CSE 202 Project Algorithm Analysis
Winter 2025

Pooja Sounder Rajan, Anusha Ravichandran, Nevasini Sasikumar, Akhila Yekalluri, Viroopaksh

March 4 2025

1 Introduction

The Hearts card game is a trick-taking game where players aim to minimize their penalty points by strategically playing their cards. This project presents an algorithmic approach to analyzing Hearts, specifically utilizing a Greedy Strategy to optimize play decisions. The game has been modified to generalize the number of suits and cards per suit while incorporating a penalty-based scoring mechanism.

1.1 Modifications to Standard Rules

To expand the problem scope, we introduce the following modifications to the traditional Hearts game:

- The deck consists of k suits (where $k \geq 2$) and n cards per suit (where $n \geq 2$).
- Hearts are always designated as suit 0, and each heart played results in a penalty of 1 point.
- The traditional "Queen of Spades" rule is modified:
  - Instead of the Queen of Spades, the $(n-1)$th card of suit 1 carries a penalty of n points when won in a trick.

1.2 Objective and Approach

The primary goal for each player is to minimize penalty points by making strategic card choices. We analyze the game using a Greedy Algorithmic Approach, where each move is chosen based on immediate benefits while adhering to the legal rules of play. The analysis will focus on the following:

- **Game State Representation**: Formally defining the elements that track the game's progress.
- **Move Selection Strategy**: Evaluating legal moves and selecting the most favorable move using a greedy heuristic.
- **Penalty Optimization**: Assessing how different strategies impact score minimization over multiple rounds.

2 State Formalization

To implement an efficient strategy, the game state is represented using structured data models and mathematical notations. The state encapsulates all relevant information about the ongoing game, including hands, played cards, scores, and turn orders.

2.1 Key Game Components

The game state consists of the following elements:

- **History of Played Cards** (T): Tracks all cards played so far in the game.
  - **Data Structure**: Dictionary mapping suits to a list of played card values.
  - **Example**: $T = \text{Hearts} : [2, 4], \text{Spades} : [10, Q]$.
- **Current Hands** (H): The set of cards currently held by each player.

  - **Data Structure**: A list of sets, where each set contains the cards in a player's hand.
  - **Example**: $H_1 = (0,3),(2,7),(3,11)$.

- **Current Trick** (T_k): The cards played in the ongoing trick.
  - **Data Structure**: A list of tuples representing cards in the trick.
  - **Example**: $T_k = [(1,5),(3,10)]$.

- **Lead Suit** (L_k): The suit that was led in the current trick.
  - **Data Structure**: Integer value representing the suit.
  - **Example**: $L_k = 1$ (indicating suit 1).

- **Turn Order** ($f : P \to P$): Determines the next player based on trick-winning rules.
  - **Example**: If Player 2 wins the trick, they lead the next round.

- **Scores** (S): Tracks the penalty points for each player.
  - **Data Structure**: A list of tuples mapping players to their current score.
  - **Example**: $S = [(P_1,4),(P_2,10),(P_3,6),(P_4,0)]$.

2.2 Formal Mathematical Representation

We define the game formally using the following notation:

- **Players**: $P = P_1, P_2, P_3, P_4$.
- **Deck**: D contains all $k \times n$ cards.
- **Player Hands**: $H = (H_1, H_2, H_3, H_4)$ where $H_i \subseteq D$.
- **Trick History**: $T = (T_1, T_2, ..., T_m)$ (m tricks played so far).
- **Lead Suit**: $L = (L_1, L_2, ..., L_m)$.
- **Trick Winner**: W_k represents the winner of trick k.
- **Penalty Scores**: $S = (S_1, S_2, S_3, S_4)$.
- **Turn Function**: $f : P \to P$ determines the next player in sequence.

By structuring the game state in this manner, we establish a foundation for designing efficient move selection strategies using the Greedy Approach.

2.3 Modified Objective Function

Each player P_i seeks to minimize their total penalty score:

$$\text{score}(P_i) = \sum \text{heart\_points} + \text{penalty\_card\_points} \tag{1}$$

where:

- **heart_points**: Every card from Suit 0 (Hearts) taken contributes 1 penalty point.
- **penalty_card_points**: The $(n-1)$th card of Suit 1 carries a penalty of n points if won in a trick.

