Search & Games: Fishing Derby

Martín Bravo
Sebastián Cavalieri

November 22, 2024

Abstract

This document details the implementation of a player controller for the Fishing Derby game, focusing on decision-making algorithms and their effectiveness. It covers key concepts such as the Minimax algorithm, alpha-beta pruning, heuristic evaluations, iterative deepening, and optimization techniques.

Contents

1 Introduction

This report focuses on implementing a player controller for the Fishing Derby game, which aims to simulate decision-making using adversarial search. Key components such as Minimax, alpha-beta pruning, and heuristics are explained and analyzed. Challenges and design decisions during development are also discussed.

2 Key Decisions Made During Development and Functionality Overview

.

2.1 Minimax Algorithm and Alpha-Beta Pruning

In the case of the maximizing player, it is implemented as follows:

```python
if player:
    value = float("-inf")
    for child in children:
        value = max(value, self.minimax(child, alpha, beta, not
           player, max_depth))
        alpha = max(alpha, value)
        if beta <= alpha:
            break
    self.states_stored[state_key] = [value, alpha, beta]
    return value
```

Listing 1: Minimax implementation

We decided for the function to start with the value of (-inf), following the pseudo code provided in the assignment specification, which then proceeds to try and maximize it. For each child computed, there is a recursive call to minimax to evaluate the child's value and alpha is updated to the highest value found until that moment. And here is where we choose the pruning to take place, when beta has a lower or equal value to alpha, since no better result can be achieved by exploring further through that path of moves, the minimizing player wouldn't allow this branch to be chosen.

On the other hand, when it comes to the minimizing player, the implementation is as follows:

```python
else:
    value = float("inf")
    for child in children:
        value = min(value, self.minimax(child, alpha, beta, not
           player, max_depth))
        beta = min(beta, value)
        if beta <= alpha:
            break
    self.states_stored[state_key] = [value, alpha, beta]
    return value
```

Listing 2: Minimax implementation

The function starts with the value of (inf) to try and minimize it, similarly as before, minimax is called recursively in order to evaluate the child's value and in this case beta is updated to the lowest value found until that moment, and the pruning happens when beta is lower or equal to alpha for the same reason as before.

2.2 Time limit

We decided for the time limit to be 60ms, instead of the 75ms specified as the limit in the assignment explanation, in order to avoid overhead issues and have successful executions without timeout errors.

2.3 Heuristic

The designed heuristic is defined as follows:

```python
fish_value = 0
for fish, pos in fish_positions.items():
    fish_score = state.fish_scores[fish]
    my_distance = self.manhattan_distance(hook_positions[0], pos)
    opp_distance = self.manhattan_distance(hook_positions[1], pos
        )

    # Calculate fish value
    fish_value += fish_score / (my_distance + 1) - fish_score /
        (2 * opp_distance + 1)
```

Listing 3: Implementation Example

Its main purpose is to combine how worthy the fish is, how far we are from them, and how far the opponent is from them. As we can see, we can compute that using the following expression:

$$\frac{fish\_score}{dist + 1} - \frac{fish\_score}{2 \cdot opp\_dist + 1}$$

The 2-factor was obtained by trial and error. In my many trial cases, we realized that it was better to reward the ship when it tried to block or make the other boat stay away from the fish (for example, test−2). The +1 is included for obvious reasons to not divide by zero if $dist = 0$.

2.4 Iterative Deepening Search

We decided to implement IDS to make sure that the search always produces a result more efficiently. And we implemented it as follows:

```python
for max_depth in range(1, 20):
    for child in children:
        score = self.minimax(
            child, alpha=float("-inf"), beta=float("inf"),
            player=False, max_depth=max_depth
        )
        if score > best_score:
            best_move = child.move
            best_score = score
```

Listing 4: Iterative deepening search implementation

Where the depth is incrementally increased in a loop bounded by a maximum depth of 20, which we agreed on because of trial and error tests, and because of that we believe it to be a significant amount of depth to search for the best moves to win the game for the player.

2.5 Move Ordering

We decided to implement move ordering since it improves the effectiveness of Alpha Beta pruning and search speed, allowing higher depth values to be explored because of that improvement of efficiency in the search process.

