Developing Strategies for the Bidding Card Game 'Diamonds' with GenAI.

Ritika Thakur

1 Introduction

This report explores GenAI's ability to understand a completely new game of cards, 'Diamond Bids', think of strategies to win the said game and provide a python code simulating a game play.

2 Problem Statement

The original prompt given to Gemini was:
"This game can be played between two players and three players. In a typical three player game diamond is the trump suit and is set aside. Each player is given one of the remaining families of cards....i.e. heart, spades and clubs. now out of the diamonds suit one card is taken and everyone sees what card it is. This is the prize card. Everyone bids one card from their deck in a closed bid i.e. no one else knows what you are bidding. After the bids are revealed the prize card is distributed among players with the highest bids (i.e. if the diamond card was a 5 and two players had the highest bids among the three then both of them get 2.5 points). In case of two players the rules don't change but we simply remove the spades deck while playing. You can assume spades, decks and clubs are equivalent. At the end the player with the largest points wins."

3 Teaching GenAI the Game

3.1 Observations

- Gemini was interested in learning the game and gave questions for the user to answer in order to properly explain the game, especially when told the game was different from the ones it already knew.

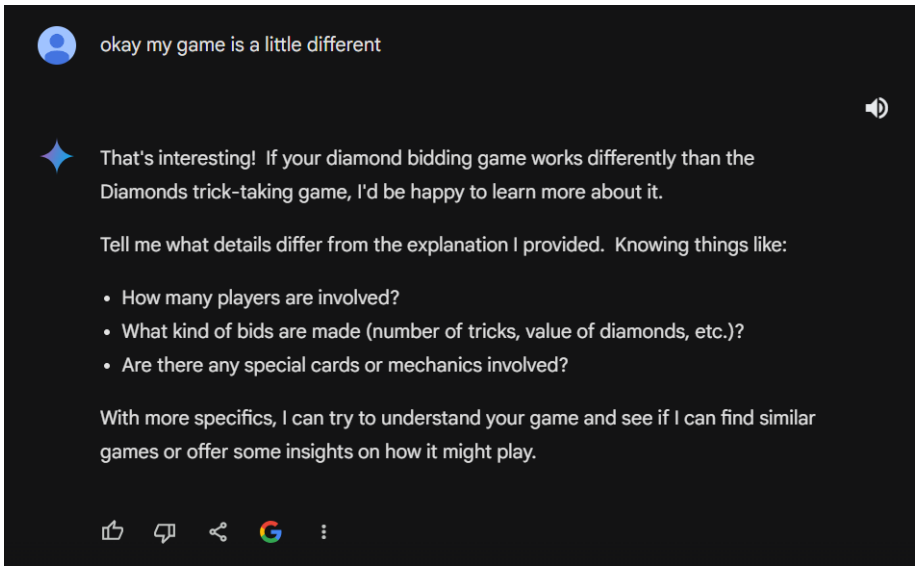

Figure 1: Prompt 1

- Gemini was able to understand almost the entirety of the game, except a point or two, which, when re-explained, it was able to catch on to.

3.2 Final Understanding of Game

Players: 2 or 3
Cards: Standard deck of 52 cards, with some modifications:

- Diamonds: Used one at a time as the prize card for each round.
- Suits: Each player receives one of the remaining three suits (hearts, clubs, spades) depending on the number of players.
  - For 3 players: all three suits are used.
  - For 2 players: spades are removed from play (essentially treating hearts and clubs as equivalent).

Gameplay: Number of Rounds: The game is played over as many rounds as there are diamond cards in the deck (13).
Rounds: In each round:

1. One diamond card is revealed as the prize card for that round.
2. Players bid one card from their assigned suit in a closed bid.
3. The player(s) with the highest bid win the prize card's face value in points (e.g., a 7 diamond wins 7 points).
4. In case of a tie for the highest bid:
   - For 3 players: Tied players split the prize card value.
   - For 2 players: Tied players also split the prize card value.

