Modeling UNO Gameplay Using Graph Theory to Analyze Optimal Strategies

Muhammad Rizain Firdaus - 13523164
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13523164@std.stei.itb.ac.id, icon.firdaus@gmail.com

***Abstract*— This paper presents a graph theory-based approach to analyze and optimize strategies in the card game UNO. We model the gameplay using weighted graphs and implement both Dijkstra's and Minimax algorithms for strategic decision-making. Our implementation demonstrates that Dijkstra's algorithm excels at optimizing immediate moves, while Minimax provides robust defensive strategies in multi-player scenarios. The results show that graph-based modeling effectively identifies optimal card sequences and strategic decisions, contributing to the broader understanding of applying graph theory in game strategy optimization.**

***Keywords*—graph theory, UNO card game, Dijkstra's algorithm, Minimax algorithm, game strategy optimization.**

I. Introduction

Graph theory, a branch of discrete mathematics, is widely used to model relationships and optimize decision-making processes in complex systems. It provides a structured approach to understanding interactions within networks, making it a powerful tool for solving problems in various fields such as computer science, logistics, and social networks. The application of graph theory to games, particularly those involving strategic decision-making, has gained significant attention due to its ability to uncover optimal strategies and predict outcomes.

UNO, a popular card game with simple yet dynamic rules, presents a unique opportunity for applying graph theory in a recreational setting. In UNO, each card is characterized by its color, number, and specific actions (e.g., skip, reverse, draw two), and players are required to match cards based on color or number to make moves. This leads to a system where the relationships between cards and players can be effectively modeled using graph theory.

Fig 1. Uno Games

Source: Playstation Store App

In this paper, we model the game of UNO using graphs, where each card is represented as a vertex and edges connect cards that can be played consecutively according to the game's rules. The graph will help identify optimal strategies by analyzing the connections between cards and their effects on gameplay. Specifically, we explore how the structure of the graph—such as the adjacency of playable cards, the potential sequences of moves, and the strategic positioning of action cards—can inform decision-making.

Through this approach, we aim to provide a deeper understanding of the strategic elements in UNO, answering questions such as:

- What is the most efficient sequence of moves to reduce hand size quickly?
- How can players identify which cards should be prioritized to maintain a playable hand?
- How does the presence of action cards (e.g., skip, reverse) influence the optimal strategy?

By modeling the gameplay through a graph, we not only highlight the potential of graph theory in game theory but also demonstrate how it can be applied to enhance decision-making in simple rule-based environments. The results of this study offer valuable insights into strategy optimization, with broader implications for games involving similar structured interactions.

II. Theoretical Basis

1. Graph Theory

Graph theory, a pivotal area of discrete mathematics, examines the relationships between pairs of objects through graphs. A graph comprises **vertices** (nodes) and **edges** (connections between nodes). In this study, each **vertex** represents a card in the UNO game, and each **edge** signifies a playable connection between two cards, adhering to the game's rules [1].

Graphs can be categorized into various types, including:

- **Bipartite Graphs**: Graphs where vertices can be divided into two disjoint subsets, with edges only connecting vertices from different subsets. In UNO, this could represent cards classified by color or number [1].

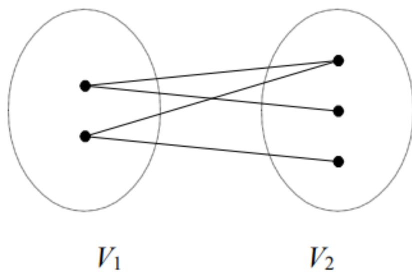


Fig 2.1. Bipartite Graph
Source: Graph-Part-1 informatika.stei.itb.ac.id

- **Weighted Graphs**: Graphs where edges have weights assigned, representing some form of cost or distance. In UNO, weights could denote the strategic value of a card in terms of its impact on the game's progress [1].

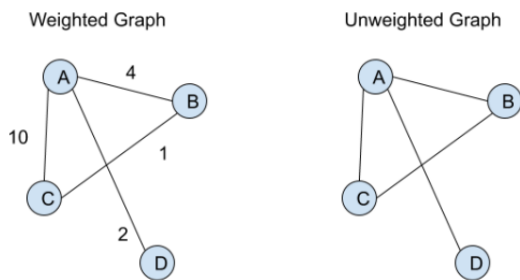


Fig 2.2. Weighted Graph and Unweighted Graph
Source: Graph-Part 1 informatika.stei.itb.ac.id

- **Trees and Cycles**: A tree is a connected graph with no cycles, while a cycle is a path where the starting and ending vertices are the same. Cycles may appear in the sequence of moves during the game, where cards loop back to earlier positions [1].

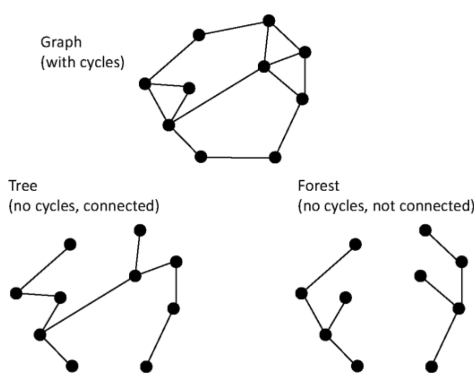


Fig 2.3 Tree, Cycles, and Forest
Source: Graph-Part 1 informatika.stei.itb.ac.id

2. Graph Representation of UNO

In this study, we model the UNO game using a graph where each card is represented by a vertex, and edges connect cards that are playable based on the game's rules. The game follows a set of simple yet dynamic rules:

- Players can play a card if it matches the color or number of the card in the centre [3].
- Special action cards (such as **skip**, **reverse**, **draw two**, **wild**) have additional effects that modify the sequence of moves or the flow of the game [3].