Every time we deal with children, we order them:

```
children = current_node.children
children.sort(key=lambda child: self.heuristic(child), reverse=
    player)
```

Listing 5: Move ordering implementation

Before the minimax algorithm takes place(in the first level), the children are ordered in a way that the best move out of them is evaluated first using the heuristic function to determine that. Inside the minimax algorithm, if it is the maximizing player's turn, the sort is still in descending order so that the best move is evaluated first, but if it's the minimizing player's turn, the sorting is done in ascending order, evaluating the worst move first.

2.6 Repeated States Checking

We decided to implement repeated state checking to achieve deeper values of depth, to improve Alpha Beta pruning, efficiency, and optimization. We did it initially by introducing a global variable to store states:

```
self.states_stored = {}
```

Listing 6: Repeated states checking implementation

Later, doing string representations of the game states for hashing:

```
state_key = self.state_to_key(current_node)
```

Listing 7: Repeated states checking implementation

Implementing the following function:

```
def state_to_key(self, node):
        """
        Create a string representation of the game state for
            hashing to use it as key.
        """
        state = node.state
        key = (
            f"{state.player}{state.player_scores[0]}{state.
               player_scores[1]}"
            f"{state.hook_positions[0][0]}{state.hook_positions
               [0][1]}"
            f"{state.hook_positions[1][0]}{state.hook_positions
               [1][1]}"
        )
        for fish in state.fish_positions:
            key += f"{state.fish_positions[fish][0]}{state.
               fish_positions[fish][1]}"
        for score in state.fish_scores.values():
            key += str(score)
        return key
```

Listing 8: Repeated states checking implementation

Then the stored states are reviewed to check if the states have been encountered before, and if it has, the algorithm retrieves the previously evaluated value for that state, using the already explored progress from there:

```
if state_key in self.states_stored:
    # We retrive the stored value
    cached = self.states_stored[state_key]
    cached_value, cached_alpha, cached_beta = cached[0], cached
       [1], cached[2]
```

Listing 9: Repeated states checking implementation

Also, redundant calculations are avoided by checking if the current state's value are bounded by alpha and beta, this in order to save time and reducing the number of nodes that are evaluated:

```
# We know that the value is bounded by alpha
if cached_value == cached_alpha:
    alpha = cached_alpha

# We know that the value is bounded by beta
if cached_value == cached_beta:
    beta = cached_beta

# We know that the value is bounded by alpha and beta
if cached_value > cached_alpha and cached_value < cached_beta:
    return cached_value
```

Listing 10: Repeated states checking implementation

3 Answers to Assignment Questions

Answer the specific questions required by the assignment here, elaborating on the design choices, challenges, and improvements.

Q1: Describe the possible states, initial state, and transition function of the KTH fishing derby game.

Answer: The possible states in the KTH Fishing Derby game are defined by the following:

- Positions of the fishing hooks of each player.
- Positions of the fish in the 20x20 grid.
- The values associated to each fish.
- The index of caught fish by both players
- Current scores of both players of the fishing game.
- Game time

The initial state is characterized by:

- Both players and their hooks have predefined positions.
- The positions of the fish that are distributed on the grid.
- Scores for both players start as zero.
- 7 minutes and 30 seconds of game time.

The transition function $\mu(p, s)$ describes how a player's action (LEFT, RIGHT, UP, DOWN, STAY) changes the game state. This includes updating the position of the boat, moving the hook, and checking for fish caught or penalties.

Q2: Describe the terminal states of the KTH fishing derby game.

Answer: The terminal states are the following:

- When all of the fish on the grid have been caught, so no fish remain.
- The timer of the game reaches 0 seconds.

After a terminal state is met, the player with the higher score wins or the game is declared as a tie.

Q3: Why is $\nu(A, s) =$ Score(Green boat)$-$Score(Red boat) a good heuristic function for the KTH fishing derby?

Answer: The heuristic function $\nu(A, s)$ is acceptable because it represents the advantage that the player(in this case the green boat) over the opponent (in this case the red boat). And even though it's simple, it is a good way to evaluate the game state, by prioritizing the moves that increase the score that the player has and reducing the advantage of the opponent.

