Report on PA1: Foxes & Chickens

COSC 76, Fall 2024

Wesley Tan

1 Modeling the Problem: States and Actions

1.1 State Representation

First, I needed an efficient way to represent the state of the problem at any point. I decided to use a tuple of three integers:

- `(chickens_left, foxes_left, boat_position)`

Here:

- `chickens_left`: Number of chickens on the **left bank**.
- `foxes_left`: Number of foxes on the **left bank**.
- `boat_position`: Position of the boat (1 for left bank, 0 for right bank).

The total number of chickens and foxes are constants defined by the problem and do not change, so they are not included in the state representation.

1.2 Actions and Successor States

Next, I identified all possible actions that could be taken from any given state, considering the boat's capacity and the movement rules. The boat can carry one or two animals, and at least one animal must be in the boat for it to move. The possible actions are:

- Move 1 chicken.
- Move 2 chickens.
- Move 1 fox.
- Move 2 foxes.
- Move 1 chicken and 1 fox.

1.3 Legality of States

A state is considered legal if:

- The number of chickens and foxes on both banks is within 0 to the total number.
- Chickens are not outnumbered by foxes on either bank unless there are no chickens on that bank.

I implemented an `is_legal_state` method to check these conditions for any new state generated. This checks for invalid numbers, and whether the chickens were safe on both banks:

```
# Check for invalid numbers
if chickens < 0 or foxes < 0 or \
   chickens > self.total_chickens or foxes > self.total_foxes:
    return False

# Check if chickens are safe on the left bank
if chickens > 0 and foxes > chickens:
    return False

# Check if chickens are safe on the right bank
if chickens_right > 0 and foxes_right > chickens_right:
    return False
```

1.4 State Space Size

The calculation of the state space size in the given text is incorrect for the general case with F foxes and C chickens.

1. Chickens on the left bank can be $0, 1, 2, \ldots,$ or C: $C + 1$ possibilities
2. Foxes on the left bank can be $0, 1, 2, \ldots,$ or F: $F + 1$ possibilities
3. Boat position can be 0 or 1: 2 possibilities

Therefore, the total number of possible states (without considering legality) is:
$(C + 1) \times (F + 1) \times 2$

1.5 State Graph Illustration

See Figure 1 below.

2 Breadth-First Search Implementation

2.1 Algorithm Explanation

BFS explores the state space level by level, expanding all nodes at the current depth before moving to the next. This ensures that the first solution found is the shortest path in terms of the number of moves, as BFS guarantees that the shortest path to the goal state is found by exploring all possible paths of equal depth before moving deeper.

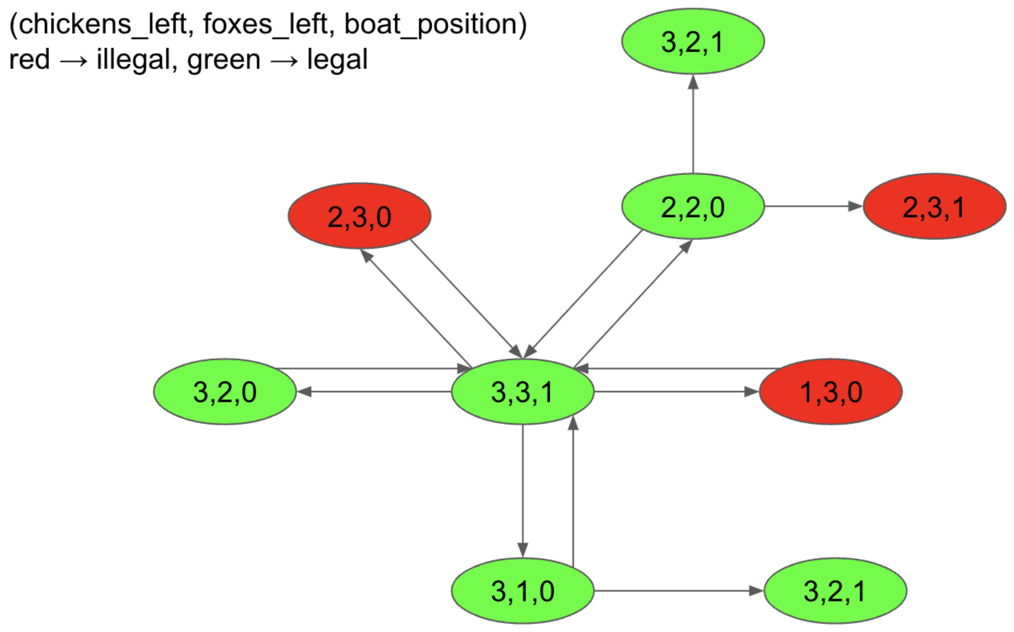

(chickens_left, foxes_left, boat_position)
red → illegal, green → legal
3,2,1
2,3,0
2,2,0
2,3,1
3,2,0
3,3,1
1,3,0
3,1,0
3,2,1

