Analysis of Optimal Strategy in Chopsticks Game Using Graph-Based Approach

Dave Daniell Yanni - 13523003[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]d06163606@gmail.com, 13523003@std.stei.itb.ac.id

***Abstract*—This paper presents a graph-based approach to analyzing optimal strategies in the hand game Chopsticks, utilizing both Minimax algorithm and Breadth-First Search. The game states are modeled as nodes in a directed graph, with possible moves represented as edges, resulting in 583 unique states. The Minimax algorithm evaluates positions by simulating future outcomes and identifying moves that maximize winning probability while considering opponent responses. Simultaneously, BFS efficiently determines the shortest path to victory by minimizing required moves. Results demonstrate that while Minimax provides comprehensive strategic analysis accounting for opponent counter-play, BFS excels at finding quick winning sequences when available. This dual approach offers players flexibility in choosing between long-term strategic advantage and immediate tactical opportunities. The research establishes that graph-based modeling effectively optimizes decision-making in Chopsticks, with potential applications in similar turn-based games.**

***Keywords*—Graph Algorithms, Minimax, Breadth-First Search**

I. INTRODUCTION

Graph theory is a branch of mathematics that studies graphs, which are structures used to model relationships between pairs of objects. A graph consists of nodes and edges that connect these nodes, representing relationships or transitions between states

The Chopsticks game is a simple yet strategic hand game that requires careful planning and foresight to secure a win. Despite its straightforward rules, winning often depends on following a series of optimal moves, forming what can be described as winning sequences, specific combinations of actions that guarantee victory.

This research paper aims to explore the use of a graph-based approach to identify optimal strategies for the Chopsticks game. By modeling game states as nodes and possible moves as edges in a directed graph, the study seeks to analyze the underlying structure of the game and determine the best moves to maximize the chances of winning.

II. THEORETICAL BASIS

A. Graph

Graphs are structures used to model relationships between pairs of objects. A graph consists of nodes and edges that connect these nodes, representing relationships or transitions between states.

In Fig 2.1, A, B, C, and D are considered nodes, and the lines connecting them are edges.

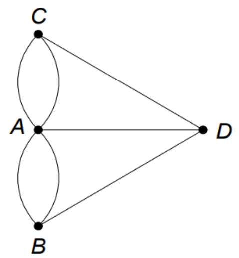


Fig. 2.1 Nodes and Edges
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

Graphs are divided into two different categories. The first one is simple graphs, a graph with only single edges, this means there are only 1 edge connecting 2 different nodes.

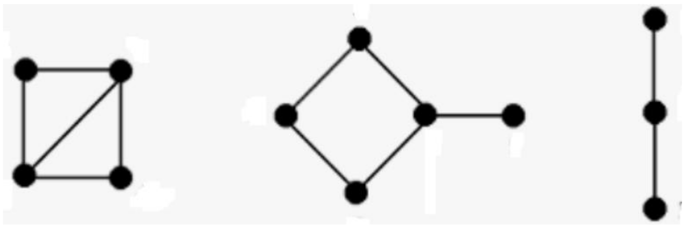


Fig. 2.2 Simple Graphs
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

The second type of graph is non simple graphs, graphs that contain a looping edge, or multiple edges connecting two different nodes.

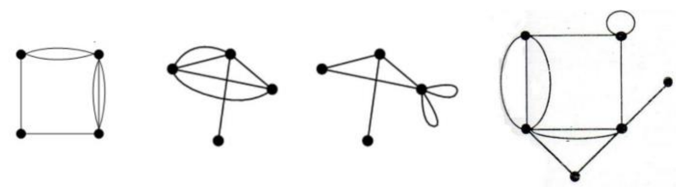


Fig. 2.3 Non-Simple Graphs
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

Non simple graphs are then differentiated into 2 different categories too, the first one is multi-graphs, graphs that have multiple edges connecting two different nodes.

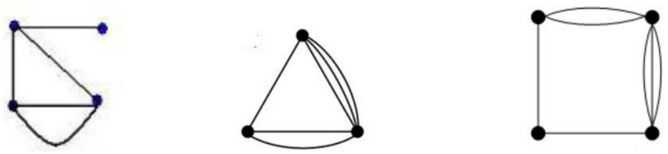


Fig. 2.3 Multi-Graphs
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

The second one is pseudo-graphs, graphs that contain a looping edge, an edge that connects to the same node.

