Leveraging Generative AI for the Diamonds Game (by Anusha Tomar, WE, Cohort 5)

1 Introduction

In this report, I explore the captivating challenge of the 'Diamonds' game, a three-player scenario. The game was introduced during a bootcamp conducted by our program dean, Asokan Pichai. This document chronicles my endeavor to harness the capabilities of Generative AI (Gen AI) in developing winning strategies, integrating it as a third player, and establishing a testing framework for the generated code.

2 Game Rules

The game of Diamonds follows a set of guidelines: Each participant receives a suit of cards, excluding the diamond suit. The diamond cards are shuffled and auctioned individually. Players bid by placing one of their own cards face down. The highest bid secures the auctioned diamond card, with points awarded based on the card's rank. If multiple players tie for the highest bid, the diamond card's points are divided equally. The player accumulating the most points at the end emerges victorious.

3 Teaching Gen AI

I engaged with the Gen AI system, Gemini, and proposed playing a game based on the provided rules. Gemini sought clarification on the number of players, and I confirmed three players. Gemini then suggested a viable approach to simulate the game, including card dealing, anonymous bidding, and bid resolution mechanisms.

4 Strategy Development

When I prompted Gemini to formulate strategies for the game, it demonstrated a remarkable understanding. Gemini proposed various approaches, such as balancing point preservation and diamond acquisition, analyzing opponents'

bidding patterns, employing incremental bidding, making strategic sacrifices, adapting to situations, and considering the overall point spread.

5 Code Implementation

I attempted to translate these strategies into code by prompting Gemini to outline an approach for writing code to act as the third player. Gemini provided a well-structured strategy, explaining different components and strategies to be included. I saved this response and prompted Gemini with parts of it to obtain code for each component and strategy.

6 Code Listing

```
from collections import Counter

class Hand:
    """
    Represents a player's hand in the diamond bidding game.
    """
    def __init__(self, suit):
        """
        Initializes the hand with a specific suit using a loop.
        """
        self.suit = suit
        self.cards = Counter()
        rank = 2
        while rank <= 14:
            self.cards[str(rank)] = 1  # Add each card value (2-A)
            rank += 1
        self.revealed_diamonds = []  # Track revealed diamond values
        self.opponent_history = {}  # Optional: Track opponent bids

    def remove_card(self, card_value):
        """
        Removes a card from the hand if present.
        """
        if str(card_value) in self.cards:
            self.cards[str(card_value)] -= 1
            if self.cards[str(card_value)] == 0:
                del self.cards[str(card_value)]  # Remove card if count reaches

    def get_remaining_cards(self):
        """
        Returns a list of remaining card values in the hand.
```

```
        """
        return list(self.cards.keys())

    def has_card(self, card_value):
        """
        Checks if the player has a specific card value in their hand.
        """
        return str(card_value) in self.cards and self.cards[str(card_value)] > 0

    def update_revealed_diamond(self, diamond_value):
        """
        Updates the list of revealed diamond values.
        """
        self.revealed_diamonds.append(diamond_value)

    def update_opponent_bid(self, opponent_name, diamond_value, bid_value):
        """
        Updates the opponent's bidding history (optional).
        """
        if opponent_name not in self.opponent_history:
            self.opponent_history[opponent_name] = []
        self.opponent_history[opponent_name].append((diamond_value, bid_value))

    def place_bid(self, diamond_value):
        """
        Places a bid for a diamond based on a bidding strategy.
        """
        # Minimum bid based on diamond value
        min_bid = diamond_value // 2

        # Prioritize high cards in the beginning (adjustable threshold)
        if len(self.get_remaining_cards()) > 5 and diamond_value < 8:
            return min_bid

        # Increase bid slightly for higher value diamonds
        if diamond_value >= 10:
            min_bid += 1

        # Adjust based on remaining high cards (optional)
        high_cards_remaining = sum(self.cards[card_value] for card_value in
                                   ['J', 'Q', 'K', 'A'])
        if high_cards_remaining > 2 and diamond_value >= 8:
            min_bid += 1
        # Be more willing to bid for high value diamonds if high cards remain

        # (Optional) Adjust based on opponent history (basic example)
```

```
        if self.opponent_history:
            # Consider recent opponent bids for similar diamond values
            # (This is a simplified approach, more complex analysis is possible)
            for opponent, bids in self.opponent_history.items():
                recent_bids = bids[-2:]  # Consider the last 2 bids
                if any(value >= diamond_value // 2 for value, _ in recent_bids):
                    min_bid += 1
                # Increase bid if opponent recently bid high for similar values

        return min_bid

# Example usage
hand = Hand("Spades")
print(hand.get_remaining_cards())
# Output: ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']

hand.remove_card(7)
print(hand.get_remaining_cards())
# Output: ['2', '3', '4', '5', '6', '8', '9', '10', 'J', 'Q', 'K', 'A']

print(hand.has_card(7))  # Output: False

# Example usage
hand = Hand("Spades")
hand.update_revealed_diamond(8)
hand.update_revealed_diamond(10)
print(hand.revealed_diamonds)  # Output: [8, 10]

# Example usage
hand = Hand("Spades")
hand.update_revealed_diamond(8)
print(hand.place_bid(5))  # Output: 2 (Minimum bid)
print(hand.place_bid(10))  # Output: 6 (Increased for higher value)

# Example usage
hand = Hand("Spades")
hand.update_revealed_diamond(8)
print(hand.place_bid(5))  # Output: 2 (Minimum bid)
print(hand.place_bid(10))  # Output: 6 (Increased for higher value)
print(hand.place_bid(8))  # Output: 4 (Prioritize high cards if many remain)
hand.remove_card('K')
print(hand.place_bid(8))
# Output: 5
# (More willing to bid for high value diamonds after losing a high card)

# Example usage
```

```
hand = Hand("Spades")
hand.update_revealed_diamond(8)
print(hand.place_bid(5))  # Output: 2 (Minimum bid)

# Simulate opponent bids (assuming opponent names are known)
hand.update_opponent_bid("Player1", 8, 4)
hand.update_opponent_bid("Player2", 5, 1)
print(hand.place_bid(8))  # Output: 4 (Prioritize high cards if many remain)
```

7 Analysis and Conclusion

As highlighted in a previous discussion, explaining a card game is considered one of the most challenging technical tasks. I express my appreciation for Gemini's impressive response. However, I note that certain portions of the generated code required modifications. Overall, the experience was deemed successful, showcasing the capabilities of Gen AI in tackling intricate problems.