Scoring: Players accumulate points throughout the game by winning prize cards. The player with the most points at the end of the game (after all diamonds have been played) wins.
Point System:
The point value of a prize card is directly tied to its face value (2 for a 2, 10 for a 10, etc.) with face cards (J, Q, K) having values of 11, 12, and 13 respectively, and the Ace being worth 14 points.
Similarities to Other Games:
Similar to auction games and trick-taking games, players use bidding to win something valuable (prize card points).
Possible Variations:

- Award a bonus for the player who wins the most diamonds throughout the game.
- Introduce a limited number of card swaps between players during the game.

4 Iterating upon strategy

4.1 Observations

- Gemini was able to come up with relevant strategies for the game and was able to remember crucial points about the game which were important while giving these strategies.
- Gemini also tried giving specific strategies depending on number of players involved in the game.
- An interesting point is that it somehow related our game to a game of 'Poker' while strategizing.
- When prompted with some strategies of my own, Gemini analysed each of them and did not necessarily speak well for them. It had its reasonable reservations for some of them which was refreshing to see.
  It also gave some additional strategies which were a combination of the two.

4.2 Initial Strategies

Know your deck: Be familiar with the point distribution of your assigned suit (hearts, clubs, spades). This helps you understand the potential value of your cards relative to the prize card.
Consider the average: Since the point value directly reflects the card value, aim to bid slightly above the average value of your deck (around 7-8 for a 3-player game, slightly higher for a 2-player game).
Bidding history: Pay attention to past rounds and how opponents have been bidding. This can help you predict their tendencies and adjust your bids accordingly.
Bluffing: In some situations, a well-timed high bid with a lower value card can scare off opponents, especially if the prize card is high value. However, use this sparingly to avoid losing points unnecessarily.

4.2.1 Strategies for 3-Player Game

Bidding for high-value diamonds: For high-value prize cards (10, J, Q, K, A), be more aggressive with your bids, especially if you hold high cards in your suit.
Bidding wars: If you're locked in a bidding war with another player, consider if the potential points gained outweigh the risk of losing points on your bid. Sometimes folding and saving your high card for a later round can be a better strategy.
Forming temporary alliances: In a 3-player game, you could potentially form an unspoken alliance with another player by strategically bidding lower on certain rounds to weaken a particularly dominant player. However, be cautious as this can backfire.

4.2.2 Strategies for 2-Player Game (with Spades removed)

Bidding becomes more crucial: Since ties result in split points, precise bidding is even more important. Aim to bid just enough to win without overspending points on your bid card.
Analyze opponent's play: Since there's only one opponent, closely observe their bidding patterns and adjust your bids accordingly. If they consistently overbid, capitalize by bidding slightly lower for high-value diamonds.

4.3 Assessing User's Strategies and More

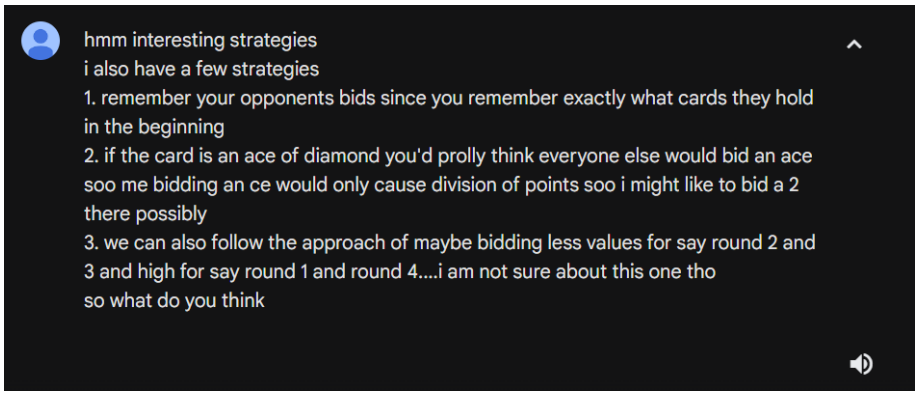

Figure 2: User's suggestion for strategies

Your strategies are interesting and can definitely be effective in the Diamond Bidding Game! Here's a breakdown of your thoughts and some additional insights:

- **Remembering Opponent Bids:**

  This is a very strong strategy, especially in a 3-player game. By remembering past bids, you can gain valuable information about your opponents' remaining cards. For example, if a player consistently bids high for high-value diamonds, you can deduce they likely don't have many high cards left in their suit.

