PA 1 - Chicken & Foxes

Julien Blanchet | CS 276 (Artificial Intelligence) | Prof. Li.

September 23, 2020

Introduction

In this report, we aim to develop a program that can find solutions to the "Chicken & Foxes" problem. You start on one side of a river with foxes and chickens and one boat. Each time the boat crosses the river, it *must* carry *one or two* animals. At any given time, there must not be more foxes than chickens on either side of the river (or else they'd get eaten).

State Representation: To represent the state of this problem, we'll use the notation `(F,C,B)`, where `F`, `C`, and `B` represent the number of foxes, chickens, and boats (respectively) on the *starting* side of the river. Note that `B` must either be zero or one - this problem doesn't make sense with multiple boats.

The number of possible states depends on the initial configuration - i.e., how many foxes and chickens you start with. Given the initial configuration of `(3,3,1)`, there are between zero and four foxes and sheep on the initial side of the river, and there are two places the boat could be, so there are at most $4 \cdot 4 \cdot 2 = 32$ possible states. Note that this is an upper bound - some of these states are illegal. In general, there may be no more than $(F+1) \cdot (C+1) \cdot (B+1)$ states. The following figure illustrates the possible states given the `(3,3,1)` initial configuration - illegal states are colored in red.

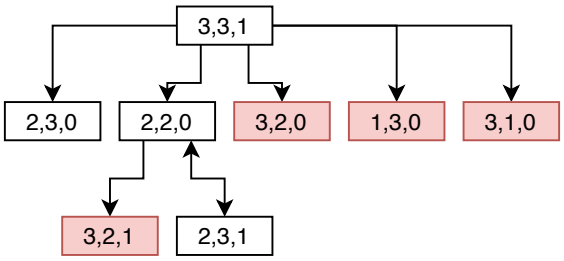

Figure 1: State Graph Figure

Technical Architecture

Code Design: I implemented the fox problem and search algorithms in accordance to the architecture suggested by the sample code. The three search algorithms-BFS, DFS, and IDS- take in a problem object which must implement the `get_successors()` and `is_goal(state)` methods and contain the `start_state` property. Search solutions are returned in a `SearchSolution` object, which contains both the path and a count of the nodes that have been

visited.

Building the Model: The specifics of the chickens-and-foxes problem are implemented in the `FoxProblem` class. The code follows the logic specificed in the problem definition pretty closely. The starting state must have exactly one boat, as the behavior for multiple boats is undefined.

Testing: In addition, I created a test file `test_pa1.py`, which performs basic validation of the search algorithms. To run these tests, simply run `pytest` in this project directory - the test file and tests within it will be automatically discovered.

Breadth-First Search

Breadth-First Search is implemented in the `bfs_search` method. To keep track of explored nodes and allow for later reconstruction of the path to the goal node, I store nodes in the `backpointer` dictionary as they're added to the frontier. The python dictionary is implemented as a hashtable, which provides constant time lookups - ideal for checking whether a node is explored or not. The frontier is implememented using a `deque`, which provides constant time performance for the operations used by BFS (popping from the front of the queue and appending to the end).

I elected to forgo using a `SearchNode` wrapper object for BFS, as it proved unnecessary. One requirement of using the state directly in the backpointer dictionary, however, is that the state be hashable. Python tuples are hashable when their constituant elements are hashable, which is the case for the `FoxProblem`. Other search problems using this code would need to ensure their states are hashable.

Like the rest of the code, BFS is tested in `test_pa1.py`, which validates that shortest path and explored nodes on the (3,3,1) problem and a few others.

Memoizing DFS

Discussion question. Does memoizing dfs save significant memory with respect to breadth-first search? Why or why not?

I don't expect that a memoizing DFS would save significant memory with respect to breadth-first search. Both BFS and memoizing DFS have $\mathcal{O}(V)$ space requirements, as they may have to store an entry in the visited set / memoizing table (respectively) for each vertex in the graph.

Path-checking depth-first search

Unlike BFS, my DFS implementation uses a `SearchNode` wrapper object to make it easier to keep track of parent nodes along the path and preventing us from having to add additional parameters to the recursive call. I identified two

base cases for my implementation - either the depth limit has been reached, in which no path is found, or the goal has been reached, in which the path is recoreded. There is also a third implicit base case - if there are no successors to the current node, no path is found.

Discussion: Does path-checking depth-first search save significant memory with respect to breadth-first search? Show and discuss an example where path-checking dfs takes much more run-time than breadth-first search

I expect path-checking depth-first search to save significant memory with respect to breadth-first search because path-checking DFS only ever needs to store the nodes along the current path, rather than (in the case of BFS) all nodes explored at this point. I would expect that the larger the branching factor and the larger the graph, the more memory that path-checking DFS would save over BFS.

However, though path-checking DFS may save memory, it may be slower, particularly in graphs where the DFS search may be caught up searching in deep subtrees when the goal node is in fact close to the starting node, as illustrated in the following figure.

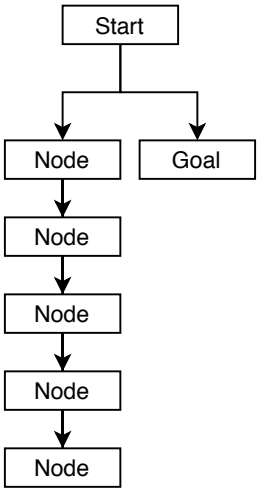

Figure 2: Example where path-checking dfs takes more run-time than dfs

Iterative deepening search

My implementation of IDS is exceedingly straightforward and does not warrent further description here. I tested IDS using the same suite of tests for DFS and BFS - and the three methods all found solutions when one was possible. However, IDS consistently explored many more nodes than the other methods - for example, in the (5,5,1) problem, IDS explored 409 nodes whereas DFS explored 18 and BFS explored 25. IDS in general appears to be hugely inefficent

with runtime, as it performs many unnecessary (in retrospect) DFS searches until it explores at sufficient depth.

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?

Using memoizing DFS for IDS would save runtime as compared to path-checking DFS, but at the cost of memory complexity. In particular, using memoizing DFS in conjunction with IDS is essentially an alternate form BFS and shares the space complexity of having to store the entire graph in memory (and may produce suboptimal paths). Using path-checking DFS for IDS is clearly inefficient with time complexity, but may be useful in cases where memory is limited and space complexity is the overriding constraint.

Discussion: *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.

The state of this problem could be represented as `(c,f,b,e)`, where `c`, `f`, and `b` are the number of chickens, foxes, and boats on the starting side of the river, and `e` is the number of chickens that may still be eaten. In this case, an upper bound on the number of states is $(c+1)\cdot(f+1)\cdot(b+1)\cdot(e+1)$. To implement a solution to this problem, we'd need to make the following changes:

- `get_successors()`: add states where chickens have been eaten to the list of possible states.
- `is_legal()`: add check that no more than the allowable number of chickens have been eaten.
- The state representation would be a 4-part tuple throughout, as described above.
- `is_goal()` doesn't need to change - in particular, it should *only* validate the equality of `c`, `f`, and `b` between the goal state and the test state.