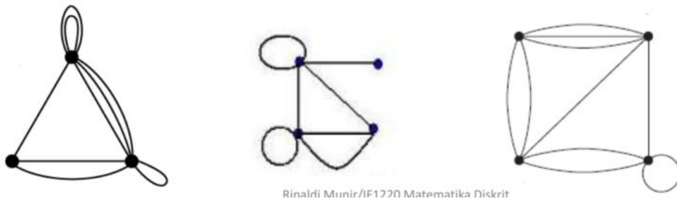


Fig. 2.3 Psuedo-Graphs
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

Graphs are also divided based on whether they have directed edges or not. The pictures below show the two

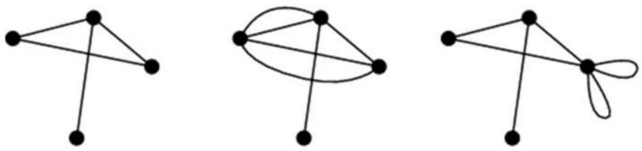


Fig. 2.4 Undirected Graphs
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

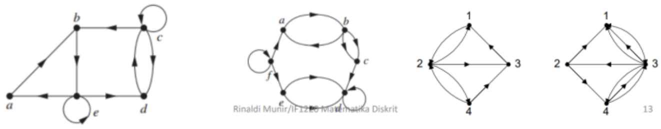


Fig. 2.5 Directed Graphs
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf

B. Graph Terminology

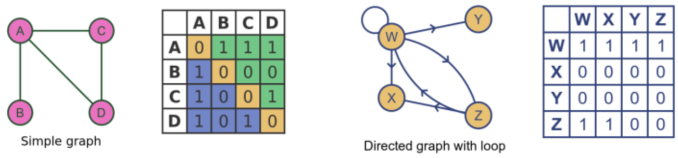
There are some terminologies used in graph theory, such as:

1. Adjacent. Two edges are adjacent if they are connected directly by an edge.
2. Incidence or intersect. An edge which is connected to a node.
3. Isolated Node. A node which is not connected to any other node.
4. Null Graph. A graph with no edges.
5. Degree. The number of edges intersecting with a node.

C. Graph Representation

Graphs can be represented through three different ways. Which includes:

1. Adjacency Matrix.



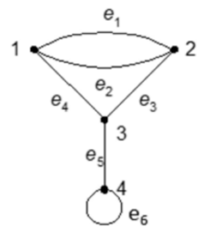
	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	1	0	0	1
D	1	0	1	0

	W	X	Y	Z
W	1	1	1	1
X	0	0	0	0
Y	0	0	0	0
Z	1	1	0	0

Fig. 2.6 Adjacency Matrix
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian2-2024.pdf

For matrix A, row i, column j, A[i, j] = 1, if node i is adjacent to node j, and A[i, j] = 0, if they are not adjacent.

2. Incidency Matrix.

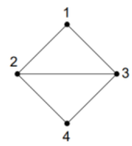


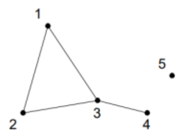
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} e1 & e2 & e3 & e4 & e5 & e6 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

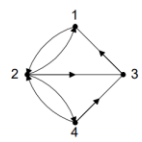
Fig. 2.7 Incidency Matrix
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian2-2024.pdf

For matrix A, row i, column j, A[i, j] = 1, if edge i intersects with node j, and A[i, j] = 0, if they do not intersect.

3. Adjacency List







Simpul	Simpul Tetangga
1	2, 3
2	1, 3, 4
3	1, 2, 4
4	2, 3

Simpul	Simpul Tetangga
1	2, 3
2	1, 3
3	1, 2, 4
4	3
5	-

Simpul	Simpul Terminal
1	2
2	1, 3, 4
3	1
4	2, 3

Fig. 2.8 Adjacency List
Source:
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian2-2024.pdf

List of every node and its adjacent nodes.

D. Chopsticks Game

Chopsticks is a two-player game where each player starts with one point on each hand. A player can add points to the opponent's hand; for example, if the current player's left hand has 1 point and the opponent's left hand has 2 points, the current player can add 1 point to the opponent's left hand, totaling 3 points. A player can also split the points between their own hands. For instance, if the player has 2 points on the left hand and 1 point on the right hand, they can split the points so that there are 3 points on the right hand and 0 points on the left hand. However, splitting points cannot simply swap values; for example, having 2 points on the right and 1 point on the left is not a valid split.

To represent the Chopsticks game as a graph, a game state is represented as a node. A game state is the current state of the game and is defined by the variables LeftCurrent, RightCurrent, LeftNext, and RightNext, where LeftCurrent represents the current player's left-hand points. The same logic applies to the other variables. Edges in the graph represent the moves made by the current player.