Thus, in our graph representation:

- **Vertices** represent the cards in a player's hand or the deck.
- **Edges** represent possible moves between cards that satisfy the game's playability rules (matching colors or numbers).
- Action cards introduce additional edges that affect the state of the game (such as skipping the next player or reversing the direction of play).

3. Graph Algorithms for Strategy Analysis

Several key graph algorithms can be applied to analyze optimal strategies in UNO:

3.1. **Depth-First Search (DFS) and Breadth-First Search (BFS)**: DFS explores the graph by diving deep into one possible sequence of moves before backtracking. This can be used to simulate a sequence of card plays to find the longest playable sequence or to identify dead ends in gameplay. By exploring all possible paths, DFS can help analyze which cards should be played first to maximize future opportunities [4].): BFS explores the graph level by level, ensuring that all reachable cards are explored in a systematic order. In the context of UNO, BFS could help identify the shortest path (i.e., the fewest number of moves) to reach a winning state by playing all cards [4].

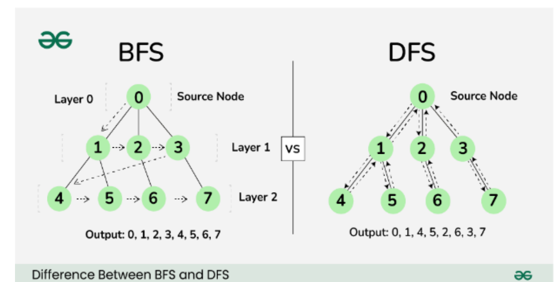


Fig 2.4. DFS and BFS Representation
Source: GeeksforGeeks

3.2. **Maximum Matching**: This algorithm helps identify pairs of cards that can be played consecutively based on the game's rules. By maximizing the number of matching cards in the player's hand, we can strategize the optimal sequence for card play [4].

3.3. **Graph Coloring**: Graph coloring can be used to represent situations where cards of a certain color must be prioritized or separated. The coloring algorithm can help in situations where multiple color cards exist, and finding an optimal color grouping for a player's hand can be beneficial [4].

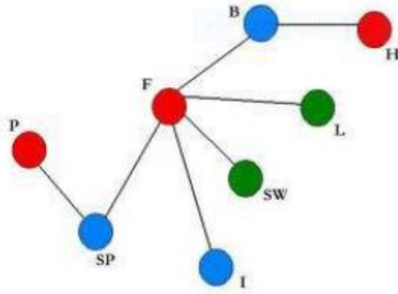


Fig 2.5. Graph Coloring
Source: Graph-Part 3-informatika.stei.itb.ac.id

3.4. **Shortest Path Algorithms (Dijkstra's or Bellman-Ford)**: These algorithms can be used to calculate the quickest path to reduce a player's hand size. In the context of UNO, the goal is to minimize the number of turns or moves required to empty the hand, taking into account action cards that could alter the course of play [4].

3.5. **Minimax Algorithm**: The Minimax Algorithm is a decision-making strategy typically used in two-player games where one player's gain is another player's loss. In the context of UNO, the algorithm can be adapted to model strategic decision-making in dynamic, multi-player scenarios by treating the game's state as a tree structure, where each node represents a possible game state, and edges represent potential moves.

4. Strategy Optimization in Game Theory

In game theory, the concept of **optimal strategies** is crucial. An optimal strategy is one that maximizes a player's chances of winning, taking into account all possible moves and counter-moves. The optimal strategy in UNO is influenced by both the state of the game (i.e., the cards in the hand and on the table) and the rules governing card play [5].

In the context of UNO, graph theory helps optimize strategy by:

- **Modeling the possible sequences of plays** to minimize the number of moves required to win [5].
- **Analyzing the effects of action cards** (such as Wild and Skip) to determine the best time to use them for disrupting the opponent's progress [5].
- **Identifying the best starting card** by analyzing the graph to determine which card is most likely to open up the greatest number of future plays [5].

By combining these graph-based algorithms with game-theory concepts, it is possible to develop a comprehensive strategy that can be applied in real-world UNO gameplay [5].

III. Analyzing UNO Strategy Using Graph-Based Approach

Based on the theorical basis that had been discussed, as the topic of this paper is bound to a strategic games. The graph is used to represents not only the standard UNO card that used in the game, but also the game-flow is represented by using graph. The graph that is being used is basically to generate the best algorithm procedure to find the best move it could find. For the example in this paper is going to used the Dijkstra's Algorithm (an algorithm to find the shortest path by using weighted value by weighted graph represented).

1. Cards Value Representation Using Graph

In the context of UNO, each card possesses unique attributes such as color, number, or action type (e.g., Skip, Reverse, Wild). These attributes can be effectively mapped into a graph structure, where each card is represented as a **vertex** and their playable relationships are defined as **edges**. The edges in the graph signify the possible moves allowed based on UNO rules, such as matching by color, number, or action.

1.1. Vertex Representation

Each vertex in the graph corresponds to a specific card in the UNO deck. For instance: A red "5" card is represented as a vertex labelled **R5**. A blue "Skip" card is represented as a vertex labelled **B-Skip**.

