Search

Saly Al Masri

March 14, 2025

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

The possible states in the KTH fishing derby game are defined by the positions of the green and red boats, the positions of the fish on the grid, the scores of both players, and the remaining time in the game. The initial state consists of the starting positions of both boats, the initial positions of all the fish, and both players' scores set to zero. The transition function, denoted as $\mu(P, S)$, determines all the possible legal moves for a player P from a given state S. These moves include `LEFT`, `RIGHT`, `UP`, `DOWN`, and `STAY`, ensuring that the boats remain within valid positions on the grid and interact with the fish appropriately.

States

The game begins with several elements in specific positions. The green and red boats are positioned in their respective starting locations on the game board. Fishing hooks are also placed in their initial positions. Various fish are scattered around the board, swimming freely. Additionally, each player's score is tracked, and the current score for each player is available. A timer is also active, showing how much time remains for the game to be played.

Starting Point

At the start of the game, the boats are positioned at predefined spots on the board. Fishing hooks are aligned with their respective boats in starting positions. Fish are distributed randomly across the board, swimming in different areas. Every player begins with a score of zero, and the game timer is initialized to a specific amount of time, ready to count down as the game progresses.

Making Moves

During the game, players can control their boats to move in one of five possible directions: left, right, up, down, or stay in their current position. Whenever a boat moves, its attached fishing hook moves with it. If the hook successfully catches a fish, the player earns points for that catch, and the fish is removed from the board, contributing to the player's score. This sequence continues as players strategize to maximize their scores within the allotted time.

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

The terminal states of the KTH fishing derby game occur under two conditions. First, when there are no fish left on the grid, the game ends. Second, if the time limit expires, the game also concludes, regardless of the remaining fish or the players' scores.

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

The heuristic function $\nu(A, s) = \text{Score(Green boat)} - \text{Score(Red boat)}$ is effective because it directly reflects the relative advantage of the green boat (player A) over the red boat (player B). Since the goal of the game is to maximize the green boat's score while minimizing the red boat's score, this heuristic effectively guides the player toward states where their score difference is maximized. It provides a simple yet meaningful evaluation of the game's current state, helping the algorithm make decisions aligned with the objective.

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

The heuristic function ν best approximates the utility function near the end of the game. This is because, at that stage, most fish have already been caught, and the score difference becomes a more accurate reflection of the final outcome. In earlier stages of the game, ν may fail to account for strategic positioning or future opportunities, making it less representative of the overall utility.

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

An example of such a state would be where player A has a higher score than player B, indicating $\nu(A, s) > 0$. However, player B is strategically positioned to catch a high-value fish worth significant points in their next move. When B catches the fish, their score surpasses A's, leading to B's victory despite $\nu(A, s)$ suggesting A was in a better position.

example with numbers

- Green boat: 10 points
- Red boat: 8 points
- There's a big fish worth 5 points near the red boat and a small fish worth 1 point near the green boat.

The green boat catches the small fish, but then the red boat catches the big fish and wins! Even though the measurement said the green boat was ahead, the red boat ended up winning.

Q6: Will η suffer from the same problem as the evaluation function ν? If so, can you provide an example?

Yes, η suffers from the same problem as the evaluation function ν. This is because η, while evaluating based on reachable winning and losing states, does not account for the opponent's optimal strategy. **For example**, in chess, η might treat two moves equally favorable for the current player, even if one move leads to the loss of a queen in the opponent's next turn. Similarly, in the KTH fishing derby, η could overestimate a player's advantage if it does not fully anticipate the opponent's optimal actions.

Code Analysis: Alpha-Beta Pruning Implementation

1. Alpha-Beta Pruning Mechanism

Code

```
if beta <= alpha:
    break  # Prune
```

Explanation

This part implements the core pruning mechanism of the alpha-beta pruning algorithm. **Alpha-Beta Pruning**, this pruning is performed whenever the value of α exceeds or equals β, as **further exploration of such branches cannot improve the outcome for the current player**. This aligns with the ' explanation of pruning efficiency, where "$\beta \leq \alpha$" means no further exploration is necessary because the MIN player's upper bound cannot surpass the MAX player's lower bound. This concept is also reflected in the pseudocode from Chapter 5, which describes pruning as a method to eliminate irrelevant branches and reduce computational cost.