Figure 1: State Map

2.2 Data Structures Used

- **Queue (deque)**: The BFS algorithm uses a queue to store the frontier, which is the set of states yet to be explored. The queue ensures that the states are processed in a first-in-first-out (FIFO) order, guaranteeing that the shallowest states are explored first.
- **Set**: To avoid revisiting the same state, I used a set to store all previously visited states. This prevents redundant explorations and improves efficiency.
- **`SearchNode` Class**: As suggested, there is a `SearchNode` class to represent each state in the search. The class not only stores the state information but also maintains a reference to the parent node. This parent-child linkage is crucial for reconstructing the solution path once the goal state is reached.

2.3 Evaluation of Algorithm Performance

The BFS algorithm worked as intended for the smaller problem instances. When tested with the initial state $(3, 3, 1)$, BFS successfully found the shortest solution path, requiring only 12 moves to transport all chickens and foxes safely across the river. The number of nodes visited was relatively small (15 nodes), demonstrating the efficiency of BFS in this case.

```
Chickens and foxes problem: Start State (3, 3, 1)
attempted with search method BFS
number of nodes visited: 15
solution length: 12
path: [(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1),
(1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]
```

For larger problems, such as the initial state $(5, 4, 1)$, BFS continued to work well, finding a solution in 16 moves after visiting 30 nodes. However, for the problem with the initial state $(5, 5, 1)$, no solution was found, which suggests that this configuration may be unsolvable without violating the safety rules.

```
Chickens and foxes problem: Start State (5, 4, 1)
attempted with search method BFS
number of nodes visited: 30
solution length: 16
path: [(5, 4, 1), (4, 3, 0), (5, 3, 1), (3, 3, 0), (4, 3, 1),
(3, 2, 0), (3, 3, 1), (2, 2, 0), (3, 2, 1), (2, 1, 0), (2, 2, 1),
(0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1), (0, 0, 0)]

Chickens and foxes problem: Start State (5, 5, 1)
attempted with search method BFS
no solution found after visiting 13 nodes
```

For larger problem instances, the BFS algorithm still performed adequately, but as the state space grew, the number of visited nodes increased, leading to higher memory usage. This is a typical trade-off with BFS, as it guarantees finding the shortest path but at the cost of increased space complexity.

For BFS, Time Complexity is $O(b^d)$. Space Complexity is $O(b^d)$

3 Path-Checking Depth-First Search Implementation

3.1 Algorithm Explanation

I implemented a recursive Depth-First Search (DFS) with path checking. DFS explores the state space by going as deep as possible along a branch before backtracking. In this version of DFS, path checking is used to ensure that cycles are avoided by keeping track of the current path. If the algorithm attempts to revisit a state already on the current path, it backtracks to prevent unnecessary exploration. This is important in problems where cyclic graphs or repeated states could cause infinite loops or redundant work.

3.1.1 Design Decisions

Several design decisions were made during the implementation of this algorithm:

- **Path Checking**: The algorithm includes path checking to avoid cycles. It keeps track of the current path in a set called `path`. If the algorithm revisits a state that is already in the path, it backtracks to avoid getting stuck in an infinite loop.
- **Recursive Approach**: The choice of recursion for this implementation was made to align with the natural depth-first nature of the algorithm.

3.2 Evaluation

3.2.1 Does the Algorithm Work?

Yes, the DFS algorithm works as expected for many initial problem configurations. It successfully finds a solution path for smaller instances of the problem, such as the $(3, 3, 1)$ configuration, where three chickens and three foxes need to be transported across the river. The recursive implementation with path checking ensures that the algorithm avoids cycles and redundant paths. However, one drawback of DFS is that it does not always find the shortest path, and in some cases, it may take longer to find a solution compared to breadth-first search (BFS).

```
Chickens and foxes problem: Start State (3, 3, 1)
attempted with search method DFS
number of nodes visited: 20
```

```
solution length: 12
path: [(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1),
(1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0),
(1, 1, 1), (0, 0, 0)]
```

The algorithm explored 20 nodes and found a solution of length 12. While this solution works, it may not be the most efficient path in terms of the number of moves.