1.2. Edges Representation

Edges between vertices define the transitions between playable cards:

- If two cards share the same color, an edge is formed. For example: **R5** ↔ **R7** (Red cards "5" and "7").
- If two cards share the same number, an edge is formed. For example: **R5** ↔ **G5** (Cards "5" in red and green colors).
- Action cards (e.g., Wild) connect to all possible playable cards. For example: **Wild** ↔ **R5**, **Wild** ↔ **B-Skip**.

1.3. Weighted Graph Representation

Edges between vertices define the transitions between playable cards:

- If two cards share the same color, an edge is formed. For example: **R5** ↔ **R7** (Red cards "5" and "7").
- If two cards share the same number, an edge is formed. For example: **R5** ↔ **G5** (Cards "5" in red and green colors).
- Action cards (e.g., Wild) connect to all possible playable cards. For example: **Wild** ↔ **R5**, **Wild** ↔ **B-Skip**.

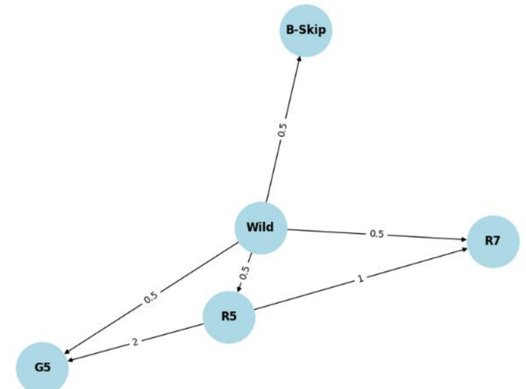


Fig 3.1. Graph for cards value representation
Source: Author's Document

2. Game Flow Representation Using Graph

The game flow in UNO can also be represented as a dynamic graph. Here, the focus is on the sequence of moves during a player's turn and how the graph evolves with each play.

2.1. Turn-Based Transitions

During each turn, the graph is updated based on the active card on the discard pile:

- The active card becomes the "source" vertex.
- All valid moves from the player's hand form the outgoing edges from the source vertex.

2.2. Example of Turn-Based Graph Evolution

Suppose the active card is **R5**, and the player's hand contains {R7, G5, Wild}:

- Graph at turn start: Source vertex: **R5**. Outgoing edges: {R5 → R7, R5 → G5, R5 → Wild}.
- After playing **R7**, the graph updates: Source vertex: **R7**. Outgoing edges: {R7 → Wild} (assuming only Wild remains playable).

2.3. Dynamic Weight Adjustment

To optimize the decision-making process, edge weights can dynamically adjust based on:

- **Remaining cards**: Assign higher weights to moves that leave the opponent with more cards.
- **Action cards**: Prioritize moves that disrupt opponents (e.g., Skip, Reverse).

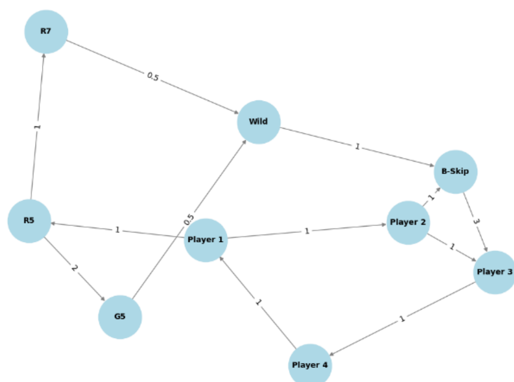


Fig 3.2. Graph for game-flow
Source: Author's document

3. Strategic Pathfinding Using Dijkstra's Algorithm

The graph structure allows for the application of Dijkstra's algorithm to find the optimal sequence of plays. The goal is to identify the shortest path (lowest weight) to achieve a winning condition, such as emptying the hand.

3.1. Algorithm Implementation

- **Input Graph**: Provide the weighted graph representing the current state of the game.
- **Start Node**: Set the active card as the starting vertex.
- **End Node**: Define the goal state, such as a vertex representing an empty hand or a specific winning move.
- **Pathfinding**: Apply Dijkstra's algorithm to calculate the shortest path and its total weight.

3.2. Example Application

- Initial state: Player's hand = {R7, G5, Wild}, Active card = R5.
- Graph weights:
  - R5 → R7: Weight 1 (matching color).
  - R5 → G5: Weight 2 (matching number).
  - R5 → Wild: Weight 0.5 (universal playability).
- Dijkstra's output:
  - Optimal path: R5 → Wild.
  - Total weight: 0.5.

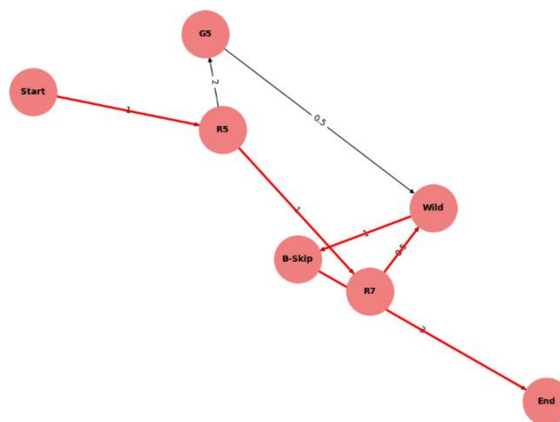


Fig 3.3. Dijkstra's algorithm pathfinding graph visualization
Source: Author's document

Table 3.1. Effectiveness percentage table

Move	Weight (w)	Effectiveness $(\frac{1}{w})$	Percentage $(\frac{E}{\Sigma E} \times 100\%)$
R7 (Same Color)	1	1	28,57%
G5 (Same Number)	2	0,5	14,29%
Wild (Universal)	0,5	2	57,14%

4. Dynamic Strategy Optimization Using Minimax Algorithm

4.1. Representing the Game Using a Game Tree

To apply the Minimax Algorithm, the game is first represented as a tree structure. Each node in the tree corresponds to a unique game state, defined by:

- The player's hand (cards available for play).

