Foxes & Chickens

Carter Kruse (September 18, 2023)

Instructions

Note: This problem is traditionally known as the ''missionaries and cannibals'' problem. The missionaries are replaced with chickens, and the cannibals with foxes, to create a less problematic setting.

Three chickens and three foxes come to the bank of a river. They would like to cross to the other side of the river. 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. (These are talking animals, from Narnia, I guess.) If at any time there are more foxes than chickens on either side of the river, then those chickens get eaten. In this assignment, we will implement algorithms that will make plans to get everyone across the river in time for lunch, without any chickens becoming lunch.

The solutions are implemented using Python 3, and the report is written using Markdown, in a file called `report.md`. If there is trouble installing the required software, or there are questions regarding the assignment itself, the questions are posted on Ed Discussion.

*The design is general-purpose; the search algorithm solves any appropriate search problem, not just chickens and foxes. Other students and TAs should be able to run the code, so there is no use of any external libraries.

The Model: States & Actions

The starting point modeling many problems is to ask what the state of the system might be. A solution will be a connected sequence of states from the start state to the goal state; each pair of states is linked by an action.

The state of the system is the information that would be needed to fully describe the current situation at some point during the task to someone else, assuming that they already know the rules of the game. For the chickens and foxes problem, the state could be described as how many chickens and foxes are on each side of the river, and on which side of the river the boat is.

Constants are not part of the state. The size of the boat (it holds two), or the total number of chickens and foxes (three each), are not part of the state, since these are constants that don't change as a

result of the actions chosen. They are part of the rules of the game. That doesn't mean that the rules can't be changed – but they are changed in the problem definition, rather than in state elements.

Minimal state representations are often best. If one fox, one chicken, and one boat are on the starting bank, where are the other chickens and foxes? Wait wait, don't tell me! They had better be on the other bank of the river, assuming nothing... untoward... has happened. Since I can compute where the others are, I don't have to keep track of their locations as part of the state. I would describe this state using the tuple `(1, 1, 1)`: one chicken, one fox, and one boat, on the starting side. If I need to know how many of each are on the other side, I can use subtraction. Using this representation, the initial state is `(3, 3, 1)`.

Actions link states. Given a state of `(3, 3, 1)`, the action of moving one chicken, one fox, and one boat to the other side changes the state to `(2, 2, 0)`. Given a particular state, we'll need to consider which actions are legal. Actions that can be carried out from that state, and don't lead to a state where someone is eaten.

States are nodes in a graph; actions are edges. There is an underlying graph of all possible legal states, and there are edges between them. The algorithms that are written will implicityly search this graph. However, we do not generate the graph ahead of time, but rather write methods that, given a state, generate legal actions and possible successor states, based on the current state and the rules of the game.

The Write-Up

Each of the following sections of the assignment is described in a section of this report. For example, the first section of the report is labeled "Introduction" and contains the work related to the assignment section "Initial Discussion & Introduction". The report is written using Markdown in `report.md`. Following this, a PDF form is generate (using `pandoc` or a similar program). (If there is any trouble with this step, just leave it in Markdown.) Both files are included in the submission, as well as all figures (as a PDF).

Introduction (Initial Discussion)

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.

Given the set-up of the problem, there is an upper bound on the number of states, without considering legality of states, given as `(#Chickens + 1) * (#Foxes + 1) * (#Boats + 1)`. This is because any given state may contain `(0, 1, 2, ..., #Chickens)` and `(0, 1, 2, ..., #Foxes)` and `(0, ...,`

`#Boats)`, as it is representative of the leftmost side of the right. In the simplest case, there are 3 chickens, 3 foxes, and a single boat, thus the upper bound on the number of states is 32.

In other words, considering the case where there are `C` chickens and `F` foxes with a single boat, the upper bound on the number of states is `2CF`.

Using a drawing program such as `inkscape`, part of the graph of states is drawn for the `(3, 3, 1)` case, including at least the first state, all actions from that state, and all actions form those first-reached states.

We show which of these states are legal and which are not (by using the color of the nodes). The legal states are encoded using the color green while those that are illegal are encoded using the color red. This figure is included in the report, and is labeled as `Figure_1.pdf`.

To display the graph, the state is written within a node, and edges are labeled according to the transition/move that connects nodes/states. The format for the nodes is `(C, F, B)`, where `C` represents the number of chickens and `F` represents the number of foxes on the left side of the river. As indicated previously, it is relatively straightforward to determine the number of chickens and foxes on the right side of the river, given the starting state.