3 Algorithms, Pseudocodes and Descriptions

3.1 Data Structures

- **Card**: Structure containing:
  - `suit` (integer)
  - `value` (integer)
- **Player**: Structure containing:
  - `player_id` (integer)
  - `hand` (set of Card objects)
  - `score` (integer)
- **Game State**: Structure containing:
  - `players`: List of Player objects
  - `trick_history`: List of completed tricks
  - `current_trick`: List of cards in the current trick
  - `lead_suit`: Suit of the first card in the current trick
  - `turn_player`: Index of the player whose turn it is
  - `trick_starter`: Index of the player who started the current trick
  - `hearts_broken`: Boolean indicating if hearts have been played
  - `special_card_played`: Boolean indicating if the special penalty card has been played
  - `special_card`: The $(n-1)$th card of suit 1, equivalent to the Queen of Spades

4 Pseudo Code

4.1 Game Initialization

Algorithm 1 InitializeGame

```
function InitializeGame(num_suits, cards_per_suit, num_players)
    players ← new List of Player objects (size: num_players)
    deck ← new List of all possible Card combinations (size: num_suits × cards_per_suit)
    Shuffle(deck)
    cards_per_player ← (num_suits × cards_per_suit) / num_players
    for $i \leftarrow 0$ to $num\_players - 1$ do
        Assign cards from deck indices [$i \times$cards_per_player] to [$(i+1) \times$cards_per_player$-1$] to players[$i$].hand
    end for
    special_card ← new Card(suit: 1, value: cards_per_suit - 1)
    for $i \leftarrow 0$ to $num\_players - 1$ do
        if Card(1, 2) is in players[$i$].hand then
            turn_player ← $i$
            trick_starter ← $i$
            break
        end if
    end for
    trick_history ← empty list
    current_trick ← empty list
    hearts_broken ← false
    special_card_played ← false
end function
```

Purpose:

Sets up the initial state of the game before play begins.

Details:

- Creates a full deck of cards based on the specified parameters (`num_suits` and `cards_per_suit`)
- Shuffles the deck randomly to ensure unpredictable card distribution
- Distributes cards evenly among all players, with each player receiving $\frac{num\_suits \times cards\_per\_suit}{num\_players}$ cards
- Identifies the player holding the $(1, 2)$ card (equivalent to 2 of Clubs in standard Hearts), as this player must lead the first trick
- Initializes game state variables:
  - Sets `trick_history` to an empty list to track completed tricks
  - Sets `current_trick` to an empty list to hold cards being played in the active trick
  - Sets `hearts_broken` to false, as no Hearts have been played yet
  - Sets `special_card_played` to false, tracking whether the special penalty card has been played
  - Defines the `special_card` as the $(n - 1)$th card of suit 1

The function effectively creates a complete game environment with proper initial conditions according to the modified Hearts rules.

4.2 GetLegalMoves

Algorithm 2 Legal Moves

```
function GetLegalMoves(player_index)
    set player = players[player_index]
                                                ▷ First trick, first player special rule
    if trick_history is empty and current_trick is empty then
        return [Card(1, 2)]
    end if
                                                ▷ If leading a trick
    if current_trick is empty then
        set legal_moves = all cards in player.hand
                                                ▷ Hearts cannot be led until broken or only hearts remain
        if not hearts_broken and not special_card_played then
            set non_heart_cards = cards in legal_moves where suit $\neq 0$
            if non_heart_cards is not empty then
                set legal_moves = non_heart_cards
            end if
        end if
        return legal_moves
    end if
                                                ▷ If following a trick
    set lead_suit = current_trick[0].suit
    set suit_cards = cards in player.hand where suit == lead_suit
    if suit_cards is not empty then
        return suit_cards
    else                                        ▷ Can play any card if cannot follow suit
        return all cards in player.hand
    end if
end function
```

Purpose:

Determines which cards a player can legally play on their turn, following the game's rules.