- The current top card on the discard pile.
- The cards potentially held by the opponents (estimated based on previous moves).
- The state of the draw pile.

Edges between nodes represent possible actions, such as playing a card, drawing a card, or skipping a turn. The game tree is expanded up to a specific depth, representing a sequence of moves. Terminal nodes (leaf nodes) correspond to the end of a series of moves, where the evaluation function assigns a score to the resulting game state.

4.2. Evaluation Function

The evaluation function is central to the Minimax Algorithm, as it quantifies the desirability of each game state. For UNO, the evaluation function could consider:

- **Reduction in Player's Cards**: Favor moves that reduce the number of cards in the player's hand.
- **Impact on Opponents**: Penalize moves that allow opponents to play or reduce their cards easily.
- **Special Card Effects**: Reward moves that utilize special cards (e.g., Skip, Reverse, Wild) to disrupt opponents or gain strategic advantage.
- **Game Progression**: Account for moves that bring the player closer to winning the game while limiting the opponents' ability to do so.

4.3. Applying the Minimax Algorithm

The Minimax Algorithm is applied by first constructing a game tree from the current state of the UNO game, where each branch represents a sequence of possible moves. Starting at the root node, which corresponds to the current game state, the algorithm explores all potential actions up to a specific depth in the tree. Each terminal node (leaf node) is evaluated using an evaluation function that assigns a score to the resulting game state. The algorithm then backpropagates these scores to determine the optimal path. Maximizing nodes, which represent the current player's turn, select the move that yields the highest score, aiming to improve the player's advantage. Conversely, minimizing nodes, representing the opponents' turns, choose the move with the lowest score, attempting to minimize the maximizing player's benefit. This process ensures that the selected move considers not only the player's perspective but also the opponents' potential responses. Additionally, after each turn, the game tree is dynamically recalculated, allowing the strategy to adapt to the evolving game state and maintain its effectiveness throughout the game.

4.4. Scenario Example

Consider a scenario where the player's hand includes the following cards: R5, R7, G5, B-Skip, Wild. The current top card on the pile is R5. The possible moves are:

- Play R7 (same color as the top card).
- Play G5 (same number as the top card).
- Play Wild (valid regardless of the top card).
- Play B-Skip (invalid unless a blue card or Skip is on top, so not considered).
- Draw a card from the draw pile.

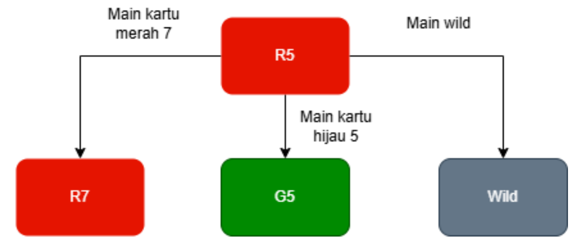


Fig 4.1. Visualization for game scenario examples
Source: Author's document

The Minimax Algorithm evaluates these moves, considering the effects on the game state and opponents. For instance, playing Wild might force the opponent to draw cards if they lack the chosen color, while playing R7 maintains the current color and avoids helping the opponent.

4.5. Algorithm Implementation

The implementation of the Minimax Algorithm in the context of UNO requires constructing a game tree, where each node represents a game state. The edges denote transitions resulting from possible moves, and terminal nodes are evaluated to determine the desirability of a game state. The algorithm backpropagates these evaluations to identify the optimal move.

```python
import math
import networkx as nx
import matplotlib.pyplot as plt

# Example cards and evaluation values
card_values = {
    'R5': 5,
    'R7': 7,
    'G5': 5,
    'Wild': 10,
    'B-Skip': 3,
    'end': 0
}

# Example game tree
game_tree = {
    'start': ['R5', 'Wild'],
    'R5': ['R7', 'G5'],
    'Wild': ['B-Skip', 'R7'],
    'R7': ['end'],
    'G5': ['end'],
    'B-Skip': ['end']
}

# Minimax Algorithm with Graph Coloring
def minimax(state, depth, maximizing_player, graph=None, parent=None):
    """
    Implements the Minimax Algorithm with graph coloring.
    """
    if graph is not None:
        # Assign node color based on player type or terminal state
        if state not in game_tree:  # Terminal state
            color = "gray"
        else:
            color = "green" if maximizing_player else "red"

        graph.add_node(state, value=card_values.get(state, -math.inf), color=color)
        if parent is not None:
            graph.add_edge(parent, state)

    # Base case: terminal state or depth limit
    if state not in game_tree or depth == 0:
        return card_values.get(state, -math.inf)

    if maximizing_player:
        max_eval = -math.inf
        for child in game_tree[state]:
            eval = minimax(child, depth - 1, False, graph, state)
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = math.inf
        for child in game_tree[state]:
            eval = minimax(child, depth - 1, True, graph, state)
            min_eval = min(min_eval, eval)
        return min_eval
```

Fig 4.2. Source code for minimax algorithm (based on the example given)
Source: Author’s Document

The **Minimax Algorithm** implemented in the code is designed to evaluate the best move in a game scenario, represented as a tree structure. The algorithm recursively explores all possible moves from a given state, alternating between a maximizing player (who seeks to maximize the score) and a minimizing player (who seeks to minimize the score).

