Search Strategy for Single Player Freckers

Chuanmiao Yu & Yurim Cho

Introduction

This report presents our implementation of the A* search algorithm for the Single Player Freckers game.

Search strategy

For our implementation of the game, we used the A* search strategy. We began by creating and initializing four lists: cell_details, which is a two-dimensional array that stores detailed information about each node, including f(n), g(n), h(n), and the parent node position; open_list, which holds the positions of nodes that have been discovered but not fully evaluated; close_list, a two-dimensional Boolean array that marks which nodes have been fully evaluated; and goal_list, which contains the positions of goal nodes in the last row (r = 7).

Detailed implementation

The A* search strategy is then applied to find the path. First, we find the node with the lowest f(n) from the open_list and remove it from the open_list, adding it to the close_list. Next, we check all reachable nodes from the current node. If a node is a goal cell, we return the path. If not, we calculate new values for h(n), g(n), and f(n). If the node is either not in the open_list or can be reached via a better path, we update its information and add or keep it in the open_list. If at any point the open_list becomes empty and no path is found, the function returns None.

Complicity

Assume L is the number of lily pad, N is the total number of nodes

- **Time complicity: $O(L^2)$**

The time complexity of our A* implementation is $O(L^2)$. The find_lowest_f_cell function performs a linear scan of the open_list, taking O(L) time in the worst case. Since the main loop runs at most L times, each lily pad being processed once, the overall complexity is $O(L^2)$. If a priority queue were used to manage the open_list, the time complexity of selecting the node with the lowest f-value could be reduced to O(log L), improving overall efficiency.

- **Space complicity: O(V)**

The algorithm's space complexity is O(V), as it maintains data structures covering the entire board. However, since it only processes lily pad positions, optimizing the data structures to store only relevant information could reduce the space complexity to O(L), making the algorithm more memory efficient.

Heuristic Function

- **Computation**

Our A* algorithm uses the Euclidean distance as the heuristic, calculating the shortest straight-line distance to the nearest goal lily pad. This ensures an efficient and admissible estimation of the cost to the goal.

- **Optimization**

The heuristic speeds up the search by guiding the algorithm toward goal states, reducing unnecessary explorations. Empirical tests show that it improves efficiency while still guaranteeing an optimal solution.

Impact of Moving All Six Red Frogs

- **Increased Search Complicity**

Moving all six red frogs to the opposite side significantly increases the complexity of the search. The state space grows exponentially as we track all frogs' positions simultaneously.

- **Lily Pad Limitation**

Since frogs share lily pads, limited space becomes a challenge. Frogs that reach their goal occupy lily pads, potentially blocking others. In such cases, the grow function is crucial for creating new lily pads and opening alternative paths.

Improvement

- **Enhanced Data Structure**

Replace current list-based implementation with priority queues to efficiently handle the larger state space of six frogs, reducing search time complexity.

- **Frog Movement Priority**

Prioritize frogs with fewer available moves to prevent deadlocks. This reduces the need for "grow" operations and optimizes the total number of moves required.

- **Jump Distance Optimization**

Give preference to frogs that can make longer jumps, particularly those that can jump over blue frogs to reach distant lily pads, maximizing board coverage with fewer moves.

Conclusion

In this implementation of Single Player Freckers, we used A* search to efficiently guide frogs to their goals. The Euclidean distance heuristic helped optimize the search, balancing efficiency and optimality. The time complexity of $O(L^2)$ and space complexity of $O(V)$ are manageable for typical game boards, but the complexity increases with multiple frogs.

To improve, we could replace the list-based open_list with a priority queue, reducing time complexity. Additionally, prioritizing frogs with fewer available moves and optimizing jumps for greater coverage would further improve efficiency and reduce the overall number of moves. These adjustments would make the algorithm better suited for handling more complex scenarios.