Nodes that are illegal do not have any children extending from them, as they represent states that we aim to avoid in finding a path from the start state to the goal state. If there are no black lines from a node, that indicates that there are no further children of the node. For example, the node `(3, 2, 0)` encodes the case where a single fox is brought across the river in the first move, in which case the only following move is to bring the fox back, as the boat does not row itself.

Finally, the black lines are labeled with the appropriate transition, though it is important to consider whethere the boat is on the left or right side of the river. If it is on the left side, the transition is multiplied by -1 before being added to a given state, otherwise, it is simply added to a given state. All of the transitions are reversible, indicating that there is no direction to the edges (they are undirected edges).

Note: It is nicest to save the figure as a PDF, which is a vector graphic format, not as a pixel-based format such as PNG, JPEG, GIF, TIFF, or BMP. Vector graphics can be scaled and will still look good in a PDF report, while bitmaps (pixel-based) will not scale well.

Code Design

You should read the provided code now to try to get a sense of how things mgiht work. It can be hard to understand someone else's code; in parallel, think about how you would write the code yourself. It may be helpful to do a little coding to understand the problem better, and then to look at the provided code again. Ultimately, the code must fit the provided design.

This process involved looking at the code, correcting syntactical errors, and developing and understanding of the structure of the code, along with how it is expected to work.

Building The Model

Now we are ready to write and test the code that implements the model. (We will write the search algorithms in the next section.) `FoxesProblem.py`` is the starting point. We need to represent key information about the problem here, including information about the starting state, some sort of method to get successors, and a way to test if a state is the goal state, as well as other things.

The constructor allows us to keep track of the start state and the goal state, according to the problem statement. As indicated, other aspects of the problem are added, including the total number of chickens and the total number of foxes.

```
class FoxesProblem:
    def __init__(self, start_state = (3, 3, 1)):
        self.start_state = start_state
        self.goal_state = (0, 0, 0)

        # The total number of chickens/foxes is determined based on the start state.
        self.chickens = self.start_state[0]
        self.foxes = self.start_state[1]
```

The `get_successors` method is responsible for determining the appropriate states that follow, given an initial state. We start by considering if the boat is on one side versus the other side of the river. This `if` statement reduces the complexity of the algorithm (in terms of runtime/memory), by reducing it in half.

The `for` statement considers all of the possible moves, and then determines if the state/position is safe, according to the constraints of the game. If so, we append the new value to the list.

Further, we introduce randomization of the moves to ensure that the algorithm functions for all possible combinations.

```
# Determine the succcessor states for a given state.
def get_successors(self, state):
    # Initialize the list of successors, which starts out as empty.
    successors_list = []

    # Enumerate the possible moves for the simple problem, according to which side the bc
    if state[2] == 0:
        possible_moves = [(0, 1, 1), (1, 0, 1), (0, 2, 1), (2, 0, 1), (1, 1, 1)]
    else:
```

```
        possible_moves = [(0, -1, -1), (-1, 0, -1), (0, -2, -1), (-2, 0, -1), (-1, -1, -1
    # Introduce randomness to allow for consecutive iterations to be independent of a det
    random.shuffle(possible_moves)

    # Cycle through the list of possible moves.
    for move in possible_moves:
        # Creating the next state, according to the appropriate transition.
        next_state = (state[0] + move[0], state[1] + move[1], state[2] + move[2])

        # Check if a given state (determined by an initial state and transition) is safe.
        if self.is_safe(next_state):
            successors_list.append(next_state)

    return successors_list
```

The following helper function is used to determine if a given state is safe, and thus is added to the list. For this problem, there are a couple of situations that are important to check. First, we must consider if there are too many chickens/foxes in total (and on either side fo the river). We consider that the initial state is safe automatically, which allows us to break the if statement down into a much simpler version.

Next, we have to check if the eating constraints are satisfied, that is, we must ensure that there are more chickens than foxes on either side of the river at any given time, or in the case where there are more foxes than chickens, then there are no chickens. This prevents the chickens from being eaten.

```
# Test if a given state is safe (before adding to successor list).
def is_safe(self, state):
    # Capacity Constraints: Check that there are not too many chickens/foxes.
    if state[0] < 0 or state[0] > self.chickens or state[1] < 0 or state[1] > self.foxes:
        return False

    # We do not need to consider checking the number of boats, because it is never out of

    # Eating Constraints: Check that none of the chickens are eaten (outnumbered) by the
    # We consider the case where there are no chickens on a given side, which is okay.
    if state[0] < state[1] and state[0] != 0:
        return False
    if self.chickens - state[0] < self.foxes - state[1] and self.chickens - state[0] != 0
        return False

    return True
```

Finally, there is a method used to check if the current state is the goal state, which returns a boolean value.

```
# Test if the state is at the goal.
def is_goal_state(self, state):
    if state == self.goal_state:
        return True
    return False
```