2. Depth Handling

Code

```
if depth == 0 or not possible_moves:
    return self.simple_evaluation(tree_node.state)
```

Explanation

This section terminates the recursion either when the fixed-depth search in adversarial games is reached or when no moves are available. Limiting the depth ensures computational feasibility while allowing the evaluation function to approximate the utility of terminal or near-terminal states. This is consistent with the pseudocode for depth-limited search and the heuristic minimax algorithm, which replaces exhaustive evaluation with heuristic-based evaluations at a cutoff depth.

3. State Caching (Transposition Table)

Code

```
state_key = str(tree_node.state.get_hook_positions()) +
    str(tree_node.state.get_fish_positions())
if state_key in self.cache:
    return self.cache[state_key]
```

Explanation

Caching previously evaluated states avoids redundant calculations, improving efficiency. Chapter 5 highlight the concept of **Transposition Tables**, which store previously encountered states along with their evaluations. This technique directly reduces the effective size of the game tree by allowing the algorithm to reuse results from identical or equivalent states. The example of "avoiding repeated calculations with hash functions" supports this implementation, as it ensures that identical placements or symmetric states are not recalculated, saving computational resources.

4. Move Ordering

Code

```
possible_moves.sort(key=lambda child: self.
    simple_evaluation(child.state), reverse=(player == "A
    "))
```

Explanation

Ordering moves based on heuristic evaluation improves pruning efficiency by exploring promising branches earlier. Chapter 5 specifically reference **Move Ordering** as a critical strategy in optimizing alpha-beta pruning. By prioritizing branches that are more likely to yield optimal outcomes, the algorithm can achieve deeper searches within the same computational budget. This aligns with the lecture pseudocode, which suggests sorting moves based on heuristic evaluations to maximize the impact of pruning.

5. Timeout Handling

Code

```
if time.time() - self.start_time > self.time_limit:
    raise TimeoutError
```

Explanation

This ensures the algorithm respects the time constraints specified in the assignment. This feature adheres to the "real-time requirements" implied by the **Iterative Deepening Search** approach, which balances depth and computational feasibility within a time budget. This mechanism ensures that the search process remains responsive, even in scenarios with high branching factors or complex state evaluations.

6. Player Turn Logic

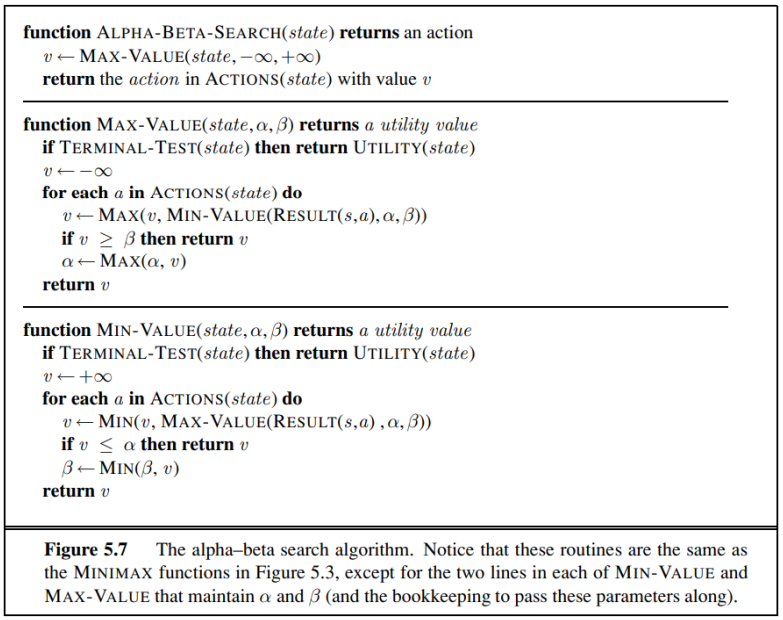
Code

```
if player == "A":
    best_value = max(
        best_value, self.initialize_model(child, depth -
            1, alpha, beta, "B")
    )
    alpha = max(alpha, best_value)
else:
    best_value = min(
        best_value, self.initialize_model(child, depth -
            1, alpha, beta, "A")
    )
    beta = min(beta, best_value)
```

Explanation

This alternates between maximizing (Player A) and minimizing (Player B) roles, consistent with the minimax principle. The tutorial describe this as the **Minimax Algorithm Basics**, where the MAX player seeks to maximize utility and the MIN player aims to minimize it. The alternating roles are mirrored in the pseudocode from Chapter 5, where the algorithm dynamically

updates α and β to reflect the best guaranteed outcomes for each player. The use of "max" and "min" functions ensures that the optimal strategy is selected at each level of the game tree.



```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Figure 1: chapter 5

Analysis of the `simple_evaluation` Function

1. Player Scores

Code