Details:

- Handles special case for the first trick:
  - If it's the very first play of the game, only the $(1, 2)$ card can be played. There is a general rule of Hearts game to be started with a 2 of Clubs. However here, as we have a lower limit of suits as 2, we shall start everytime with the 2 Card of Suit 1.
- For leading a trick (when `current_trick` is empty):
  - Returns all cards in the player's hand by default
  - Enforces the rule that Hearts cannot be led until either:
    - Hearts have been broken (a Heart was played in a previous trick)
    - The special penalty card has been played
    - The player only has Hearts remaining
- For following a trick:
  - Identifies the lead suit from the first card played in the current trick
  - If the player has cards of the lead suit, only those cards can be played (must follow suit)
  - If the player has no cards of the lead suit, any card from their hand can be played

This function ensures all game rules regarding card playing are strictly enforced, preventing illegal moves.

4.3 Greedy Card Evaluation

Algorithm 3 EvaluateCard

```
function EVALUATECARD(card, player_index, legal_moves)
    set score = 0
                                                        ▷ Avoid taking Hearts (suit 0)
    if card.suit == 0 then
        decrement score by 10
    end if
                                        ▷ Especially avoid taking the special penalty card
    if card == special_card then
        decrement score by 50
    end if
                                    ▷ If following and can't win trick, dump high-value cards
    if current_trick is not empty and card.suit $\neq$ current_trick[0].suit then
        if card.suit == 0 then
            increment score by 20
        end if
        if card == special_card then
            increment score by 100
        end if
    end if
                                        ▷ If following the led suit and might win the trick
    if current_trick is not empty and card.suit == current_trick[0].suit then
        set would_win = true
        for all played_card in current_trick do
            if played_card.suit == card.suit and played_card.value > card.value then
                set would_win = false
                break
            end if
        end for
        if would_win then                       ▷ Check if winning would give us penalty points
            set penalty = 0
            for all played_card in current_trick do
                if played_card.suit == 0 then
                    increment penalty by 1
                end if
                if played_card == special_card then
                    increment penalty by cards_per_suit
                end if
            end for
            decrement score by (penalty $\times$ 15)
        else                                        ▷ Not winning, so playing high cards is good
            increment score by card.value
        end if
    end if
                                                                ▷ Leading a trick strategy
    if current_trick is empty then
        if card.suit $\neq$ 0 then
            increment score by (cards_per_suit - card.value)
        end if
        if card.suit == special_card.suit and not special_card_played then
            decrement score by 5
        end if
    end if
    return score
end function
```

Purpose:

Implements the greedy strategy by assigning a numerical score to each potential move, where higher scores indicate more desirable moves.

Details:

- Base score starts at 0 for each card
- Penalty avoidance considerations:
  - Decreases score by 10 for Hearts (suit 0) as they carry penalty points
  - Decreases score by 50 for the special penalty card, making it highly undesirable
- Strategy when following but cannot win (different suit than lead):
  - Increases score for penalty cards (Hearts and special card) to encourage discarding them
  - Hearts get +20 to their score
  - Special penalty card gets +100 to its score, making it the most preferable to discard
- Strategy when following with the led suit:
  - Evaluates if the card would win the trick by comparing it to other cards played
  - If the card would win the trick:
    - Calculates potential penalties from winning (Hearts and special card)
    - Decreases score based on penalties (penalty $\times$ 15) to avoid winning tricks with penalties
  - If the card would not win the trick:
    - Increases score based on card value, favoring playing higher cards when losing
- Strategy when leading a trick:
  - Prefers leading low cards of non-Heart suits (score += cards_per_suit - value)
  - Avoids leading the suit of the special card if it hasn't been played yet (score -= 5)

This sophisticated evaluation function considers multiple strategic factors to make intelligent decisions about which card to play.

4.4 Greedy Strategy Selection

Algorithm 4 GreedyPlay

```
1: function GreedyPlay(player_index)
2:     player ← players[player_index]
3:     legal_moves ← GetLegalMoves(player_index)
4:     best_card ← null
5:     best_score ← negative infinity
6:     for each card in legal_moves do
7:         score ← EvaluateCard(card, player_index, legal_moves)
8:         if score > best_score then
9:             best_score ← score
10:            best_card ← card
11:        end if
12:    end for
13:    return best_card
14: end function
```

Purpose:

Uses the evaluation function to select the best card to play from all legal options.

Details:

- Gets the list of legal moves for the player using `GetLegalMoves`
- Initializes tracking variables for the best card and its score
- Iterates through each legal card:
  - Calls `EvaluateCard` to get a score for each card
  - Updates the best card if the current card's score is higher than the previous best
- Returns the card with the highest evaluation score, representing the locally optimal move

This function embodies the core of the greedy algorithm approach, always selecting the move that appears most beneficial based on current game state information.

