0/7 Questions Answered

PA1 - Chickens and foxes

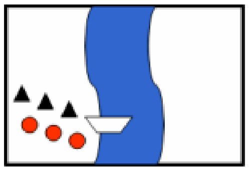
Q1 The problem
7.5 Points

==Objective==

Your goal for this assignment is to model a search problem and solve it with some uninformed search algorithms that we have seen in class.

==Getting started==

General description of the problem



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.

In this assignment, you'll implement algorithms that will make plans to get everyone across the river in time for lunch, without any chickens becoming lunch, and you'll write a report describing your implementation and results.

Provided code

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

Please take a look at this note about the coding style which you should keep in mind as you write code for all programming assignments:
https://canvas.dartmouth.edu/courses/67629/pages/notes-on-coding-for-the-assignments

Optional:

All programming will have an optional report section. These reports will be taken into account at the end of the course for giving bonus points and citations.

The report can be written using anything as long you submit a PDF. I personally recommend using Markdown, but you don't have to. If you are interested in learning Markdown, please take a look at the following guide which provides the basic syntax: https://www.markdownguide.org/basic-syntax/

==Required tasks==

You must implement your solutions in **Python 3**. If you have trouble installing the required software, or have questions regarding the assignment itself, please post your questions first on Ed Discussions.

You should choose a design that is general-purpose; your search algorithm should solve *any* appropriate search problem, not just chickens and foxes. It's important that other students and TAs are able to run your code, so you should not use any external libraries, other than possibly a recent version of `cs1lib.py` (not needed for this assignment).

Please include your Python files (included anything needed to run the code, such as `cs1lib.py`) in **one** zip file for submission on Gradescope.

Below are the specific tasks with some context that will help your implementation and discussion in the report.

1. Implement the model: states and actions

The starting point for 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. 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 you write will implicitly search this graph. However, you will 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.

Now you are ready to write and test the code that implements the model. (You'll write the search algorithms in the next section.) `FoxProblem.py` is your starting point. You'll need to represent key information about the problem here, including information about the starting state, some sort of `get_successors method`, and a `goal_test`, as well as other things.

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.

Use a drawing program such as inkscape to draw part of the graph of states, including at least the first state, all actions from that state, and all actions from those first-reached states. Show which of these states are legal and which aren't (for example, by using the color of the nodes). Include and describe this figure in your report.

Note on figures: It is nicest save your 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 your pdf report, while bitmaps (pixel-based) will not scale well.

2. Implement breadth-first search

Take a look at `uninformed_search.py`. Your first job here is to implement the breadth-first search. Your breadth-first search should work properly on a graph (not explore the same state more than once), and you 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?) You do not need to implement data structures such as hash tables, linked lists, or others yourself; Python provides just about anything you should need. I like the `set` and `deque` classes from Python.

You'll also need to implement backchaining, which will extract the path from the graph after the search has found the goal node. I stored a link to a parent node in each node except the starting node.

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

I have provided a file `foxes.py` that has some basic test code. It makes sense to comment out some of this file so you can test BFS in isolation.

3. Implement path-checking depth-first search

Write your implementation of a recursive path-checking DFS, i.e., a depth-first search that 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.

Make sure the base cases and recursive case are clearly labelled in your code. You will lose most of the credit for this problem if your implementation of DFS is not recursive.

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.
- 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.

4. Implement iterative deepening search

BFS returns a shortest path, 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.)

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 your code and verify that it gives a shortest path. In my tests, I used a starting state of 5 chickens, 4 foxes, and 1 boat on the starting bank, rather than (3, 3, 1), which is after all not very interesting.

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.)

==Optional: The writeup==

Write about any or all of the issues below.

1. a report to discuss your implementation, which contains the following information:
   a. Header with the code of the class, the term, and year, the assignment number, and your name.
   b. A short typewritten report describing your results. The report should contain three main sections:
   (a) Description: How do your implemented algorithms work? What design decisions did you make? How you laid out the problems?
   (b) Evaluation: Do your implemented algorithms actually work? How well? If it doesn’t work, can you tell why not? What partial successes did you have that deserve partial credit? Include a comparison of running time results.
   (c) Responses to discussion questions that are included within the points in "Required tasks" and the following point:
   - 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.)

Q1.1 Submission and grading
0 Points

Your code should be efficient and well-designed, with excellent style, formatting, and comments. It should be brief if possible; a long chain of if statements is not good code if it can be replaced with something terser and easier to read. Follow good style guidelines, although we will not be strict here. For example, Python style guidelines suggest that there should be spaces after '#', spaces around operators like '+'.
Use whitespace! Group lines of code into logical units that are no more than 3-8 lines long using blank lines. Factor code out into functions. Make sure to claim authorship of your code in a comment at the top of the file (include date as well.)

Here is the rubric:

- FoxProblem model: 1.5
- BFS implementation: 1.5
- Recursive path-checking DFS implementation: 1.5
- IDS implementation: 1.5
- Overall code style and efficiency: 1.5

Submit only one zip file containing all the code and the pdf for the optional report. Please feel free to add comments in text box (e.g., use of a late pass).

Please select file(s) Select file(s)

Save Answer

Q1.2 FoxProblem model (placeholder for grading)
1.5 Points

Save Answer

Q1.3 BFS implementation (placeholder for grading)
1.5 Points

Save Answer

Q1.4 Recursive path-checking DFS implementation (placeholder for grading)
1.5 Points

Save Answer

Q1.5 IDS implementation (placeholder for grading)
1.5 Points

Save Answer

Q1.6 Overall code style and efficiency (placeholder for grading)
1.5 Points

Save Answer

Q2 Bonus: Extensions beyond the basic requirements

0 Points

Please feel free to go beyond the required tasks. For example, you could implement the lossy chicken and foxes variation or the memoizing DFS. As written in the syllabus (please check the syllabus for complete information), the extra credit can help later on to resolve borderline letter grades. Don't hesitate to reach out in Ed Discussions to ask questions.

Same as the optional report, these extensions will be taken into account at the end of the course for giving bonus points and citations.

Save Answer

Save All Answers

Submit & View Submission