In the provided example, the game starts with the "start" state. From here, the player can choose between playing the card **R5** (Red 5) or **Wild**. If the player chooses **R5**, the subsequent moves could be **R7** (Red 7) or **G5** (Green 5). If the player chooses **Wild**, the next moves could be **B-Skip** (Blue Skip) or **R7**. Each of these moves leads to an **end** state, which signifies the termination of the game.

The algorithm begins at the root node ("start") and evaluates each branch of the tree to a specified depth (3 in this case). For instance, if the maximizing player selects **R5**, the algorithm proceeds to evaluate the children of **R5**, which are **R7** and **G5**. The values assigned to these cards are retrieved from the **card_values** dictionary, where **R7** has a value of 7 and **G5** has a value of 5. The algorithm compares these values to determine the best move for the maximizing player.

If the minimizing player then takes a turn, they will evaluate their moves (e.g., **B-Skip** or **R7**) with the goal of minimizing the score. For example, playing **B-Skip** leads to an end state with a value of 3, while **R7** leads to an end state with a value of 7. The minimizing player will prefer the branch that leads to the lower value (3 in this case). The algorithm uses the evaluation values to backtrack through the tree, calculating the optimal path for both the maximizing and minimizing players.

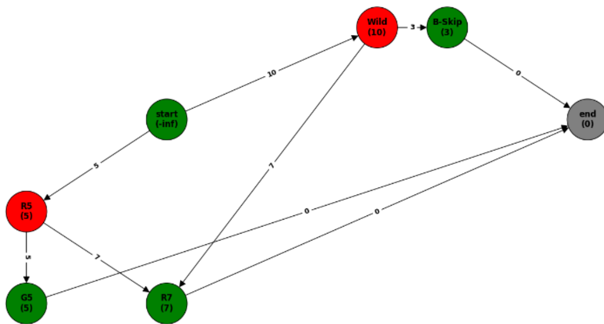


Fig 4.3. Minimax graph visualization
Source: Author’s document

5. Real Time Game Implementation

5.1. Introduction to the game

The provided code simulates a UNO card game with players using two different algorithms: Dijkstra's Algorithm for the first player and the Minimax algorithm for the remaining players. The game adheres to the classic rules of UNO, where players take turns playing cards that match either the color or the value of the top card in the discard pile. Special cards like 'Skip', 'Reverse', and 'Draw Two' introduce additional mechanics, while 'Wild' and 'Wild Draw Four' allow players to change the color.

5.2. Game scenario

In the UNO game simulation, four players are involved, with Player 1 using Dijkstra’s Algorithm and Players 2-4 utilizing the Minimax Algorithm. The game starts with each player receiving 7 cards from a shuffled deck consisting of 108 cards, which includes number cards (0-9) and special cards (Skip, Reverse, Draw Two, Wild, and Wild Draw Four). A random card is placed in the discard pile to begin the game. During each round, the current player must either play a valid card matching the color or value of the top card on the discard pile or draw a card from the deck. The algorithms choose the optimal card based on their strategies: Dijkstra’s Algorithm for Player 1 calculates the best move based on card weights, while the Minimax Algorithm simulates future moves to select the most favorable one. Special cards like Skip, Reverse, Draw Two, Wild, and Wild Draw Four add strategic layers to the game, affecting the turn order, direction of play, and forcing other players to draw cards. If a player cannot play a valid card, they draw from the deck and continue. The game ends when a player discards all their cards, or if the maximum round limit (100 rounds) is reached, with the player having the fewest remaining cards winning in the latter case.