4.5 Play Card Function

Algorithm 5 PlayCard

```
function PLAYCARD(card, player_index)
    player ← players[player_index]
    if card is not in GetLegalMoves(player_index) then
        Raise Error("Illegal move")
    end if
    Remove card from player.hand
    Add card to current_trick
    if card.suit == 0 then
        hearts_broken ← true
    end if
    if card == special_card then
        special_card_played ← true
    end if
    turn_player ← (player_index + 1)   mod  num_players
    if length of current_trick == num_players then
        Call CompleteTrick()
    end if
end function
```

Purpose:

Executes a selected card play and updates the game state accordingly.

Details:

- Validates that the card is a legal move by checking against `GetLegalMoves`
- Updates the player's hand by removing the played card
- Adds the card to the current trick
- Updates game state flags:
  - If a Heart is played, sets `hearts_broken` to true
  - If the special penalty card is played, sets `special_card_played` to true
- Advances the turn to the next player (`turn_player = (player_index + 1) % num_players`)
- Checks if the trick is complete (all players have played a card):
  - If complete, calls `CompleteTrick` to resolve the trick

This function handles the mechanics of playing a card and maintains the integrity of the game state after each move.

4.6 Complete Trick Function

Algorithm 6 CompleteTrick

```
function CompleteTrick
    lead_suit ← current_trick[0].suit
    winner_card ← current_trick[0]
    winner_index ← 0
    for $i$ ← 1 to length of current_trick - 1 do
        if current_trick[i].suit == lead_suit and current_trick[i].value ¿ winner_card.value then
            winner_card ← current_trick[i]
            winner_index ← $i$
        end if
    end for
    winner_player_index ← (trick_starter + winner_index)   mod   num_players
    for each card in current_trick do
        if card.suit == 0 then
            players[winner_player_index].score ← score + 1
        end if
        if card == special_card then
            players[winner_player_index].score ← score + cards_per_suit
        end if
    end for
    Add current_trick to trick_history
    current_trick ← empty list
    trick_starter ← winner_player_index
    turn_player ← winner_player_index
end function
```

Purpose:

Resolves a compelted trick by determining the winner and updating scores.

Details:

- Identifies the lead suite from the first card in the trick.
- Initializes the winner as the player who played the first card
- Finds the true winner by finding the highest card of the lead suit:
  - Compares each card in teh trick to the current winner.
  - Updates the winner if a higher card of the lead suit is found.
- Calculates the actual player index of the winner (adjusting from the trick order to player index)
- Updates the winner's score based on penalty cards in the trick:
  - +1 point for each Heart (Suit 0).
  - +n points ($cards_per_suit$) for the special penalty card.
- Updates game state for the next trick:
  - Adds the completed trick to the trick history.
  - Clears the current trick list.
  - Sets the trick starter to the winner of this trick.
  - Sets the turn player to the winner of this trick(winner leads next trick)

This function ensures that tricks are properly scored according to the game rules and sets up the correct turn order for the next trick.

4.7 Play Full Round

Algorithm 7 PlayRound

```
function PlayRound
    set total_tricks = (num_suits $\times$ cards_per_suit) / num_players
    while length of trick_history < total_tricks do
        set current_player = turn_player
        set card = GreedyPlay(current_player)
        call PlayCard(card, current_player)
    end while
    return scores of all players
end function
```

Purpose:

Manages the flow of the game by playing a complete round until all cards have been played.

Details:

- Calculates the total number of tricks in a round: $\frac{num\_suits \times cards\_per\_suit}{num\_players}$
- Continues playing tricks until all tricks have been played:
  - Identifies the current player
  - Calls `GreedyPlay` to determine the best card for that player
  - Calls `PlayCard` to play the selected card
  - Prints game state information after each completed trick for monitoring
- When all tricks are complete, prints final scores
- Returns the final scores of all players

This function orchestrates the entire gameplay process, using all the other functions to execute a complete round of the modified Hearts game using the greedy strategy.

5 Conclusion

Each of these functions combines to create a complete implementation of the modified Hearts game with a greedy algorithmic approach to strategy, where decisions are made based on the immediate benefit of each move rather than considering long-term outcomes. The evaluation function takes into account multiple factors to make intelligent decisions about which card to play in any given situation, while the game mechanics functions ensure that the rules of the game are properly enforced and the game state is correctly maintained.