For larger problem instances, such as the $(5, 4, 1)$ configuration, DFS takes longer to find a solution. It explores deeper branches first, which sometimes leads to unnecessary exploration. This is evident in the number of nodes visited and the length of the solution path compared to BFS.

```
Chickens and foxes problem: Start State (5, 4, 1)
attempted with search method DFS
number of nodes visited: 42
solution length: 18
path: [(5, 4, 1), (4, 3, 0), (5, 3, 1), (3, 3, 0), (4, 3, 1),
(3, 2, 0), (3, 3, 1), (2, 2, 0), (3, 2, 1), (2, 1, 0),
(2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (2, 1, 1),
(1, 0, 0), (1, 1, 1), (0, 0, 0)]
```

In this case, DFS visited 42 nodes and found a solution of length 18. In comparison, BFS for the same problem visited only 30 nodes and found a shorter solution of length 16. This demonstrates that DFS can take longer and may not find the shortest path.

Time Complexity Worst-case: $O(b^m)$

- b is the branching factor
- m is the maximum depth of the search tree

The time complexity is exponential because DFS might explore all possible paths to the deepest level before finding a solution. This explains why DFS explored more nodes than BFS for the larger problem.

Space Complexity Worst-case: $O(bm)$

- This is because DFS only needs to store the current path and the siblings of nodes on this path

3.2.2 Partial Successes and Areas for Improvement

While the DFS implementation works for smaller instances and finds valid solutions, it has some limitations:

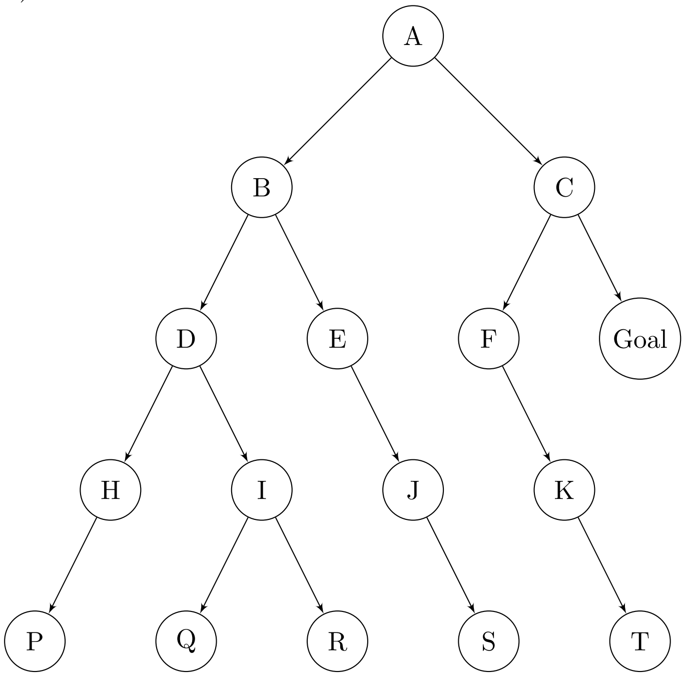
- **Suboptimal Paths**: As DFS explores branches to their full depth before backtracking, it does not guarantee the shortest path. This is evident from the test results where DFS found longer paths compared to BFS.
- **Increased Run Time for Larger Instances**: For larger problem configurations, DFS tends to explore more nodes than necessary, especially when deep branches do not lead to a solution. This results in longer run times compared to BFS, which systematically explores shallower nodes first.

Despite these limitations, DFS remains a useful approach for problems with large state spaces, as its memory usage is lower than BFS. For example, in problems where the state space is too large for BFS to handle due to memory constraints, DFS with path checking can be a viable alternative.

3.3 Discussion

3.3.1 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.

Path-checking DFS is memory-efficient because it only needs to store the nodes along the current path, rather than all visited nodes. This significantly reduces memory usage compared to BFS. The path-checking DFS space complexity is $O(d)$ but the space complexity for BFS is exponential, being $O(b^d)$



Consider a graph where the goal is at a shallow depth, but there are many deep branches that do not lead to the goal. DFS may explore these deep branches first, taking a long time to backtrack and find the goal. BFS, on the other hand, would find the goal quickly by exploring all nodes at each depth level.

Consider the results from the $(5, 4, 1)$ initial state:

- **BFS**: Visited 30 nodes, solution length 16.
- **DFS**: Visited 42 nodes, solution length 18.

DFS visited more nodes and found a longer solution path compared to BFS, demonstrating that DFS can take more time and may not find the shortest path first.

3.3.2 Does memoizing (this is path-checking) DFS save significant memory with respect to breadth-first search? Why or why not?

Yes, path-checking DFS can potentially save significant memory compared to BFS. The memory complexity of path-checking DFS is $O(m)$, where m is the maximum depth of the search. In contrast, BFS has a memory complexity of $O(min(n, b^d))$, which can be much larger, especially for graphs with large branching factors or deep solutions.