- **Bidding Low for High-Value Aces:**

This is a clever strategy for Aces! Since Aces are the highest value card, bidding an Ace might lead to a tie. Bidding a lower card, especially a 2 (the lowest value) can be a good way to secure some points even if you don't win the full value.

- **Bidding Patterns (Low-High-Low-High):**

  This strategy can be effective in some situations, but it requires careful consideration. Bidding low early on can be risky if your opponents are also bidding low, as you might miss out on valuable early points. Similarly, bidding high later can be risky if you've depleted your high cards earlier.

Here's a more nuanced approach to consider:

- **Early Rounds (1-3):**

  You can bid slightly lower than average in these rounds, especially if the prize card value is low. This allows you to conserve your high cards and potentially bait opponents into overbidding.

- **Middle Rounds (4-7):**

  This is where you can strategically use your high cards, especially if you have a good understanding of your opponents' remaining cards based on past bids.

- **Late Rounds (8-13):**

  As the game progresses, high-value diamonds become more frequent. Be prepared to use your remaining high cards to compete for these valuable points.

Combining Strategies:
The best approach is to combine these strategies based on the situation. Here are some additional tips:

- Observe your opponents' reactions: Pay attention to how your opponents react to your bids. Are they surprised? Do they adjust their bidding accordingly?
- Adapt your strategy: Don't be afraid to adapt your strategy as the game progresses. If your initial approach isn't working, try something different based on the information you've gathered.
- Balance risk and reward: Every bid involves some risk (losing points on your bid card) and potential reward (winning the prize card). Aim to find a balance that maximizes your points while minimizing risk.

By combining your observations, these strategies, and a bit of luck, you can become a formidable Diamond Bidding Game player!

5 Pseudocode

Gemini surprisingly gave a very apt pseudocode for simulating the game play at it's first try.
An interesting thing to note is that it did a decent job at functional decomposition but did not go for OOPS, something that would have been very efficient for this kind of problem.

Algorithm 1 Diamond Bidding Game Pseudocode

```
Define number of players (2 or 3)
num_players ← INPUT("Enter number of players (2 or 3): ")
Define suits based on player count
suits ← ["hearts", "clubs", "spades"]                                   ▷ For 3 players
if num_players == 2 then
    suits.remove("spades")                                              ▷ For 2 players, remove spades
end if
procedure SHUFFLE_AND_DEAL(deck)                                        ▷ Function to shuffle and deal cards
end procedure
procedure GET_HIGHEST_BID(bids)                                         ▷ Function to get highest bid
end procedure
procedure CALCULATE_POINTS(bid_card, prize_card) ▷ Function to calculate points based on bid and prize card
end procedure
Main game loop
deck ← CREATE_DECK(())                                                  ▷ Function to create a standard deck (52 cards)
SHUFFLE_AND_DEAL(deck)
for round_num in range(1, 14) do                                        ▷ Iterate for 13 rounds (number of diamond cards)
    Reveal prize card (diamond from the deck)
    prize_card ← deck.pop()                                             ▷ Simulates removing a card from the deck
    Players make closed bids (one card from their suit)
    bids ← []
    for player in range(num_players) do
        bid ← ...                ▷ Replace with logic to get player's bid (considering hidden information)
        bids.append(bid)
    end for
    Find highest bid(s) and winner(s)
    highest_bid ← GET_HIGHEST_BID(bids)
    winners ← []                                                        ▷ List to store player indexes who won
    for player in range(num_players) do
        if bids[player] == highest_bid then
            winners.append(player)
        end if
    end for
    Calculate and award points based on winner(s)
    for winner in winners do
        points_awarded ← CALCULATE_POINTS(bids[winner], prize_card)
                                                                        ▷ Update player score based on points awarded
    end for
end for
Print final scores and determine winner
```

