Search Strategy for Two-Player Freckers

Chuanmiao Yu & Yurim Cho

Introduction

In this project, we developed an agent for the two-player game Freckers. The goal of the agent is to make optimal decisions in each round to defeat the opponent. To achieve this, we explored and evaluated several algorithms in search of the most effective approach. This report outlines the final method we selected, along with a detailed explanation, an analysis of the performance evaluation of each method, the steps we took to optimize efficiency and decision-making speed, and additional inspirations that contributed ideas applied in our agent.

Approach

- **Algorithm Type**

We used the A* search combined with Alpha-Beta Minimax Pruning. This method was selected because it helps us maximize our advantage while minimizing the opponent's potential gain. While it can select the frog that is optimal to move for current round, it also finds the optimal path.

- **Modifications**

We made several modifications to the standard Minimax algorithm to better suit the Frogs Game:

1. **Alpha-Beta Pruning**

We integrated Alpha-Beta Pruning into the Minimax algorithm to improve efficiency by eliminating branches that cannot influence the final decision. This significantly reduces the number of nodes evaluated.

2. **Depth Limiting**

A configurable depth limit was applied to ensure that recursion terminates within a reasonable time. This makes the algorithm computationally efficient during gameplay while still providing good strategic decisions.

3. **A* Search Implementation**

We implemented A* search with a custom heuristic function to find optimal paths more efficiently. This algorithm uses a priority queue to explore the most promising states first, focusing computational resources on the most valuable search paths.

4. **Path Validation**

We added specialized functions to validate move paths and detect invalid jump sequences. This prevents the agent from attempting illegal moves and ensures robust gameplay.

- **Evaluation Function**

We designed multiple evaluation functions:

1. **Evaluate(): Evaluates each movement with a score**

It scores all reachable non-terminal nodes and selects the move that leads to the state with the highest score. The score is determined based on both the player's and the opponent's potential actions and positions.

Positive Contributions (Player's Perspective):

- The number of the player's frogs that can move: each movable frog adds 1 point.
- The distance from each player frog to the goal: shorter distances yield higher scores, with each step closer worth 1 point.
- The number of the player's frogs that have reached the goal: each such frog adds 10 points.

Negative Contributions (Opponent's Perspective):

- The number of the opponent's frogs that can move: each movable opponent frog subtracts 1 point.
- The number of opponent frogs that have reached the goal: each such frog subtracts 10 points.

This scoring system allows the algorithm to evaluate intermediate states even when the game has not reached a terminal state, enabling it to choose moves that offer the most strategic advantage at each stage.

2. **Heuristic(): Evaluates distance of current frog to the destination**

- Calculates how far each frog is from its goal row
- Provides admissible estimates to guide the A* search toward promising states

- **Machine learning application**

We did not use machine learning in our approach because the problem is relatively small and deterministic. Traditional search algorithms like Minimax with Alpha-Beta pruning provide an efficient way to explore all possible moves and outcomes without the need for training a model. Additionally, machine learning requires a large dataset for training and might be excessive for this problem, where a direct search strategy can achieve optimal or near-optimal performance.

Performance evaluation

We tried and evaluated two different approaches by comparing total time and average time per move. This helped us choose the approach with better performance.

- **Simple Alpha-Beta Minimax Pruning**

```
Game Performance:
Total Game Time: 239.8804 seconds
Number of Moves: 36
Average Time per Move: 2.8668 seconds

Game Performance:
Total Game Time: 239.9934 seconds
Number of Moves: 37
Average Time per Move: 3.6643 seconds
* referee : game over, winner is RED
* referee @ result: player_1 [agent:Agent]
```

This method did not perform well. It took a long time to run and required many rounds, which was not efficient. The main reason is that Minimax uses brute force to generate all possible paths for each frog. This process is very slow because the number of paths is extremely large. Also, the frog may not always choose the best path, some paths seem good but lead to poor results.

- **A* with Alpha-Beta Minimax Pruning**

```
Game Performance:
Total Game Time: 2.5027 seconds
Number of Moves: 33
Average Time per Move: 0.0227 seconds

Game Performance:
Total Game Time: 2.6181 seconds
Number of Moves: 34
Average Time per Move: 0.0260 seconds
* referee : game over, winner is RED
* referee @ result: player 1 [agent:Agent]
```

To improve performance, we added A* search on top of Minimax. Minimax is used to decide which frog should move to get the best overall result for the player. A* is used to find the best path for a specific frog to reach the goal. With this combination, Minimax no longer needs to evaluate all possible paths for every frog. A* finds the best movement path for each frog in advance, and Minimax only evaluates the result of that path. This greatly improves both speed and decision quality.

Technical Innovations and Optimizations

1. Implementation of Alpha-Beta Pruning

- Reduced the search space by cutting off unnecessary branches, which significantly improved search efficiency.
- Alpha and Beta values were properly set in the alpha_beta_search function to avoid exploring unpromising paths.

2. Use of A* Search Algorithm

- Used a priority queue to expand states based on their priority.
- Designed a specific heuristic function to estimate the distance from each frog to its goal.
- Introduced a visited set to avoid repeated searches of the same states.

3. Efficient Board Evaluation System

- Considered multiple factors such as the number of frogs, their positions, and their movement ability.
- Gave extra weight to frogs on important positions.
- Tested and compared different evaluation functions.

4. Custom Data Structure Design

- Used a dictionary to map each frog to its reachable cells.
- Used nested lists to track jump paths.

5. Path Validation and Optimization

- Applied Depth-First Search to explore all possible jump paths for a frog.

6. Memory Optimization

- Used copy.deepcopy to avoid changing the original state during expansion.
- Designed efficient methods for representing and comparing board states.
- Limited the queue size to prevent memory overflow.

7. Optimized Terminal State Detection

- Instead of checking the entire board, only checked the number of frogs in the target row.

These technical improvements helped the AI agent make better decisions and significantly increases overall performance.

Supporting Work

During our research and testing, we gathered several useful insights from existing game strategies, particularly from chess programming, which we applied to our agent. Here are the key ideas we derived:

1. Hybrid Search Strategy

From studying traditional search algorithms, we found that combining heuristic search (like A*) with adversarial search (like Minimax) can significantly improve efficiency. In Freckers, the game state is highly dynamic, and relying on just one search method often makes it difficult to process all possible moves within limited computational resources. We were inspired by the hybrid search methods used in chess AI and decided to combine A* and Alpha-Beta pruning to optimize the search process.

2. Multi-Dimensional Evaluation Function

We learned from chess programs that traditional Minimax algorithms usually rely on a simple evaluation function, which can be insufficient for complex games. To address this, we used an evaluation method that considers multiple factors, allowing the agent to make better decisions based on these factors.

3. Dynamic Resource Management

Drawing insights from chess engines, we adopted effective time management techniques. These strategies were applied to our agent including pruning, state tracking, and queue size limitations to ensure that the AI consistently made efficient decisions within strict time limits.

4. Performance Evaluation Timing Mechanism

To assess the performance of each method, we implemented a timing mechanism to measure the total execution time. This approach enabled us to identify the most efficient algorithm and optimizing the agent's performance.

These insights from research helped us design a more efficient and adaptive agent, handles different circumstances of Freckers and making optimal decisions.

Conclusion

Through several attempts with different algorithms and search methods, we successfully created an agent that makes optimal decisions and defeats the opponent. By applying these methods, we were able to compare the efficiency of each one and observe how combining them could further improve performance. Overall, the choice of methods has a significant impact on the agent’s effectiveness.