- *Memoizing DFS* explores as deep as possible along each branch before backtracking, while avoiding revisiting states.
- *BFS* explores nodes level by level, ensuring that the shortest path (in terms of the number of steps) is found.

4 Iterative Deepening Search Implementation

4.1 Algorithm Explanation

I implemented Iterative Deepening Search (IDS) by combining the depth-first approach with an iterative deepening mechanism. IDS repeatedly performs Depth-First Search (DFS) with increasing depth limits, starting from 0 and incrementing the limit until a solution is found. The key advantage of IDS is that it combines the memory efficiency of DFS with the completeness and optimality of Breadth-First Search (BFS). Unlike DFS, which explores the search space to its full depth before backtracking, IDS ensures that shallower solutions are found first, similar to BFS.

4.2 Evaluation

4.2.1 Does the Algorithm Work?

Yes, the IDS algorithm successfully finds solutions for the foxes and chickens problem. It consistently finds the shortest solution path, similar to BFS, but with significantly lower memory usage. However, the repeated searches in each iteration result in higher time complexity compared to BFS.

```
Chickens and foxes problem: Start State (3, 3, 1)
attempted with search method IDS
number of nodes visited: 296
solution length: 12
path: [(3, 3, 1), (2, 2, 0), (3, 2, 1), (3, 0, 0), (3, 1, 1),
(1, 1, 0), (2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0),
(1, 1, 1), (0, 0, 0)]
```

In this test, IDS visited 296 nodes to find the solution path of length 12. The number of nodes visited is much higher than in BFS (which visited only 15 nodes for the same problem), indicating that IDS is less time-efficient due to the repeated searches at increasing depth limits.

For larger problem configurations, IDS also successfully found solutions, though the number of nodes visited continued to increase due to the repeated nature of the search.

```
Chickens and foxes problem: Start State (5, 4, 1)
attempted with search method IDS
number of nodes visited: 2107
solution length: 16
path: [(5, 4, 1), (4, 3, 0), (5, 3, 1), (3, 3, 0), (4, 3, 1),
(3, 2, 0), (3, 3, 1), (2, 2, 0), (3, 2, 1), (2, 1, 0),
(2, 2, 1), (0, 2, 0), (0, 3, 1), (0, 1, 0), (1, 1, 1),
(0, 0, 0)]
```

In this case, IDS visited 2107 nodes, which is significantly higher than both BFS and DFS for the same problem. This demonstrates that IDS is less efficient in terms of time, but it maintains its memory advantage over BFS.

4.2.2 Partial Successes and Areas for Improvement

The IDS implementation works well in terms of finding optimal solutions without requiring excessive memory. However, its major drawback is the high number of nodes visited due to repeated exploration at increasing depth limits. While this is expected behavior for IDS, it results in longer runtimes compared to BFS.

The IDS implementation works well in terms of finding optimal solutions without requiring excessive memory. However, its major drawback is the high number of nodes visited due to repeated exploration at increasing depth limits. While this is expected behavior for IDS, it results in longer runtimes compared to BFS.

- **Time Complexity**: IDS suffers from repeated exploration of nodes at each depth limit, which leads to higher time complexity than BFS. In the worst case, the time complexity of IDS is $O(b^l)$, where b is the branching factor and l is the depth limit. However, the constant factor is higher than BFS due to repeated explorations. In the example of the initial state $(5, 4, 1)$, IDS visited over 2000 nodes, whereas BFS visited only 30.
- **Memory Efficiency**: The key advantage of IDS over BFS is its low memory usage. By avoiding the storage of all visited states across iterations, IDS achieves a space complexity of $O(bl)$, which is significantly better than the $O(b^d)$ space complexity of BFS. This makes IDS suitable for problems with large state spaces where BFS would be impractical due to memory constraints.

4.3 Discussion

4.3.1 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.)

If the node is far away from the node, path-checking DFS saves significant memory compared to BFS because it only needs to store the current path. While BFS stores all visited states and can consume a lot of memory for large graphs, path-checking DFS's memory usage grows linearly with the depth of the search tree.

Path-checking DFS saves memory by only storing the current path, resulting in memory usage that grows linearly with the depth of the search tree ($O(d)$, where d is the maximum depth). However, this approach can lead to significantly increased run-time in graphs with cycles or multiple paths, as it may redundantly explore the same nodes via different paths. The time complexity in such cases can become exponential ($O(b^m)$, with b as the branching factor and m as the maximum depth), making it inefficient.