The to_str statement was already written, but it is simply responsible for displaying what the problem is, along with the start state. This fully defines the problem, given the problem set-up. We know the goal state, and we do not need to consider the transitions, as these are already considered in previous functions.

```
def __str__(self):
    string =  "Foxes & Chickens Problem: " + str(self.start_state)
    return string
```

Breadth-First Search

Take a look at `uninformed_search.py` *. The first goal here is to implement the breadth-first search. The breadth first search should work properly on a graph (not explore the same state more than once), and should use appropriate data structures to make the search fast. (Using a linked list to keep track of which states have been visited would be a poor choice. Why?) There is not implementation of data structures such as hash tables, linked lists, as Python provides just about anything needed. The* `set` *and* `deque` *classes are helpful.*

As indicated, using a linked list to keep track of which states have been visited would be a poor choice, as you would have to iterate through the entirety of the linked list, following the pointers, to see if a given state has been visited. This is rather inefficient in terms of time/space.

The BFS algorithm for a graph is based on the following pseudocode:

```
BFS (Graph)
    frontier = new queue
    pack start_state into a node
    add node to frontier

    explored = new set
    add start_state to explored

    while frontier is not empty:
        get current_node from the frontier
        get current_state from current_node
```

```
        if current_state is the goal:
            backchain from current_node and return solution

        for each child of current_state:
            if child not in explored:
                add child to explored
                pack child state into a node, with backpointer to current_node
                add the node to the frontier

    return failure
```

The breadth-first search algorithm is implemented appropriately, indicating that the algorithm works properly on a graph (and does not explore the same state more than once). Further, the appropriate data structures are used to make the search fast. To reiterate, using a linked list (as opposed to a set) to keep track of which states have been visited would be a poor choice because of the need to iterate through the linked list every single time you wanted to check if something was already in the explored set of the states/nodes.

There is no need to implement data strucutres such as hash tables or linked lists, as Python provides just about anything required. The set class is used to create the explored set in Python, while the deque class is used to create a queue in Python, specifically with the `popleft` method. The time and space complexity of BFS is `O(B^D)`, where `B` is the branching factor and `D` is the depth.

According to (GeeksForGeeks)[https://www.geeksforgeeks.org/deque-in-python/], "Deque (Doubly Ended Queue) in Python is implemented using the module "collections". Deque is preferred over a list in the cases where we need quicker append and pop operations from both the ends of the container, as deque provides an `O(1)` time complexity for append and pop operations as compared to a list that provides `O(n)` time complexity."

Backchaining is also implemented, which extracts the path from the graph after the search has found the goal node. A link to a parent node is stored in each node except the starting node.

As indicated, the backchaining method is implemented and is responsible for extracting the path from the graph after the search has found the goal node. A link to the parent node is stored in each node except the starting node, which allows us to determine the backchaining.

The `append` method is used when creating the path, as it is faster than the `insert` method.

```
# Backchaining
def construct_solution_path(current_node):
    # Backtrack from the current node (presumably the goal node) to construct the solutic
    path = []
```

```
    # Cycle through until the root (start node) is reached.
    while current_node is not None:
        # Append the 'state' to the path. The 'append' function is faster than 'insert'.
        path.append(current_node.state)

        # Update the pointer of the current node.
        current_node = current_node.parent

    # We should reverse the path at the end, as we use the 'append' function.
    return path[::-1]
```

Test the code, and discuss the code in the report. A good test would at the very least check the breadth-first search with respect to the nodes drawn in the introduction.

The file `foxes.py` is provided which provides basic test code. It makes sense to comment out some of this file so you can test BFS in isolation.

The code, when tested, provides an appropriate solution.

Memoizing Depth-First Search

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.

The implementation of memoizing DFS is not given, though the following is discussed in the report.

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

As we talked about in class, memoizing DFS does *NOT* save significant memory with respect to breadth-first search. The result of adding an explored `set` to DFS makes the memory size `O(n)`, as we are essentially considering keeping track of all the nodes in the graph. This assumes that the goal state is not always located shortly after the DFS algorithm starts, which is the case.

In this sense, the purpose of DFS is to limit the number of nodes that we consider and store in memory, while BFS considers all of the "leaf" nodes (for a given depth). DFS dives deep into a tree/graph, only considering the branching parts. By keeping track of the visited nodes, we tend to keep track of most of the nodes within the graph, thus when we traverse using DFS, memoizing results in storing most of the nodes in the graph.