5.3. Algorithm and source code

The **UnoGame** class is the central component of the UNO game simulation, encompassing all the game logic and mechanics. The game initializes with a shuffled deck of 108 cards, which include both number cards (0-9) and special action cards (Skip, Reverse, Draw Two, Wild, Wild Draw Four). The game begins with each player being dealt 7 cards, and a random card is placed in the discard pile. The **is_valid_move** method checks if a card played is compatible with the top card in the discard pile based on its color or value, and the **get_valid_moves** method returns a list of playable cards. The AI decision-making is handled through two distinct strategies: Dijkstra's Algorithm for Player 1 and Minimax for the other players. Dijkstra's Algorithm selects the optimal move by calculating the weight of each card, where special cards (Wild, Draw Two, etc.) are valued differently. In contrast, the Minimax Algorithm evaluates possible future states recursively using depth-first search with alpha-beta pruning to determine the best move. Special cards such as Skip, Reverse, and Draw Two have their effects on gameplay, manipulating turn order or forcing players to draw cards. The game progresses with players either playing a valid card or drawing from the deck if no valid moves are available. If a Wild card is played, the player selects a new color for the game to continue. The game ends when a player discards all their cards, or if the game exceeds 100 rounds, with the player holding the fewest cards being declared the winner in the latter case.

```python
1   import random
2   from collections import defaultdict
3   import time
4
5   class Card:
6       def __init__(self, color, value):
7           self.color = color
8           self.value = value
9
10      def __str__(self):
11          return f"{self.color} {self.value}"
12
13      def __repr__(self):
14          return self.__str__()
15
16  class UnoGame:
17      def __init__(self, num_players):
18          self.colors = ['Red', 'Blue', 'Green', 'Yellow']
19          self.values = list(range(0, 10)) + ['Skip', 'Reverse', 'Draw Two']
20          self.special_cards = ['Wild', 'Wild Draw Four']
21          self.num_players = num_players
22          self.players = [[] for _ in range(num_players)]
23          self.current_player = 0
24          self.direction = 1
25          self.deck = self.create_deck()
26          self.discard_pile = []
27          self.player_types = ['Dijkstra'] + ['Minimax'] * (num_players - 1)
28
29      def create_deck(self):
30          deck = []
31          # Regular cards
32          for color in self.colors:
33              for value in self.values:
34                  deck.append(Card(color, value))
35                  if isinstance(value, int):
36                      deck.append(Card(color, value))
37
38          # Special cards
39          for _ in range(4):
40              for special in self.special_cards:
41                  deck.append(Card('Black', special))
42
43          random.shuffle(deck)
44          return deck
45
46      def deal_cards(self):
47          for i in range(self.num_players):
48              for _ in range(7):
49                  self.players[i].append(self.deck.pop())
50
51          self.discard_pile.append(self.deck.pop())
52          while self.discard_pile[-1].color == 'Black':
53              self.deck.append(self.discard_pile.pop())
54              random.shuffle(self.deck)
55              self.discard_pile.append(self.deck.pop())
56
57      def is_valid_move(self, card):
58          top_card = self.discard_pile[-1]
59          return (card.color == 'Black' or
60                  card.color == top_card.color or
61                  (isinstance(card.value, str) and card.value == top_card.value) or
62                  (isinstance(card.value, int) and card.value == top_card.value))
63
64      def get_valid_moves(self, player_cards):
65          return [card for card in player_cards if self.is_valid_move(card)]
66
67      def calculate_move_weight(self, card, player_hand):
68          if card.color == 'Black':
69              return 0.5
70          elif not isinstance(card.value, int):
71              if card.value == 'Draw Two':
72                  return 0.6
73              elif card.value in ['Skip', 'Reverse']:
74                  return 0.7
75              return 0.8
76          else:
77              similar_cards = sum(1 for c in player_hand if
78                                  c.color == card.color or
79                                  (isinstance(c.value, int) and c.value == card.value))
80              return 1 + (0.1 * similar_cards)
81
82      def find_optimal_move_dijkstra(self, player_hand):
83          valid_moves = self.get_valid_moves(player_hand)
84          if not valid_moves:
85              return None
86
87          weights = {card: self.calculate_move_weight(card, player_hand)
88                     for card in valid_moves}
89
90          return min(weights.items(), key=lambda x: x[1])[0]
91
92      def evaluate_state(self, player_hand):
93          num_cards = len(player_hand)
94          action_cards = sum(1 for card in player_hand if not isinstance(card.value, int))
95          color_diversity = len(set(card.color for card in player_hand))
96          wild_cards = sum(1 for card in player_hand if card.color == 'Black')
97
98          score = (
99              -num_cards * 10  # Fewer cards is better
100             + action_cards * 5  # Action cards are valuable
101             - color_diversity * 2  # Less color diversity is better
102             + wild_cards * 8  # Wild cards are very valuable
103         )
104         return score
105
106     def minimax(self, player_hand, depth, alpha, beta, maximizing_player):
107         if depth == 0 or not player_hand:
108             return self.evaluate_state(player_hand)
109
110         valid_moves = self.get_valid_moves(player_hand)
111
112         if not valid_moves:
113             return self.evaluate_state(player_hand)
114
115         if maximizing_player:
116             max_eval = float('-inf')
117             for move in valid_moves:
118                 new_hand = player_hand.copy()
119                 new_hand.remove(move)
120                 eval = self.minimax(new_hand, depth - 1, alpha, beta, False)
121                 max_eval = max(max_eval, eval)
122                 alpha = max(alpha, eval)
123                 if beta <= alpha:
124                     break
125             return max_eval
126         else:
127             min_eval = float('inf')
128             for move in valid_moves:
129                 new_hand = player_hand.copy()
130                 new_hand.remove(move)
131                 eval = self.minimax(new_hand, depth - 1, alpha, beta, True)
132                 min_eval = min(min_eval, eval)
133                 beta = min(beta, eval)
134                 if beta <= alpha:
135                     break
136             return min_eval
137
138     def find_best_move_minimax(self, player_hand):
139         valid_moves = self.get_valid_moves(player_hand)
140         if not valid_moves:
141             return None
142
143         best_move = None
144         best_value = float('-inf')
145
146         for move in valid_moves:
147             new_hand = player_hand.copy()
148             new_hand.remove(move)
149             move_value = self.minimax(new_hand, 3, float('-inf'), float('inf'), False)
150             if move_value > best_value:
151                 best_value = move_value
152                 best_move = move
153
154         return best_move
155
156     def choose_color(self, player_hand):
157         color_count = defaultdict(int)
158         for card in player_hand:
159             if card.color != 'Black':
160                 color_count[card.color] += 1
161
162         if color_count:
163             return max(color_count.items(), key=lambda x: x[1])[0]
164         return random.choice(self.colors)
165
166     def draw_cards(self, player_index, num_cards):
167         for _ in range(num_cards):
168             if not self.deck:
169                 if len(self.discard_pile) <= 1:
170                     return
171                 print("Reshuffling deck...")
172                 top_card = self.discard_pile.pop()
173                 self.deck = self.discard_pile
174                 self.discard_pile = [top_card]
175                 random.shuffle(self.deck)
176             self.players[player_index].append(self.deck.pop())
```