Memoizing DFS, on the other hand, keeps track of all visited states to avoid redundant explorations, resulting in time complexity of $O(n)$ and memory usage of $O(n)$, where n is the number of nodes in the graph. This memory consumption is similar to that of breadth-first search (BFS), which also has $O(n)$ memory usage but benefits from finding the shortest path more efficiently due to its level-by-level exploration.

Considering that memoizing DFS does not offer significant memory savings over BFS and may not provide additional advantages, it would be preferable to use BFS when searching a graph. BFS ensures optimality in finding the shortest path and maintains acceptable time and memory complexities. Therefore, unless memoizing DFS offers clear benefits in a specific context, BFS is generally the better choice for graph search problems.

5 Lossy Foxes & Chickens

To accept the loss of a certain number of chickens, you would edit the state representation to include `chickens_eaten`:

- `(chickens_left, foxes_left, boat_position, chickens_eaten)`

  Here:

- `chickens_left`: Number of chickens on the **left bank**.
- `foxes_left`: Number of foxes on the **left bank**.
- `boat_position`: Position of the boat (1 for left bank, 0 for right bank).
- `chickens_eaten`: Number of chickens eaten.

  We can use `max_chickens_eaten` as the value for E. We fist initialize this.

```python
def __init__(self, start_state, max_eaten_chickens):
    self.start_state = start_state + (0,)
    self.goal_state = (0, 0, 0, min(start_state[0], max_eaten_chickens))
    self.total_chickens = start_state[0]
    self.total_foxes = start_state[1]
    self.max_eaten_chickens = max_eaten_chickens
```

We will need to edit the function `get_successors` to update the number of chickens eaten.

```python
for move in possible_moves:
    new_chickens = chickens - move[0] if boat == 1 else chickens + move[0]
    new_foxes = foxes - move[1] if boat == 1 else foxes + move[1]
    new_chickens_eaten = chickens_eaten + move[2]
    new_state = (new_chickens, new_foxes, boat_new, new_chickens_eaten)
    if self.is_legal_state(new_state):
        successors.append(new_state)
```

And likewise, what constitutes a legal state changes as well as the cases where `chickens_eaten <= max_chickens_eaten` are legal

```python
def is_legal_state(self, state):
    chickens_left = state.chickens_left
    foxes_left = state.foxes_left
    chickens_eaten = state.chickens_eaten
    total_chickens = self.total_chickens
    total_foxes = self.total_foxes

    chickens_alive = total_chickens - chickens_eaten

    # Ensure the counts on the left bank are valid
    if chickens_left < 0 or foxes_left < 0 or
    chickens_left > chickens_alive or foxes_left > total_foxes:
        return False

    # Check for foxes eating chickens on the left bank
    if chickens_left > 0 and foxes_left > chickens_left:
        # Add chickens that will be eaten on the left bank
        chickens_eaten += chickens_left
        chickens_left = 0

    # Check for foxes eating chickens on the right bank
    chickens_right = chickens_alive - chickens_left
    foxes_right = total_foxes - foxes_left
    if chickens_right > 0 and foxes_right > chickens_right:
        # Add chickens that will be eaten on the right bank
        chickens_eaten += chickens_right
        chickens_right = 0

    # Ensure that no more than E chickens have been eaten
    if chickens_eaten > self.max_chickens_eaten:
        return False

    return True
```

And importantly, there are now mulitple goal states as:

```python
def is_goal_state(self, state):
```

```
"""
Check if the current state is the goal state.
"""
chickens_left, foxes_left, boat_position, chickens_eaten = state
return chickens_left == 0 and foxes_left == 0
and boat_position == 0 and chickens_eaten <= self.max_chickens_eaten
```

5.1 Chickens Left (`chickens_left`)

Possible values range from 0 to C, where C is the total number of chickens that are still alive. Since up to E chickens can be eaten, the number of chickens left alive ranges from $C - E$ to C. However, to simplify the calculation, we can consider 0 to C.

- Number of possible values for `chickens_left`: $C + 1$

5.2 Foxes Left (`foxes_left`)

Foxes are not eaten, so their possible counts range from 0 to F, where F is the total number of foxes.

- Number of possible values for `foxes_left`: $F + 1$

5.3 Boat Position (`boat_position`)

Two possible positions: 0 (right bank) or 1 (left bank).

- Number of possible values: 2

5.4 Chickens Eaten (`chickens_eaten`)

Possible values range from 0 to E.

- Number of possible values: $E + 1$

Total number of states: $(C + 1) \times (F + 1) \times 2 \times (E + 1)$**.**

6 References

- Russell, S. J., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.

- Python Documentation: collections.deque, sets.