6 Final code

6.1 Observations

- Gemini forgot/misunderstood important aspects of the game while writing the code which it was very clear about until now. One such common mistake was regarding shuffling and dealing of cards.
- It also had a hard time with scoring system as well as overlooking certain syntax issues.
- Trying to get an interactive code was the hardest, as there were overlooked syntax errors, Gemini was unable to take into account user's previous inputs correctly while forming a strategy for AI.
- When given it's own code to Gemini, it was able to recognize several limitations and suggest improvements instead of simply praising and explaining it.
- Re-prompting it's explanation of the game along with strategies significantly improved the code accuracy, other than minor conceptual misunderstandings regarding the game.
- After a few hours of testing and fixing the code we had to switch to GitHub co-pilot.
  GitHub co-pilot was extremely helpful in fixing bugs in the code provided by Gemini. However, when given the fresh prompt of the game it performed worse than Gemini at understanding it.

6.2 The code we finally achieved

The code below applies all strategies in theory. However, the AI always ends up bidding its lowest available card, a bug which Gemini, GitHub co-pilot or ChatGPT couldn't solve.
In my experience, this code will not get better if we were to rely on GenAI.

```
# -*- coding: utf-8 -*-
"""Diamond_bids.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1l_1xlyZFxyD9-sZACcEnYddV7sS0vED2

# Code 1
"""

import random

def bid_diamonds(hand, remaining_diamonds, opponent_seen_cards, num_players,
    player_previous_bids):
  """
  This function implements a bidding strategy for the Diamonds game that
      considers:

  - Opponent's potential cards based on seen bids (opponent_seen_cards).
  - Player's hand strength (number of high cards).
  - Round number (early, middle, late).
  - Number of players (2 or 3).

  Args:
    hand: A list of card values in the player's hand.
    remaining_diamonds: A list of card values of the remaining diamonds.
    opponent_seen_cards: A list of card values seen from opponent's bids.
    num_players: Number of players in the game (2 or 3).
    player_previous_bids: A list of the player's previous bids.
```

```
29
30      Returns:
31          A list of card values representing the player's bids for each diamond.
32      """
33
34      bids = []
35      total_rounds = len(remaining_diamonds)
36
37      for i, diamond in enumerate(remaining_diamonds):
38          current_round = i + 1
39          potential_highest_opponent_card = max(opponent_seen_cards, default=0)
40
41          if current_round <= int(total_rounds / 3):
42              if diamond <= 7:
43                  bid = min((card for card in hand if card >
                        potential_highest_opponent_card), default=0)
44              else:
45                  bid = choose_bid(hand, potential_highest_opponent_card, diamond,
                        player_previous_bids, aggressive=False)
46          elif current_round <= int(2 * total_rounds / 3):
47              bid = choose_bid(hand, potential_highest_opponent_card, diamond,
                    player_previous_bids, aggressive=False)
48          else:
49              bid = choose_bid(hand, potential_highest_opponent_card, diamond,
                    player_previous_bids, aggressive=True, prioritize_high=True)
50
51          bids.append(bid)
52          hand.remove(bid)  # remove the bid card from AI's hand
53
54      return bids
55
56  def choose_bid(hand, potential_highest_opponent_card, diamond_value,
        player_previous_bids, aggressive=False, prioritize_high=False):
57          # Consider bidding low for Aces even if aggressive
58          if 14 in hand and not prioritize_high:
59              non_ace_cards = [card for card in hand if card != 14]
60              if non_ace_cards:  # Check if there are any non-Ace cards
61                  return min(non_ace_cards)
62              else:
63                  return 14  # If all cards are Aces, return 14
64
65          # Prioritize high cards in late rounds if available
66          if prioritize_high and any(card > 10 for card in hand):
67              return max(card for card in hand if card > 10)
68
69          # Choose a card higher than opponent's potential highest, considering
                hand strength
70          higher_cards = [card for card in hand if card >
                potential_highest_opponent_card]
71          if higher_cards:
72              if aggressive or diamond_value > 10:  # Bid aggressively for high-
                    value diamonds
73                  return max(higher_cards)
74              else:
75                  # Sometimes choose a lower card for strategy
```

```
            return random.choice(higher_cards)
    else:
        # If no high card or not aggressive, consider bluffing or low bid
        if diamond_value < 7:  # If the diamond value is low, bid a low card
            return min(hand)
        else:  # If the diamond value is high, bid a high card