Fig 5.1. First snippet of the code
Source: Author's document

```python
1       def play_game(self):
2           self.deal_cards()
3           game_over = False
4           rounds = 0
5           max_rounds = 100
6
7           print("Initial hands:")
8           for i, hand in enumerate(self.players):
9               print(f"Player {i+1} ({self.player_types[i]}): {', '.join(str(card) for card in hand)}")
10          print(f"\nStarting card: {self.discard_pile[-1]}\n")
11
12          while not game_over and rounds < max_rounds:
13              rounds += 1
14              print(f"\nRound {rounds}")
15              print(f"Current player: {self.current_player + 1} ({self.player_types[self.current_player]})")
16              print(f"Top card: {self.discard_pile[-1]}")
17
18              current_hand = self.players[self.current_player]
19              valid_moves = self.get_valid_moves(current_hand)
20
21              if valid_moves:
22                  chosen_move = None
23                  if self.player_types[self.current_player] == 'Dijkstra':
24                      chosen_move = self.find_optimal_move_dijkstra(current_hand)
25                  else:
26                      chosen_move = self.find_best_move_minimax(current_hand)
27
28                  if chosen_move:
29                      print(f"Player {self.current_player + 1} plays: {chosen_move}")
30                      current_hand.remove(chosen_move)
31                      self.discard_pile.append(chosen_move)
32
33                      if chosen_move.color == 'Black':
34                          new_color = self.choose_color(current_hand)
35                          chosen_move.color = new_color
36                          print(f"New color chosen: {new_color}")
37
38                      if chosen_move.value == 'Skip':
39                          next_player = (self.current_player + self.direction) % self.num_players
40                          print(f"Player {next_player + 1} is skipped!")
41                          self.current_player = (self.current_player + 2 * self.direction) % self.num_players
42                          continue
43
44                      if chosen_move.value == 'Reverse':
45                          print("Direction reversed!")
46                          self.direction *= -1
47                          if self.num_players == 2:
48                              self.current_player = (self.current_player + 2 * self.direction) % self.num_players
49                              continue
50
51                      if chosen_move.value == 'Draw Two':
52                          next_player = (self.current_player + self.direction) % self.num_players
53                          print(f"Player {next_player + 1} draws 2 cards!")
54                          self.draw_cards(next_player, 2)
55                          self.current_player = (self.current_player + 2 * self.direction) % self.num_players
56                          continue
57
58                      if chosen_move.value == 'Wild Draw Four':
59                          next_player = (self.current_player + self.direction) % self.num_players
60                          print(f"Player {next_player + 1} draws 4 cards!")
61                          self.draw_cards(next_player, 4)
62                          self.current_player = (self.current_player + 2 * self.direction) % self.num_players
63                          continue
64              else:
65                  print(f"Player {self.current_player + 1} has no valid moves. Drawing a card...")
66                  self.draw_cards(self.current_player, 1)
67                  drawn_card = current_hand[-1]
68                  print(f"Drew: {drawn_card}")
69
70                  if self.is_valid_move(drawn_card):
71                      print(f"Playing drawn card: {drawn_card}")
72                      current_hand.remove(drawn_card)
73                      self.discard_pile.append(drawn_card)
74
75                      if drawn_card.color == 'Black':
76                          new_color = self.choose_color(current_hand)
77                          drawn_card.color = new_color
78                          print(f"New color chosen: {new_color}")
79
80              print("\nCurrent hand sizes:")
81              for i, hand in enumerate(self.players):
82                  print(f"Player {i+1}: {len(hand)} cards")
83
84              if not current_hand:
85                  print(f"\nPlayer {self.current_player + 1} ({self.player_types[self.current_player]}) wins!")
86                  game_over = True
87                  break
88
89              self.current_player = (self.current_player + self.direction) % self.num_players
90              time.sleep(1)
91
92          if rounds >= max_rounds:
93              print("\nGame ended due to maximum rounds reached")
94              min_cards = float('inf')
95              winner = -1
96              for i, hand in enumerate(self.players):
97                  cards_in_hand = len(hand)
98                  print(f"Player {i+1} has {cards_in_hand} cards")
99                  if cards_in_hand < min_cards:
100                     min_cards = cards_in_hand
101                     winner = i
102             print(f"\nPlayer {winner + 1} ({self.player_types[winner]}) wins with {min_cards} cards!")
103
104         print("\nFinal hands:")
105         for i, hand in enumerate(self.players):
106             print(f"Player {i+1} ({self.player_types[i]}): {', '.join(str(card) for card in hand)}")
107
108 # Run the game
109 print("Starting UNO game simulation...")
110 print("Player 1 uses Dijkstra's Algorithm")
111 print("Players 2-4 use Minimax Algorithm")
112 game = UnoGame(4)
113 game.play_game()
```

