented by a tuple of length 4: (F, C, B, E) where the total number of states is upper bounded by $(F+1)(C+1)(2)(E+1)$
- Implementation Notes: Changes would have to be made to the FoxesProblem class:
  - Add an instance variable *chickens_eaten* and make the state be represented by (f, c, b, e) instead of (f, c, b)
  - Starting state = $(F, C, -1, 0)$
  - Goal states = $(0, 0, 1, e), e \in \{0, 1, ..., E\}$
  - Update *get_successors* to consider changes to *chickens_eaten*. Note that the max number of chickens that could be eaten in a given turn equals the size of the boat. This is because at most boat_size chickens can be moved from one side to another and we know the state we are coming from is legal so no side already has more foxes than
  - Update *is_valid* state to include more complex logic if there is a case when the condition left_fox > left_chicken > 0 or right_fox > right_chicken > 0 is broken. We are going to be updating the state if a fox gets to eat a chicken, so we want our *is_valid* method to now return states. Lets consider 3 cases:
    - If a chicken can be eaten: Instead of returning False right away and rejecting this state, will want to compute the number of chickens that are going to be eaten. Let this variable be y. If chickens_eaten + y > E, we return None since this would be over our threshold of number of chickens we are allowed to eat and the state would be invalid. If chickens_eaten + y <= E, subtract y from the chicken count on the current side the boat is on to model them being eaten add y to the count of *chickens_eaten*. We will then return this updated state.
    - If a chicken cannot be eaten but the state is otherwise valid, return the same state that we input.
    - If a node is invalid we return None. None now represents the state

being invalid. We can make a quick update to our *get_successors* method to check if the return type is None and if its not then we add the state to legal neighbors.

If at any point during the game there are more total foxes than living chickens, it becomes impossible to win the game because the goal state itself is no longer valid; if all of the foxes and chickens were able to get to the other side, all of the chickens would get eaten! So this setup only make sense when we have more starting chickens than foxes, but as we will see in the next section, the game is always solveable when we start with more chickens than foxes. It would be cool to analyze how adding this eating condition changes the shortest path to a solution when total chickens > total foxes, but I have already spent too much time on this pset :)

Theory & Experimenting with theory.py

The classical problem of 3 chickens, 3 foxes and a boat that fits 2 is interesting. It is natrual to ask the following questions: how do the solutions change as a function of the number of chickens? What about the number of foxes? What about the boat size?

In the theory.py file, I wrote some code to test how the the solutions change as we increase the number of foxes and chickens as well as the boat size. When executing the file, we see the following results: * With a boat size of 2, the problem *appears* to be solvable for $F = C \in \{1, 2, 3\}$ but impossible for all numbers larger than 3 * With a boat size of 3, the problem *appears* to be solvable for $F = C \in \{1, 2, 3, 4, 5\}$ but impossible for all numbers larger than 5 * With a boat size of 4, the problem *appears* to be always solvable $\forall\ F = C = x,\ x > 0$ such that $x \in N$ * When there is at least one more chicken than there are foxes, problem *appears* to be always solvable

Motivated by these computational experiments, I will provide theoretical justification for claims 3 and 4. In both of these proofs I assume the total number of foxes and chickens in the game is equal to the starting number of foxes and chickens.

Claim: With a boat size of at least 4 and a starting state of $(x, x, -1)$, $\exists$ a path from $(x, x, -1) \rightarrow (0, 0, 1)\ \forall\ x > 0$ such that $x \in N$

- **Proof by Induction on x**:
  - Base Case: For $x = 1$ and $x = 2$, we see that the we can reach the goal state $(0, 0, 1)$ from $(x, x, -1)$ in a single move by carrying all of the foxes and the chickens in the boat at the same time. Since the size of the boat ≥ 4, this is a legal transition. Note that to begin, there was an equal number of foxes and chickens on the right side (0 of both, but equal)
  - Inductive Hypothesis: Assume that $\exists$ a path from $(k, k, -1) \rightarrow (0, 0, 1)$