```
player_score, opponent_score = state.get_player_scores()
evaluation = player_score - opponent_score
```

Explanation

This segment calculates the difference in scores between the player (green boat) and the opponent (red boat).

- This aligns with tutorial on the **Utility Function**, $\psi(s, p)$, where the function evaluates the score difference, representing the advantage or disadvantage for one player in a zero-sum game.
- The assignment explicitly states (Objective Section) that a heuristic evaluation function should compare player scores to guide decision-making. The score difference serves as a baseline heuristic for evaluating the game state.
- This representation relates directly to the zero-sum property mentioned in the tutorial: *any gain for one player results in an equal loss for the other.*

2. Hook and Fish Positions

Code

```
hook_positions = state.get_hook_positions()
fish_positions = state.get_fish_positions()
fish_scores = state.get_fish_scores()
player = state.get_player()
```

Explanation

This segment retrieves essential state information, including:

- `get_player_scores`: To calculate the score difference.
- `get_hook_positions`: For evaluating the proximity of hooks to fish.

- `get_fish_positions` and `get_fish_scores`: To determine the value and location of fish.
- `get_player`: To identify the current player's turn.

These methods are critical for accessing the game tree state and align with the **perfect information game** property in the tutorial, where all game state information is visible and deterministic. The assignment emphasizes using these state representations to make heuristic evaluations.

The methods used here are defined in `game_tree.py` and are essential for accessing the game state. The README explicitly states (Objective section) that the solution should work with the game tree and its associated state, as the Fishing Derby game revolves around state manipulation.

3. Proximity-Based Scoring for Player

Code

```
for fish_id, fish_pos in fish_positions.items():
    hook_pos = hook_positions[player]
    distance = abs(hook_pos[0] - fish_pos[0]) + abs(
       hook_pos[1] - fish_pos[1])
    evaluation += fish_scores[fish_id] / (distance + 1)
        # Weighted by proximity
```

Explanation

This part evaluates the player's proximity to each fish and updates the score based on the fish's value. Key points:

- The Manhattan distance calculation aligns with chapter 3 on **heuristics**, where the goal is to approximate utility without fully computing the game outcome. Closer fish are weighted more heavily, reflecting the player's likelihood of capturing them.
- This heuristic evaluation mirrors the concept of prioritizing valuable game states (e.g., maximizing gains) in the **Minimax Algorithm Basics** section of your notes.
- The division by `(distance + 1)` ensures diminishing returns for farther fish, aligning with the assignment's goal of making efficient moves to maximize scoring potential.

4. Penalizing Opponent's Proximity

Code

```
opponent_hook_pos = hook_positions[1 - player]
for fish_id, fish_pos in fish_positions.items():
    opponent_distance = abs(opponent_hook_pos[0] -
       fish_pos[0]) + abs(opponent_hook_pos[1] -
       fish_pos[1])
    evaluation -= fish_scores[fish_id] / (
       opponent_distance + 1)
```

Explanation

This part calculates the opponent's proximity to each fish and penalizes the evaluation score if they are close. Key points:

- This aligns with the **Zero-Sum Game** principle in the tutorial, where a loss for the player (the opponent capturing valuable fish) is treated as an equivalent gain for the opponent.
- The logic reflects **optimizing the next move** by factoring in the opponent's likely actions, as highlighted in the tutorial: *"Assume the opponent is as smart as you."*
- Penalizing based on opponent proximity ensures that the evaluation discourages moves that might give the opponent an advantage, aligning with the assignment's emphasis on adversarial planning.

5. Returning the Evaluation

Code

```
return evaluation
```

Explanation

This final step consolidates all scoring updates into a single evaluation metric for the current game state.

- The assignment requires such a heuristic to make quick decisions, ensuring the algorithm operates efficiently within the time constraints.

- This result is then used by the higher-level Minimax or Alpha-Beta pruning algorithms to determine the optimal move, reflecting the recursive structure of **game tree traversal** described in your notes.

Function Analysis: `search_best_next_move`

Explanation

The `search_best_next_move` function employs iterative deepening with parallel computation for evaluating top-level moves. Below, each section of the code is explained with its relevance to the assignment.

1. Iterative Deepening

