Foxes & Chickens - Extension

Carter Kruse (September 18, 2023)

The following implementations are designed to enhance the problem, thus meriting "extra credit" as defined by the outlines of the assignment. By goign above and beyond to consider a problem with different set-ups (including multiple end goal states, different boat capacities, etc), we consider a greater level of complexity for the problem.

Due to the structure of our code, we are able to apply the general BFS, DFS, and IDS algorithms to our specific problems by encoding the information into the functions as appropriate.

Extension 1

This extension allows for modification of the carrying capacity of the boat, thus enabling more than 2 animals to cross the river at any given time. Regardless, we maintain the constraints limiting the number of foxes in comparison to the number of chickens, even those that are on the boat.

The main modification is to the `get_successors` method, which is responsible for determining the valid next states, given a current state. This is as follows:

```
# Using a clever for loop, we enumerate all of the possible moves for the advanced proble
# according to which side the boat is on. This allows for ANY capacity constraint.
possible_moves = []
proposed_moves = [(c, f, 1) for c in range(self.chickens + 1) for f in range(self.foxes +

# Cycle through the proposed moves to see what is valid.
for move in proposed_moves:
    # Check if the capacity constraint (of the boat) are met, and at least one animal is
    if move[0] + move[1] <= self.capacity and (move[0] != 0 or move[1] != 0):
        # Check that none of the chickens are eaten (outnumbered) on the boat itself.
        if move[0] <= move[1] or move[1] == 0:
            # Add the proposed move to the list of possible moves.
            possible_moves.append(move)
```

Further, we make a slight modification as to the way that transitions are incorporated, by implementing subtraction, rather than just adding negative numbers, depending on the side of the river that the boat is located on. In this sense, the enumeration of the proposed moves is done in a more efficient fashion. We create a list of all proposed moves using a nested `for` loop, then use a `for` loop to check if the proposed move should be considered as a legal move.

Typically, we find that increasing the capacity of the boat shortens the solution path, as we are able to consider a graph that has a larger branching factor, and thus, may reach the end goal faster.

Extension 2

This extension implements the discussion question of the last discussion question. We consider the case if, in the service of their community, some chickens were willing to be made into lunch, designing a problem where no more than `E` chickens could be eaten, where `E` is some constant.

As highlighted, the state of the problem changes to be a 4-tuple representing the previous values, along with the number of chickens "available" to be eaten while still maintaining the integrity of the problem. In this case, the goal states change, as we do not need to include only the case where all of the chickens allowed to be eaten have been. This is represented as follows:

```
self.goal_states = [(0, 0, 0, n) for n in range(start_state[3] + 1)]
```

The primary modification for this is within the `is_safe` function, which now must return a tuple, representing the number of chickens eaten on each side of the river. (We do not consider the case where chickens are eaten in the boat.) This impacts the `get_successors` method in the following way:

```
# 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], state[3])

    # Determine the number of chickens eaten by modifying the 'is_safe' method.
    chickens_eaten = self.is_safe(next_state)

    # Check if a given state (determined by an initial state and transition) is safe.
        # The tuple (-1, -1) is considered to be an error state, or false, which does not
    if chickens_eaten != (-1, -1):
        successors_list.append((next_state[0] - chickens_eaten[0], next_state[1], next_st
```

The explanation of the way that the `is_safe` method changes is outlined within the comments, though we will cover it briefly here. Essentially, we calcluate the total number of chickens left, according to the instance variables and the current state. This allows for the instance variables to remain as they are (without modification) during any loop to determine the successor states.

The capacity constraints remains the same, with the exception of returning `(-1, -1)` in the case where the there is an error, indicating an illegal move. Finally, we consider the eating constraints, which allows up to a certain number of chickens to be eaten, and must return the relevant information. This is given as follows:

```
# Eating Constraints: Check to see what number of the chickens are eaten (outnumbered) by
# We consider the case where there are no chickens on a given side, which is okay.

# The 'lost' variables encode the number of chickens lost on the corresponding sides of t
(lost_1, lost_0) = (0, 0)

if state[0] < state[1] and state[0] != 0:
    lost_1 = state[0]
if total_chickens - state[0] < self.foxes - state[1] and total_chickens - state[0] != 0:
    lost_0 = total_chickens - state[0]

if lost_1 + lost_0 <= state[3]:
    return (lost_1, lost_0)
else:
    return (-1, -1)
```

As indicated, the state for this problem is a 4-tuple, representing the original information, along with the number of chickens left "available" to be eaten. The changes to the code to implement such a solution are relatively extensive, meriting the "extra credit". While the upper bound on the number of possible states for this problem is given as `(#Chickens + 1) * (#Foxes + 1) * (#Boats + 1) * (#Eaten + 1)`, this does not reflect the number of legal states, thus the graph is typically much smaller with a lower branching factor.

Extension 3

This extension simply provides a visualization of the appropriate solution path, using the console. In this way, we are able to visually confirm that all of the constraints are met. The code is written as follows:

```
for problem in problem_list:
    print(problem, end = '\n---\n')
    for state in bfs_search(problem).path:
        print("State: " + str(state))
        print(f"{'C ' * state[0] : <20}", end = '||')
        print(f"{'C ' * (problem.start_state[0] - state[0]) : >20}", end = '')
        print()
        print(f"{'F ' * state[1] : <20}", end = '||')
        print(f"{'F ' * (problem.start_state[1] - state[1]) : >20}", end = '')
        print('\n' + '_' * 41 + '\n')
    print()
```

This allows for visualization of solution paths that have relatively few chickens/foxes to start out with, as we are only able to visualize so many.