            if player_previous_bids and max(player_previous_bids) > max(hand)
                :
                return max(hand)
            else:
                return random.choice([card for card in hand if card > max(
                    player_previous_bids, default=0)])

def play_diamonds_game(num_players=2):
  """
  Simulates a Diamonds game between the player and the AI.
  """

  suits = ["hearts", "clubs", "spades"]  # Define suits (adjust for 2 players
      )
  num_players = 3  # Change to 2 for a 2-player game

  # Separate suits (without shuffling player cards)
  hearts = [rank for rank in range(2, 15)]
  clubs = [rank for rank in range(2, 15)]
  spades = [rank for rank in range(2, 15)]

  # Deal cards based on players (no shuffling player cards)
  player_cards = []
  if num_players == 3:
    player_cards = [hearts, clubs, spades]
  elif num_players == 2:
    player_cards = [hearts, clubs]
  else:
    print("Invalid number of players (must be 2 or 3)!")
    exit()

  # Create and shuffle diamonds deck
  diamonds = [rank for rank in range(2, 15)]
  random.shuffle(diamonds)

  # Separate remaining cards (all cards except diamonds)
  remaining_cards = player_cards.copy()  # Combine player cards
  player_hand = remaining_cards.pop(0)  # Player gets first suit

  # Distribute remaining cards to opponent(s)
  opponent_hand = []
  for _ in range(num_players - 1):
    opponent_hand.extend(remaining_cards.pop(0))

  opponent_seen_cards = []  # Track opponent's seen cards

  # Player score and AI score initialization
  player_score = 0
  opponent_score = 0
```

```
    ai_bids = []  # Track AI's bids
    player_previous_bids = []

    # Game loop for each diamond round
    for diamond in diamonds:
        print(f"\nRound: {diamonds.index(diamond) + 1} (Diamond: {diamond})")

        # Player bid
        player_bid = int(input("Your bid (enter card value 2-14): "))
        player_hand.remove(player_bid)  # Remove bid card from player's hand
        player_previous_bids.append(player_bid)
        # AI bid (using bid_diamonds function)
        while True:
            ai_bid = choose_bid([card for card in opponent_hand if card not in
                ai_bids], max([player_bid] + opponent_seen_cards), diamond,
                player_previous_bids, aggressive=True)
            if ai_bid not in ai_bids:
                break
        ai_bids.append(ai_bid)
        opponent_hand.remove(ai_bid)
        opponent_seen_cards.append(ai_bid)
        print(f"AI bid: {ai_bid}")

        # Determine winner and award points
        if player_bid > ai_bid:
            winner = "Player"
            player_score += diamond
        elif player_bid < ai_bid:
            winner = "AI"
            opponent_score += diamond
        else:
            winner = "Tie"
            player_score += diamond // 2
            opponent_score += diamond // 2

        print(f"{winner} wins the round! (Scores: Player: {player_score}, AI: {
            opponent_score})")

    # Print final scores and winner
    print(f"\nFinal Scores: Player: {player_score}, AI: {opponent_score}")
    if player_score > opponent_score:
        print("You win!")
    elif player_score < opponent_score:
        print("AI wins!")
    else:
        print("It's a tie!")

# Play the game (change num_players to 2 for a 2-player game)
play_diamonds_game()

"""# Code 2"""

import random

# Function to calculate the point value of a card
```

```
def get_card_value(card):
    if card in ['J', 'Q', 'K']:
        return 11
    elif card == 'A':
        return 14
    else:
        return int(card)

# Function to simulate a round of bidding
def play_round(user_card, ai_card):
    if get_card_value(user_card) != get_card_value(ai_card):
        if get_card_value(user_card) > get_card_value(ai_card):
            user_points = get_card_value(user_card)
            ai_points = 0
        else:
            user_points = 0
            ai_points = get_card_value(ai_card)
    else:
        user_points = get_card_value(user_card) // 2
        ai_points = get_card_value(ai_card) // 2

    return user_points, ai_points

# Function to simulate the game
def play_game():
    suits = ['hearts', 'clubs', 'spades']
    diamonds = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K',
        'A']
    random.shuffle(diamonds)
    ai_cards = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K',
        'A']
    total_points_user = 0
    total_points_ai = 0
    user_suit = random.choice(suits)
    ai_suit = random.choice([suit for suit in suits if suit != user_suit])

    for diamond in diamonds:
        prize_card = diamond

        print(f"Prize card: {prize_card} of diamonds")
        print(f"Your suit: {user_suit}")
        print(f"AI suit: {ai_suit}")

        user_card = input("Enter your card: ")
        ai_card = random.choice(ai_cards)
        ai_cards.remove(ai_card)

        user_points, ai_points = play_round(user_card, ai_card)
        total_points_user += user_points
        total_points_ai += ai_points

        print(f"You won {user_points} points")
        print(f"AI won {ai_points} points")
        print("")

