Freckers Project A Report

Search Strategies

In order to begin searching for the best path for the Red Frog, we need to start with building a traversal data structure for the searching algorithms. We decided on a traversal tree consisting of TreeNode class instances, representing the different Coordinates that the Frog can travel to, according to the rules in the game specification. The TreeNode instances contain information on the position, nominally the children and parents (represented by dictionaries that map the node to its direction from the subject node).

This traversal tree structure allows for a **dynamic representation of the game state at any given time**. The tree will be expanded **recursively** until the goal state is found or all possible paths have been explored. Ultimately, the search algorithms can easily and efficiently traverse through the tree, expanding nodes and easily accessing the information they need, such as building a priority queue utilizing the heuristic in A* Search. The **parent-child relationships** between the nodes allow the bidirectional search to efficiently traverse both ways without adding time complexity.

We started with **DFS and BFS** since they were the most basic algorithms, but overall, they were both inefficient. Therefore, we decided to implement both **A* and Bidirectional Search**, as they are proven to be more efficient in terms of time and space complexity.

Variables definition for complexity analysis:

b: maximum branching factor of the search tree **d**: depth of the least-costing solution

m: maximum depth of the goal state **G**: number of goals

1. Depth-First Search

Used recursion with a 'visited' list and jump variable to reduce node storage and avoid loops. Explored all paths to find the shortest solution.

2. Breadth-First Search

Used an iterative approach with a queue, 'visited' and 'added' lists to avoid redundant expansions, and a jump variable for multiple jumps.

3. **Bidirectional Search (Chosen)**

Used two queues, 'visited' lists, and dictionaries to alternate forward and backward search, merging paths at intersections via `combine_paths`. Repeated for each goal. Time and space complexity are both **O(G * d * b^d/2)** due to deep path copying and queue storage.

4. A* Search

Used a priority queue with 'visited' and 'added' lists and a jump variable for iterative search. Time complexity is **O(b^d * (d+logN))**, from path copying and queue operations, and space complexity is **O(d * b^d)**, mainly from the queue and stored paths.

Heuristic Function

Fundamentally, the heuristics of a node X in our search tree is the distance from node Y to the goal divided by 2, with node Y being the closest node (or nodes with the same distance) to the goal that from X the frog can travel to **in one move**:

$$h_X = \frac{BOARD\_N - max(r_{Y1}, r_{Y2}, ...)}{2}$$

(with Y1, Y2, … being the nodes that can be travelled to from X in 1 move)

We also manually set Goal Nodes' (r = BOARD_N - 1) heuristic to 0.

In general, this heuristic speeds up the search process significantly since our AStar will only explore the nodes that have a path leading it closer to the goal, since the frog can jump over other frogs, making some moves' distance further => not exploring already expensive paths. In the end, **our A* explored, on average, the least nodes while running compared to the others**, but we ultimately **chose to submit our Bidirectional**.

The reason for this is that **even though our AStar performs the best**, we did some testing with **deadends with desirable jumps into**, and chose bidirectional because it seems to be more reliable in these trap cases or similar, since it also explores from the goal back. **We will continue to perfect our A* and heuristic in Project B.**

6-Red-Frog Game Discussion

Impact: We must represent the state as a combination of the positions of all six frogs. This increases the number of possible states exponentially because if each frog can move up to **b** positions, then there are roughly **b**^{6} possible configurations. With one frog, each move only involves that frog, so the branching factor might be **b**. With six frogs, if every frog can move independently, the branching factor might be roughly **6 * b** in the worst case.

Solution: First, we will **redefine the state** to include all six frogs' positions. Then, we **adapt our search algorithm** (e.g., use multi-agent A*) to handle the increased branching factor and combinatorial state space. Our **heuristic function** will be redesigned so that it considers the joint state and helps prune the search space. Finally, we will **consider more memory-efficient methods** or hierarchical planning approaches if the full joint search space is too large.