6 Proof of Correctness

In this section, we establish the correctness of our algorithm for the modified Hearts game. To ensure a rigorous proof, we analyze two fundamental properties:

- **Correctness:** The algorithm produces valid outputs that adhere to the game rules.
- **Termination:** The algorithm completes execution in a finite number of steps.

To prove correctness, we will employ two complementary techniques:

- **Mathematical Induction:** This method will demonstrate that the algorithm always selects a legal move and maintains a valid game state at every stage.
- **Loop Invariants:** These will ensure that essential game properties, such as turn order, legal move enforcement, and score conservation, hold throughout the game.

By combining these approaches, we can rigorously confirm that the algorithm correctly executes each stage of the game while adhering to the prescribed rules and optimizing penalty minimization.

6.1 Correct Initialization

The game starts with:

- A shuffled deck of $k \times n$ cards.
- An equal distribution of cards among all players.
- The correct identification of the special penalty card.
- The correct assignment of the first lead player (who has the card $(1, 2)$, the 2 of suit 1).

Correctness Argument:

- The deck contains all unique cards, ensuring a well-formed initial game state.
- Each player receives exactly $\frac{k \times n}{4}$ cards, ensuring fairness.
- The search for $(1, 2)$ in players' hands guarantees the first player is correctly selected.

Thus, the initialization correctly follows the game constraints.

6.2 Legal Moves Enforcement

At every turn, the `GetLegalMoves(player_index)` function:

- Enforces the first trick rule (the first player must play $(1, 2)$).
- Ensures players follow suit if possible.
- Allows off-suit plays only when the player has no matching suit cards.
- Prevents leading with hearts until hearts are broken.

Correctness Argument:

- Since every trick begins with a valid lead (either by rule or because no other suit is available), all plays conform to the rules.
- The function ensures that if a player has the lead suit, they must play it, preventing invalid moves.

Thus, all generated moves are legal.

6.3 Trick Resolution and Score Updating

The highest card of the lead suit wins the trick. The trick winner starts the next round. Scores are updated correctly when a player wins penalty cards.
Correctness Argument:

- The comparison operation ensures the highest-ranked card of the lead suit wins.
- The `turn_player` is updated to reflect the new leader.
- The penalty is correctly assigned based on the game's modified rules.

Thus, trick resolution maintains correctness.

6.4 Game Termination

A game consists of $\frac{k \times n}{4}$ rounds (one per card per player). The number of tricks is finite and decreases with every round. No infinite loops are possible, as all players eventually run out of cards.
'Thus, the algorithm terminates in at most $\frac{k \times n}{4}$ rounds.

7 Proof by Induction on the Number of Tricks Played

We prove by induction on the number of tricks, T, that:

- The algorithm always selects a legal move.
- The algorithm correctly updates the game state.
- The greedy heuristic minimizes penalty score.

7.1 Base Case ($T = 1$)

The game starts with the $(1, 2)$ card (i.e., 2 of suit 1). The algorithm enforces this explicitly (InitializeGame() ensures that the first player starts correctly).
Other players follow the rules:

- If they have cards of the same suit, they must play them.
- Otherwise, they can play any card.

This follows the standard game rules, so the first trick is guaranteed to be legal. Thus, the algorithm correctly handles the first trick.

7.2 Inductive Hypothesis

Assume that after m tricks, the algorithm:

- Ensures all moves made are legal.
- Maintains a valid game state (turn order, trick updates, scorekeeping).
- Ensures that players follow a greedy heuristic to minimize penalty points.

7.3 Inductive Step ($T = m + 1$)

We show that the next trick also follows these properties.

7.3.1 Case 1: Normal Trick Play (No Hearts Broken)

The leading player must play a card:

- If they have no choice (only one card in suit), they play it.
- Otherwise, they choose the least dangerous card.

Other players must follow suit if possible (GetLegalMoves(player) guarantees this). If a player cannot follow suit:

- The algorithm ensures they can play any other card.
- Hearts cannot be led unless already broken.

The trick winner is determined correctly and updated. Thus, the $(m+1)$th trick is played legally and optimally.