```

```
    print("Game over!")
    print(f"Your total points: {total_points_user}")
    print(f"AI's total points: {total_points_ai}")

    if total_points_user > total_points_ai:
        print("You win!")
    elif total_points_user < total_points_ai:
        print("AI wins!")
    else:
        print("It's a tie!")

# Start the game
play_game()
```

Listing 1: Python code for the Diamond Bidding Game

7 Analysis and Conclusion

In this report, we explored the capabilities of GenAI in understanding and strategizing for the card game 'Diamond Bids'. Through a series of interactions, we taught GenAI the rules of the game and discussed various strategies to improve its gameplay.
Our interactions with GenAI revealed several strengths and weaknesses:

7.1 Strengths

- **Understanding Complexity:** GenAI demonstrated the ability to understand complex game rules and adapt its strategies accordingly very quickly.
- **Strategic Insights:** GenAI was able to provide valuable strategic insights, including appropriate bidding patterns, bluffing techniques, and adaptive gameplay strategies.
- **Interactive Learning:** Through iterative discussions and prompts, GenAI learned from user feedback and improved its understanding of the game mechanics and strategies.

7.2 Weaknesses

- **Implementation Accuracy:** While GenAI could generate pseudocode and Python code for the game, the implementation often contained syntax errors, conceptual misunderstandings, and logic flaws.
- **Limited Problem-Solving:** GenAI struggled with certain aspects of the game, such as interactive gameplay with user inputs, scoring calculations, and handling edge cases.
- **Dependency on Human Intervention:** Despite its learning capabilities, GenAI relied heavily on human intervention to correct errors, provide guidance, and improve its performance.

7.3 Conclusion

I would like to point out that Gemini did an admirable job when it came to explaining to the user, understanding what the user had to say, as well as testing the code.
Overall, our experiment with GenAI in developing strategies for the 'Diamond Bids' game was insightful but highlighted the limitations of current AI technologies in complex problem-solving domains. While GenAI showed promise in understanding game mechanics and generating strategic insights, its implementation capabilities fell short of practical usability.
Moving forward, advancements in AI technologies, particularly in natural language understanding, reasoning, and code generation, may enhance the capabilities of AI systems like GenAI in tackling complex tasks. However, for now, human intervention and collaboration remain essential for bridging the gap between AI's potential and real-world applications.

This project clearly demonstrated the importance of learning prompt engineering as well importance of humans in order to fully utilise GenAI in problem solving. It is definitely a harder task when you have to teach it before you can ask questions than when it already knows what you need it to know to answer your questions.

8 References

- Google Docs link for the chat here
- Google Colab link for the final codes here