Further, it is important to consider the case where the graph is not a tree, as it has loops. This will be taken into account in the next section. Memoizing memory is not costly for BFS, as the frontier is

already big `O(n)`. This is because we consider the leaves, which is (approximately) half the nodes for a tree with a branching factor of 2. However, for DFS, memoizing is rather expensive, as the frontier we are considering is relatively small `O(D*B)`.

Path-Checking Depth-First Search

A different style of depth-first search keeps track only of states on the path currently being explored, and makes sure that they are not visited again. This ensures that at least the current path does not have any loops.

In other words, to avoid building the complete explored set for DFS while eliminating loops, we may use path-checking. According to the slides from class, path-checkign involves the following: keep track of the states on the current path only, and do not loop.

This does not eliminate redundant paths. We consider instead a large number of loop-free paths in the graph: `O(B*D)`

The implementation of a recursive path-checking DFS is written. The base cases and recursive case are clearly labelled in the code.

As requested, the implementation of the path-checking DFS is recursive.

Discussion: 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.

Typically, it depends on the graph (given the specific case) to determine if path-checking depth-first search saves significnat memory with respect to breadth-first search. Given an arbitrary problem, DFS may be more memory efficient than BFS, and vice versa.

A graph where path-checking DFS takes much more run-time than BFS is shown as `Figure_2.pdf`. In the graph, a DFS algorithm that follows the leftmost branch will not reach the goal state until all of the nodes are covered. Contrastingly, a BFS algorithm will reach the goal in after a single increase in the depth level.

This highlights that BFS is typically efficient for finding a goal state/node that is relatively shallow in a given tree/graph.

Iterative Deepening Search

BFS returns a shortest path, and can require a lot of memory. DFS on a tree certainly does not require much memory, if the tree is not very deep. (How about on a graph? You just discussed this above.)

As indicated previously, DFS may require a significant amount of memory if there are cycles within a graph, which we can account for by using path-checking.

Can we design an algorithm that returns a shortest path but doesn't use much memory? Here's a simple idea. Limit the depth of the depth-first search, by passing the current depth to each recursive call, and exiting the search if the depth is higher than some pre-determined number. This is called depth-limited search.

Run depth-limited search of depth 0. If it finds a path (well, the start is the goal, so this is not an interesting case), then this is the optimal path. If not, then there is no path of length 0 to the goal. Run DFS to depth 1. If it finds a path, then this is optimal, since there was no path of length 0. If not, run DFS (depth 2). Continue. This is called iterative deepening search, and it is easy to implement – just a loop around a depth-limited DFS. Implement it. Discuss your code in the report if there is anything worth mentioning that you'd like us to know about. Make sure to test the code and verify that it gives a shortest path.

The implementation of IDS (iterative deepending search) works as expected, and produces the shortest path. In this sense, it allows for a way to determine the shortest path without requiring the same amount of memory as BFS.

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.)

On a graph, it may make sense to use path-checking DFS, though there are some cases where memoizing DFS in the iterative deepening search would make more sense. DFS on a tree certainly does not require much memory, if the tree is not very deep.

When using path-checking with iterative deepending, the time complexity is typically similar, while the space complexity is less. However, in the case where space complexity is not an issue, we may aim to memoize, as this allows for a faster run-time.

As highlighted, the use of each depends on the specific case, as you would not have to loop back through all of the nodes in memoizing DFS, though this may not be most memory efficient.

In comparing BFS with memoizing DFS, BFS will almost always be faster, so this is typically what should be used in cases where we do not care about memory.

It always will depend on the shape of the graph, so gaining an understanding of this in advance is the way to go.

Discussion Question: Lossy Chickens & 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 community, 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.)

In the case where some chickens were willing to be made into lunch, we may design a problem where no more than `E` chickens chould be eaten, where `E` is some constant. In this case, we have an advanced state, which includes an extra component encoding the number of chickens eaten. This value may be appended to the current tuple representing the chickens/foxes on each side of the river, along with the boat. Thus, we have a starting state of `(3, 3, 1, 0)`. The ending value of the tuple does not go beyond `E`, which encodes the maximum number of chickens able to be eaten.

In this way, we greatly expand the graph search for this particular problem, as the encoding becomes more complex, and thus we have a greater number of successor states.

Now, this actually becomes more interesting of a problem to solve considering that we have several "goal" states to consider. If we aim to reduce this to a graph problem, there is a significant level of complexity that is introduced into the problem, making it slightly more difficult to find an answer, and set up the problem.

To implement a solution, we would need to change the way we consider the goal states, along with the way we consider successors. Further, the upper bound on the number of possible states for this problem would be `(#Chickens + 1) * (#Foxes + 1) * (#Boats + 1) * (#Eaten + 1)`.