7.3.2 Case 2: Breaking Hearts

Hearts cannot be led unless broken. If a player is forced to play a heart (due to no other suits), the algorithm updates the game state to indicate that hearts are broken. Future tricks are now allowed to lead with hearts. Thus, the game state correctly transitions when hearts are broken.

7.3.3 Case 3: Handling High-Penalty Cards (Queen of Spades, Hearts)

If a player must take a trick, they will attempt to minimize their penalty. The algorithm ensures that:

- A player with unavoidable high-penalty cards (e.g., Queen of Spades) plays them at the best moment (typically when an opponent is forced to take the trick).

This ensures the greedy strategy of minimizing penalty points. Thus, penalties are assigned correctly, and players minimize their losses.
Since the $(m + 1)$th trick follows the game rules and maintains optimal play, the algorithm remains correct.

8 Proof Using Loop Invariants

We define loop invariants that hold at every stage of execution.

8.1 Invariant 1: Turn Order Validity

Claim: At any time during the game, the player who plays a card is the correct one according to:

- The initial lead.
- The winner of the previous trick.

Proof: The function turn_function($f : P \to P$) correctly assigns turns. Each trick determines a unique winner. The algorithm always starts the next trick with the previous trick's winner.
Conclusion: This invariant holds throughout the game.

8.2 Invariant 2: Legal Move Enforcement

Claim: At any point, every player only plays a legal move.
Proof: GetLegalMoves(player) ensures:

- If a player has the lead suit, they must follow suit.
- Otherwise, they can play any card.
- Hearts cannot be led until broken.

The algorithm never allows an illegal move. Since this holds from the first trick onward, it remains valid throughout.
Conclusion: This invariant always holds.

8.3 Invariant 3: Score Conservation

Claim: The sum of penalty points in the game is equal to the number of penalty cards played.
Proof: The algorithm correctly tracks penalties at every step. Each trick assigns penalty points based on captured cards. No extra penalty points are assigned or removed incorrectly.
Conclusion: The penalty score remains valid at all times.

9 Additional Scenarios to Validate Correctness

To further reinforce correctness, we analyze additional edge cases.

9.1 Scenario 1: A Player Runs Out of a Suit Early

If a player runs out of a suit early:

- They can play any other card.
- The algorithm correctly allows hearts to be broken if necessary.
- The correct trick winner is computed.

This ensures the game remains legal.
Conclusion: The algorithm correctly handles missing suits.

9.2 Scenario 2: A Player is Forced to Take a High-Penalty Trick

If a player must take a trick:

- The algorithm ensures they choose a move that minimizes total penalty.
- Example: If a player must take a trick containing the Queen of Spades, they try to minimize hearts gained.

This ensures the greedy strategy works.
Conclusion: The algorithm minimizes penalties optimally.

9.3 Scenario 3: The Last Trick

The final trick is handled identically to the rest:

- Players follow suit.
- The winner is determined normally.
- The game correctly computes final scores.

Conclusion: The game correctly terminates.

10 Final Conclusion

Since we have:

- Inductively proven that every trick follows game rules.
- Maintained loop invariants ensuring correct turn order, move legality, and scoring.
- Handled all edge cases to ensure correct game behavior.

We conclude that the algorithm always produces a correct sequence of moves while minimizing penalty points as per the greedy heuristic.

11 Time Complexity Analysis Without Greedy Approach

If the algorithm does not use a greedy approach, it must evaluate all possible move sequences instead of selecting a locally optimal move at each step. This results in a combinatorial explosion in complexity.

1. Move Selection Without Greedy Strategy

- Instead of choosing a single best move, the algorithm explores all legal moves at each step.
- If each player has h legal moves and there are p players, the number of possible move sequences grows exponentially:

  $$O(h^{t \cdot p})$$

  where t is the number of tricks per round.

2. Evaluation of Move Sequences

- Each move sequence must be evaluated for scoring or decision-making.
- Assuming backtracking or exhaustive search, the worst-case complexity becomes:

  $$O(h^{t \cdot p} \cdot t)$$

3. PlayRound Complexity

- Without greedy, each round expands to:

  $$O(h^{t \cdot p} \cdot t)$$

Comparison

Approach	**Time Complexity**
Greedy Approach	$O(t \cdot h \cdot p)$
Exhaustive Search (No Greedy)	$O(h^{t \cdot p} \cdot t)$

The non-greedy approach results in an exponential time complexity, making it infeasible for large values of h, t, p.