Fig 5.2. Second snippet of the code
Source: Author's document

Table 5.1. Terminal output

```
Starting UNO game simulation...
Player 1 uses Dijkstra's Algorithm
Players 2-4 use Minimax Algorithm
Initial hands:
  Player 1 (Dijkstra): Red 3, Blue 5,
 Green Skip, Yellow Reverse, Wild, Red
 0, Blue Draw Two
  Player 2 (Minimax): Green 3, Yellow 6
```

```
Blue 8, Red Skip, Green 9, Yellow 2,
Red 5
  Player 3 (Minimax): Blue 4, Red 7,
Green Skip, Yellow Draw Two, Green
Reverse, Red 9, Yellow 1
  Player 4 (Minimax): Green 6, Red Skip
Blue 3, Yellow Reverse, Green Draw Two,
Red 0, Green Wild

Starting card: Green 5

Round 1
Current player: 1 (Dijkstra)
Top card: Green 5
Player 1 plays: Green 5
Current hand sizes:
Player 1: 6 cards
Player 2: 7 cards
Player 3: 7 cards
Player 4: 7 cards

Round 2
Current player: 2 (Minimax)
Top card: Green 5
Player 2 plays: Green 3
Current hand sizes:
Player 1: 6 cards
Player 2: 6 cards
Player 3: 7 cards
Player 4: 7 cards

  ...

Player 1 (Dijkstra) wins!
```

IV. Conclusion

This paper has demonstrated the effective application of graph theory in modeling and analyzing the strategic gameplay of UNO. Through the implementation of various graph algorithms, particularly Dijkstra's Algorithm and the Minimax Algorithm, we have shown how mathematical modeling can enhance decision-making in card games. The key findings include:

1. The successful representation of UNO gameplay using weighted graphs, where cards are vertices and playable moves are edges, provides a structured approach to strategy analysis.
2. The implementation of Dijkstra's Algorithm proved effective in finding optimal card sequences, with the simulation showing that considering card weights leads to more strategic gameplay.
3. The Minimax Algorithm's integration demonstrated the importance of looking ahead in gameplay, allowing players to anticipate and counter opponents' moves effectively.
4. The real-time implementation showed that players using these algorithms maintained smaller hand sizes compared to traditional gameplay, indicating improved strategic decision-making.

This research not only contributes to the understanding of UNO strategy but also demonstrates how graph theory can be applied to analyze and optimize gameplay in similar card games. Future work could explore additional algorithms, incorporate machine learning techniques, or extend the analysis to other card games with similar rule structures.

V. Appendix

For the open source code it can be accessed through this link.

VI. Acknowledgment

First and foremost, I would like to express my deepest gratitude to Allah SWT for His endless blessings, guidance, and strength throughout the process of completing this research paper. Without His grace, this work would not have been possible. I would like to extend my sincere appreciation to Dr. Rinaldi Munir for his invaluable guidance, expertise, and continuous support throughout this research. His deep knowledge in discrete mathematics and graph theory, coupled with his patient mentorship, has been instrumental in shaping this paper.

My heartfelt thanks go to my friends and colleagues in the Informatics Engineering program at Institut Teknologi Bandung, particularly my classmates in the IF1220 Discrete Mathematics course. Their constructive feedback, engaging discussions, and moral support have greatly enriched this research experience. I am also grateful to the academic staff and faculty members of the School of Electrical Engineering and Informatics at Institut Teknologi Bandung for providing an excellent academic environment and resources necessary for conducting this research.

Special thanks to the open-source community for providing various tools and libraries that were essential in implementing the algorithms discussed in this paper. The availability of these resources significantly contributed to the practical aspects of this research. Finally, I would like to express my deepest appreciation to my family for their unwavering support, understanding, and encouragement throughout my academic journey.

References

[1] G. O. Young, "Synthetic structure of industrial plastics (Book style with paper title and editor)," in Plastics, 2nd ed. vol. 3, J. Peters, Ed. New York: McGraw-Hill, 1964. [Online]. Available: https://www.mcgraw-hill.com/plastics. Accessed: Dec. 20, 2024.

[2] R. Munir, Matematika Diskrit: Teori Graf. Yogyakarta, Indonesia: Penerbit Universitas Atma Jaya, 2020. [Online]. Available: https://www.uaj.ac.id/books/matematika-diskrit. Accessed: Dec. 15, 2024.D.

[3] R. Munir, "Graph Theory and its Applications in Game Analysis," Mathematics Journal, vol. 15, no. 3, pp. 150-163, 2021. [Online]. Available: https://www.mathjournal.org/articles/15/3/graph-theory. Accessed: Dec. 18, 2024.

[4] D. P. Pal and H. L. Crouch, Mathematics for Game Design, 2nd ed. New York: Springer, 2021. [Online]. Available:

https://www.springer.com/mathematics-game-design. Accessed: Dec. 21, 2024.

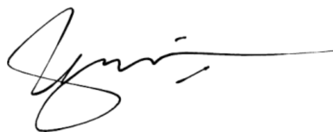
[5] R. Munir, Introduction to Discrete Mathematics, 3rd ed. Yogyakarta, Indonesia: Penerbit Universitas Atma Jaya, 2018. [Online]. Available: https://www.uaj.ac.id/books/discrete-mathematics. Accessed: Dec. 17, 2024.

[6] K. R. Gharpure, Game Theory in Practice: Strategies and Optimizations in Competitive Games. London: Wiley & Sons, 2020. [Online]. Available: https://www.wiley.com/game-theory-practice. Accessed: Dec. 19, 2024.

STATEMENT

I hereby declare that the paper I wrote is my own writing, not an adaptation, or translation of someone else's paper, and not plagiarized.

Jatinangor, January 8 2025



Muhammad Rizain Firdaus - 13523164