```
depth = 1
while True:
    ...
    depth += 1
```

Explanation: This iterative deepening mechanism incrementally increases the search depth until a limit (depth > 7 or time constraint) is reached. As stated in Section 3.3 of the assignment, achieving depths of at least 7 is required for grades B and A. Iterative deepening ensures that the algorithm explores as deeply as possible within the given time constraints.

2. Parallel Top-Level Move Evaluation

```
with ThreadPoolExecutor() as executor:
    futures = {
        executor.submit(
            self.initialize_model, child, depth, float("
               -inf"), float("inf"), "A"
        ): child.move
        for child in moves
    }
```

Explanation: Parallel processing of top-level moves optimizes time efficiency, which is critical as per Section 3.2 of the assignment. This approach ensures multiple moves are evaluated simultaneously, allowing for faster traversal of the game tree.

3. Alpha-Beta Pruning

```
self.initialize_model(child, depth, float("-inf"), float
   ("inf"), "A")
```

Explanation: The `initialize_model` function implements alpha-beta pruning (Algorithm 2 in the assignment). This optimization reduces the effective branching factor of the game tree, as described in Section 2.4, enabling deeper exploration.

4. Timeout Handling

```
if depth > 7 or time.time() - self.start_time > self.
   time_limit:
    break
```

Explanation: This ensures the algorithm operates within the time constraints mentioned in assignment . A timeout mechanism is essential for real-time gameplay scenarios.

5. Move Selection Logic

```
if value > best_value:
    best_value = value
    best_move = move
```

Explanation: This logic identifies the move with the highest heuristic value, which aligns with the goal of optimizing player A's (green boat's) chances of winning, as outlined in Section 3.1.

Code Analysis: `search_best_next_move`

1. Child Node Expansion

Code

```
initial_tree_node.compute_and_get_children()
```

Explanation

This line generates the child nodes for the current game tree node, representing all possible game states resulting from Player 0's legal moves. **Dynamic Node Expansion** ensures that only relevant parts of the game tree are explored, reducing unnecessary computation. This is consistent with the concept of lazy evaluation in the **Game Tree Representation** section of the lectures, where nodes are generated on demand during the search process. The assignment emphasizes evaluating possible actions using such an expandable tree structure, as outlined in the problem statement.

2. Depth and Alpha-Beta Initialization

Code

```
depth = 4  # adjust
alpha = float("-inf")
beta = float("inf")
player = "A"
```

Explanation

Here, the search depth is limited to 4 levels to ensure computational feasibility. α and β represent the best guaranteed scores for Player 0 (MAX) and Player 1 (MIN), respectively. Limiting depth is critical for balancing accuracy and efficiency, as explained in the **Depth-Limited Search** section of the lectures. The use of α and β aligns with the optimization techniques discussed in the **Alpha-Beta Pruning** section. The assignment explicitly requires these parameters to improve the efficiency of the Minimax algorithm.

3. Time Tracking and State Caching

Code

```
self.start_time = time.time()
self.cache = {}
```

Explanation

Tracking time ensures that the algorithm adheres to the 75ms time constraint specified in the assignment. The cache stores previously evaluated states, avoiding redundant calculations and improving efficiency. This implementation aligns with the **Real-Time Constraints** discussed in the lectures and the **Repeated States Checking** concept in the tutorial slides. The assignment instructions explicitly stress efficiency within strict time limits, making caching a crucial optimization.

4. Evaluating Possible Moves

Code

```
move_values = {}
for child in initial_tree_node.compute_and_get_children
    ():
     move = child.move
     value = self.initialize_model(child, depth - 1,
        alpha, beta, "B")
     move_values[move] = value
```

Explanation

This loop iterates over all possible moves (child nodes) and evaluates their utility using the Minimax algorithm with Alpha-Beta pruning. Each move's value is stored in a dictionary for later comparison. This step directly implements the **Minimax Algorithm Basics** as described in the lectures, where child nodes represent potential game states evaluated to determine the best strategy. The assignment explicitly requires evaluating actions for Player 0 in this manner.

5. Selecting the Best Move

Code

```
best_move = max(move_values, key=move_values.get)
```

Explanation

This selects the move with the highest evaluated utility, ensuring that Player 0 (green boat) makes the optimal choice. This process follows the **Maximization Principle** of the Minimax algorithm, as discussed in the lectures. Selecting the optimal move is the core objective of the assignment, aligning with its requirement to implement Minimax for decision-making.

6. Timeout Handling

Code