12 Time and Space Complexity Analysis of Algorithms in the Hearts Game with greedy approach

This document provides the time and space complexity analysis of various algorithms used in the modified Hearts card game.

1. InitializeGame Algorithm

- **Time Complexity:**
  - Creating player objects and initializing structures: $O(p)$, where p is the number of players.
  - Generating and shuffling the deck: $O(kn)$, where k is the number of suits, and n is the number of cards per suit.
  - Distributing cards to players: $O(kn)$.
  - Identifying the first player with the special card: $O(kn)$.
  - **Total Complexity:** $O(kn)$.
- **Space Complexity:**
  - Storage for player objects: $O(p)$.
  - Storage for the deck: $O(kn)$.
  - Storage for all player's hand: $O(kn)$.
  - **Total Complexity:** $O(kn)$.

2. GetLegalMoves Algorithm

- **Time Complexity:**
  - Checking special conditions for the first move: $O(1)$.
  - Iterating over the player's hand to determine valid moves: $O(h)$ (i.e h is the number of cards in the player hand)
  - **Total Complexity:** $O(h)$, where h is the number of cards in the player's hand.
- **Space Complexity:**
  - Temporary storage for legal moves: $O(h)$.
  - **Total Complexity:** $O(h)$.

3. EvaluateCard Algorithm

- **Time Complexity:**
  - Evaluating a single card: $O(1)$.
  - Checking penalty and trick-winning conditions: $O(1)$.
  - Iterating over the current tricks: $O(t)$ t is the number of tricks played till now.
  - **Total Complexity:** $O(t)$.
- **Space Complexity:**
  - Temporary storage for evaluating a card: $O(1)$.
  - **Total Complexity:** $O(1)$.

4. GreedyPlay Algorithm

- **Time Complexity:**
  - Calling `GetLegalMoves`: $O(h)$.
  - Iterating through all legal moves: $O(h)$.
  - Evaluating each move using `EvaluateCard`: $O(t)$.
  - **Total Complexity:** $O(h * t)$.
- **Space Complexity:**

  - Storage for legal moves: $O(h)$.
  - Temporary storage for best card selection: $O(1)$.
  - **Total Complexity:** $O(h)$.

5. PlayCard Algorithm

- **Time Complexity:**
  - Removing a card from the player's hand: $O(1)$.
  - Updating the current trick: $O(1)$.
  - Updating game state: $O(1)$.
  - Calling the complete trick if it is the last card in current trick: O(P)
  - **Total Complexity:** $O(P)$.
- **Space Complexity:**
  - Storage for game state updates: $O(1)$.
  - Should also include the complete trick complexity: $O(1)$
  - **Total Complexity:** $O(1)$.

6. CompleteTrick Algorithm

- **Time Complexity:**
  - Determining trick winner by iterating through p players: $O(p)$.
  - Updating scores based on penalty cards: $O(p)$.
  - Updating trick history and resetting the game state: $O(1)$.
  - **Total Complexity:** $O(p)$.
- **Space Complexity:**
  - Storage for trick history: $O(1)$.
  - **Total Complexity:** $O(1)$.

7. PlayRound Algorithm

- **Time Complexity:**
  - Playing multiple tricks in a round, calls `GreedyPlay` and `PlayCard` for each trick.
  - Each round consists of t tricks.
  - Each trick involves `GreedyPlay` ($O(h * t)$) and `PlayCard` ($O(p)$).
  - **Total Complexity:** $O(t \times h \times t \times p$
- **Space Complexity:**
  - Storage for game state updates and tracking tricks: $O(t \times p)$.
  - **Total Complexity:** $O(t \times p)$.

Final Complexity Summary

Algorithm	**Time Complexity**	**Space Complexity**
InitializeGame	$O(kn)$	$O(kn)$
GetLegalMoves	$O(h)$	$O(h)$
EvaluateCard	$O(t)$	$O(1)$
GreedyPlay	$O(h \times t)$	$O(h)$
PlayCard	$O(h + p)$	$O(1)$
CompleteTrick	$O(p)$	$O(1)$
PlayRounds	$O(t \times p \times h \times t)$	$O(t \times p)$