such that $k \in N$ where there are an equal number of foxes and chickens on the right side. We wish to show that this implies that $\exists$ a path from $(k+1, k+1, -1) \rightarrow (0, 0, 1)$.
  - Inductive Step: Given the starting node $(k+1, k+1, -1)$, carry 2 chickens and 2 foxes to the otherside to begin. This next state $(k-1, k-1, 1)$ has 2 chickens and 2 foxes on the right side. Since the number of foxes equals the number of chickens on both sides, this is a legal state. Now, carry 1 fox and 1 chicken from the right side to left. This state is $(k, k, -1)$ with an equal number of foxes and chickens on the right side. Thus, by the inductive hypothesis, $\exists$ a path from $(k+1, k+1, -1) \rightarrow (0, 0, 1)$
  - By Mathematical Induction, we have shown that with a boat size of at least 4 and a starting state of $(x, x, -1)$, $\exists$ a path from $(x, x, -1) \rightarrow (0, 0, 1)$ $\forall$ $x > 0$ such that $x \in N$.

Note that for any boat size larger than 4, we could just not fill the boat up all the way so the same argument applies for all larger boats!

Claim: With a boat size of at least 2 and a starting state of $(x-1, x, -1)$, $\exists$ a path from $(x-1, x, -1) \rightarrow (0, 0, 1)$ $\forall$ $x > 1$ such that $x \in N$

- **Proof by Induction on x**:
  - Base Case: For $x = 2$, our starting state is $(1, 2, -1)$. Note that we have an equal number of chickens and foxes on the right side to start. Even though they are both 0, this is important for the proof. Moving 1 fox and 1 chicken together to the other side results in $(0, 1, 1)$. Then move the fox back over to state $(1, 1, -1)$. From here, we can move the fox and the chicken together to get to the goal state of $(0, 0, -1)$. Thus, we have construced a sequence of actions that results in a solution from $(x-1, x, -1) \rightarrow (0, 0, 1)$ for $x = 2$
  - Inductive Hypothesis: Assume that $\exists$ a path from $(k-1, k, -1) \rightarrow (0, 0, 1)$ such that $k \in N$ where there are an equal number of foxes and chickens on the right side. We wish to show that this implies that $\exists$ a path from $(k, k+1, -1) \rightarrow (0, 0, 1)$.
  - Inductive Step: Given the starting node $(k, k+1, -1)$ there are an equal number of foxes and chickens on the right side:
    - Carry 1 fox and 1 chicken to the right side $\rightarrow (k-1, k, 1)$. Since we are adding one of each to the right side, the property of the number of foxes being less than or equal to the number of chickens is maintained.
    - Carry 1 fox back to the left $\rightarrow (k, k, -1)$. Since we started with one more chicken on the left side, there is now an equal number of foxes and chickens on the left and one more chicken than foxes on the right.
    - Carry 1 fox and 1 chicken to the right side $\rightarrow (k-1, k-1, 1)$. Same logic applies from step one. Still one more chicken on the right side.

    - Carry 1 chicken to the left side $\rightarrow (k-1, k, -1)$. There is now one more chicken than foxes on the left side and an equal number of foxes and chickens on the right.

  Thus, we reduced the problem of having k foxes and $k+1$ chickens on the left side with the boat and an equal number of foxes and chickens on the right side to $k-1$ foxes and k chickens on the left side with the boat and also an equal number of foxes and chickens on the right side. Thus, by the inductive hypothesis $\exists$ a path from $(k, k+1, -1) \rightarrow (0, 0, 1)$
- By Mathematical Induction, we have shown that for a boat size of at least 2 and a starting state of $(x-1, x, -1)$, $\exists$ a path from $(x-1, x, -1) \rightarrow (0, 0, 1)$ $\forall$ $x > 1$ such that $x \in N$

Same argument from the last proof also applies here. Since for any boat size larger than 2, we could just not fill the boat up all the way so the proof applies for all larger boats!

Note that this argument quickly extends to the case where we have several more chickens than foxes in our starting state. If that was ever the case, we could just move chickens over to the right side until we have 1 more chicken than fox on the left side and the right side being all chickens. No state would ever be illegal because there are never more foxes than chickens on the right side during our algorithm. We can see this displayed as well in the theory.py file!