Q4: When does ν best approximate the utility function, and why?

Answer: The heuristic ν best approximates the utility function when:

- The score difference predicts the outcome of the game reliably (e.g., no high-value fish are left).
- There's not a lot of game time left, minimizing the impact of possible future actions on the scores.

And ν approximates the utility function well in these situations because of how much more predictable the game's outcome becomes, allowing the evaluation function to reflect the actual utility of the player reliably, with less uncertainty.

Q5: Provide an example of a state s where $\nu(A, s) > 0$ and B wins in the following turn.

Answer: An example state could be:

- The player has a higher score by catching multiple fish with lower values.
- The opponent has a high-value fish close to them, and catches it in the next turn, finally having a higher score than the player.

Q6: Will η suffer from the same problem as ν? If so, provide an example.

Answer: Yes, η can suffer from the same problem as ν, because it assumes that all the terminal states that are reachable have the same chances to happen, which is probably not the case against a difficult or experienced opponent.
Example: A heuristic η might suggest a favorable position for the player with a score of 10, given there are 2 fish left, one worth 1 point and one worth 5 points and the opponent has 6 points. And it makes this suggestion since it considers the terminal states of the player winning versus when the player loses, but it does not account for the threat of the opponent's next move.

4 Individual Reports

4.1 Sebastián

I mainly worked on the minimax algorithm with alpha-beta pruning, the move order, and some of the heuristics (which is something that we changed many times). I mainly followed the pseudo code and the execution examples that were used in class to implement the minimax algorithm, and since I am an exchange student and in my university back home we never used Python because we used mostly Java, this was a new experience for me but I did enjoy learning from it even if it took a lot of mistakes, using Python it was easier to follow and implement the code using inspiration from the pseudo-code than otherwise.

I also feel like I did learn more profoundly about the minimax algorithm by implementing and actually visualizing it myself, being able to play around with it, and experimenting with different ways of doing it until finding the best one really helped me understand the idea of it.

The move ordering was pretty simple to implement, but I did have problems making sure it was implemented correctly, since I didn't know how it should be sorted depending if it was the minimizing or the maximizing players we were dealing with, later on I found a solution that was documented previously.

Finally, when it came to the heuristics, there was a lot of trial and error on my part, but I also learned a lot from it since I could directly observe how the values of the heuristics affect the moves that the player makes, and with that, it was interesting to find out the best way to define a good heuristic function in order to find the best possible move to make.

4.2 Martín

I mainly worked with the heuristics, the implementation of Iterative Deepening Search, and the Repeated States Check. I started working a couple of days after Sebastián started working, so things like the minimax-alpha-beta pruning were already implemented, the current code by that time already achieved 7/25 points, so the problem was to improve the algorithm so it could reach deeper depth.

The hardest part for us was to find a good heuristic, we wanted one that maximized the $score - score\_opp$, but later we realized that we also had to have in mind the distance to the other fishes and their value. I remember that we had an entire work session just changing the heuristic function.

The IDS ended up being easy to implement, we made a for loop between all the possible depths we wanted to compute, and added a depth parameter in the minimax function to finish the recursive calls when the maximum depth was reached, this improved our score on Kattis to 12/25 points.

To achieve the > 20 score, we implemented the Repeated States Check, this allowed us to save a checkpoint, and whenever a state is happening again, it will be already precalculated in the States Set, to implement this we first tried to build a hash table, though this could have worked pretty well, it required more parser functions if we wanted to write a clear code. Then we decided to create a *state_to_key* that transforms a state into a string that can be used as a key for the States Set, this ended up working so well that we achieved 21/25.

After achieving 21 points, a good candidate for an A grade, we started thinking about how we could reach 25 and finally beat the opponent once and for all, we thought about making a better string to key function, since the one that we have maps each possible state to a key, we know that there are states that even though they look different, they can be solved with the same game moves. This looked promising, but given we also had other assignments, we will implement it another day.

Finally, we can say that this assignment helped us a lot to understand how the algorithms prior to Machine Learning work. We always are told about DeepBlue by IBM or AlphaGO by DeepMind, so through this assignment, our knowledge about game algorithms improved far better.