```
except TimeoutError:
    best_move = random.choice(["stay", "left", "right",
        "up", "down"])
```

Explanation

This ensures that the algorithm respects the 75ms time constraint by selecting a random move when a timeout occurs. This fallback mechanism adheres to the **Real-Time Requirements** discussed in the tutorial, where time-bound decision-making is critical. The assignment emphasizes handling time constraints efficiently, making this implementation essential for compliance.

7. Returning the Move

Code

```
return ACTION_TO_STR[best_move]
```

Explanation

This converts the chosen move into a human-readable string (e.g., "left", "right") for compatibility with the game interface. This step ensures proper integration with the Fishing Derby environment, aligning with the assignment's requirement to output the best possible move in a usable format.

Reference to Assignment

This section explains how each part of the provided code fulfills the requirements of the assignment for grade E and relates to the lectures and tutorial materials on search algorithms.

1. The `initialize_model` Function

This function implements the **Minimax algorithm with Alpha-Beta pruning** to evaluate possible game states and find the optimal move for the green player.

- **Timeout Handling:**

```
if time.time() - self.start_time > 0.075:
    raise TimeoutError
```

  Ensures the algorithm respects real-time constraints, crucial for decision-making in games. This fulfills the assignment's requirement for practical implementation under computational limits.
  Lecture Reference: This approach is aligned with the discussion of real-time constraints in the *Iterative Deepening Search (IDS)* section of the tutorial slides.

- **State Caching:**

```
if state_key in self.cache:
    return self.cache[state_key]
```

  Reduces computational redundancy by avoiding re-evaluation of states. This addresses repeated state checking as required in the assignment.
  Lecture Reference: This concept is emphasized in the *Repeated States and Hashing* section of the lecture slides.

- **Base Case (Depth Limit or Terminal State):**

```
if depth == 0 or not possible_moves:
    return self.simple_evaluation(tree_node.state)
```

Implements depth-limited search to constrain the depth of the game tree exploration.
Lecture Reference: Depth-limited search is covered in the *Search Tutorial Slides* as a method to handle large state spaces.

- **Alpha-Beta Pruning Logic:**

```
if player == "A":
    best_value = float("-inf")
else:
    best_value = float("inf")

for child in possible_moves:
    if player == "A":
        best_value = max(
            best_value,
            self.initialize_model(child, depth - 1, alpha, beta, "B")
        )
        alpha = max(alpha, best_value)
    else:
        best_value = min(
            best_value,
            self.initialize_model(child, depth - 1, alpha, beta, "A")
        )
        beta = min(beta, best_value)
    if beta <= alpha:
        break
```

Optimizes the Minimax search by pruning irrelevant branches of the game tree.
Lecture Reference: The concept of pruning branches to reduce computational overhead is detailed in the *Alpha-Beta Pruning* section of the slides.

2. The `simple_evaluation` Function

This function provides a basic heuristic to estimate the utility of a given state.

```
def simple_evaluation(self, state):
    player_score, opponent_score = state.get_player_scores()
    return player_score - opponent_score
```

- **Purpose:** Approximates the utility function, fulfilling the heuristic evaluation requirement for grade E.
- **Lecture Reference:** The *Heuristic Functions* section in the lecture discusses simple heuristics like $\nu(A, s) = \text{Score(Green)} - \text{Score(Red)}$, which aligns directly with this implementation.

3. The `search_best_next_move` Function

This function orchestrates the decision-making process to select the best move using `initialize_model`.

- **Depth-Limited Search:**

  ```
  depth = 4
  ```

  Constrains the depth of the search to balance accuracy and computational efficiency, meeting the assignment's grade E requirement.

- **Iterating Over Possible Moves:**

  ```
  for child in initial_tree_node.compute_and_get_children():
      move = child.move
      value = self.initialize_model(child, depth - 1, alpha, beta, "B")
      move_values[move] = value
  ```

  Evaluates all legal moves and determines their respective utility values.
  Lecture Reference: This aligns with the exploration of the game tree as explained in the *Search and Games Lecture Slides*.

- **Best Move Selection:**

```
best_move = max(move_values, key=move_values.get)
```

Selects the move with the highest evaluated utility. This directly implements the decision-making aspect required in the assignment.

- **Timeout Handling:**

```
except TimeoutError:
    best_move = random.choice(["stay", "left", "right", "up", "down"])
```

Ensures graceful handling of time constraints, preventing program failure.
Assignment Justification: Prevents issues under strict deadlines, aligning with real-time game requirements.