E. Breadth First Search

Breadth-First Search is a searching algorithm used to explore all possible adjacent nodes from a given starting node. After identifying all adjacent nodes of the starting node, it proceeds to explore the adjacent nodes of each of those nodes.

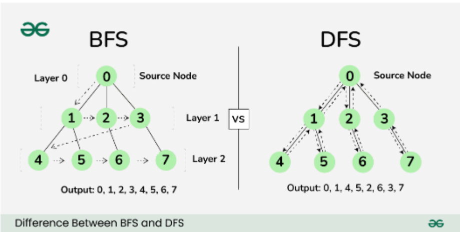


Fig. 2.9 BFS vs DFS
Source:
https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/

F. Minimax Algorithm

The Minimax Algorithm is used to find the most optimal move in a decision-based game, assuming the opponent also plays optimally.

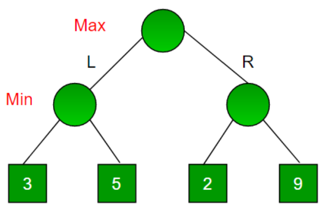


Fig. 2.10 Example Of Minimax 1
Source:
https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

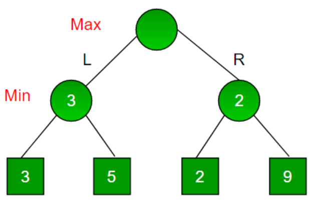


Fig. 2.11 Example Of Minimax 2
Source:
https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

This is a backtracking algorithm that starts from the edge or a leaf node. In Figure 2.10, the left subtree has nodes valued at 3 and 5. After the current player's move (left or right), the opponent will choose the lower value, as this is the most optimal move to minimize the current player's score. Similarly, in the right subtree, the chosen value would be 2. This leads to the state shown in Figure 2.11. The current player aims to maximize their points, so the left subtree is selected because it has the highest value.

III. Implementation

A. Code Description

This code is used to generate all possible nodes in the chopsticks game and write the output in a csv file. This is to represent the graph as an adjacency list.

```
class ChopsticksGame:
    def __init__(self):
        self.states = {}  # Dictionary to store all states
        self.node_counter = 1  # Counter for node IDs

    def is_valid_hand(self, value):
        """Check if a hand value is valid (0-4)"""
        return 0 <= value <= 4

     def get_next_states(self, left_current, right_current,
next_left, next_right):
          """Generate all possible next states from current
position"""
        next_states = []

        # Skip if both hands are dead (0)
        if left_current == 0 and right_current == 0:
            return next_states

        # Try all possible combinations of tapping
        for tap_from in ['left', 'right']:
            for tap_to in ['left', 'right']:
                if tap_from == 'left' and left_current == 0:
                    continue
                if tap_from == 'right' and right_current == 0:
                    continue

                # Calculate the new value after tapping
                 tap_value = left_current if tap_from == 'left' else
right_current

                new_next_left=next_left
                new_next_right=next_right

                if tap_to == 'left':
                    new_value = next_left + tap_value
                          new_next_left = 0 if new_value >= 5 else
new_value
                else:
                    new_value = next_right + tap_value
```

```
                        new_next_right = 0 if new_value >= 5 else
new_value

                # Add valid next state
                                if self.is_valid_hand(new_next_left) and
self.is_valid_hand(new_next_right):
                                        next_states.append((new_next_left,
new_next_right, left_current, right_current))

        # Add splitting as a possibility
        total = left_current + right_current
        for split_left in range(max(0, total - 4), min(4, total) +
1):
            split_right = total - split_left
                if (split_left != left_current or split_right !=
right_current) and (split_left != right_current and split_right
!= left_current):
                next_states.append((next_left, next_right, split_left,
split_right))

        return next_states

    def generate_all_states(self):
        """Generate all possible game states"""
        # Start with initial state (1,1,1,1)
        states_to_process = [(1, 1, 1, 1)]
        processed_states = set()

        while states_to_process:
            current_state = states_to_process.pop(0)
            if current_state in processed_states:
                continue

            # Add current state to processed set
            processed_states.add(current_state)

            # Get node ID for current state
            if current_state not in self.states:
                self.states[current_state] = self.node_counter
                self.node_counter += 1

            # Generate next possible states
                current_left, current_right, next_left, next_right =
current_state
                next_positions = self.get_next_states(current_left,
current_right, next_left, next_right)

            # Add new states to processing queue
            for next_pos in next_positions:
                new_state = (next_pos[0], next_pos[1], next_pos[2],
next_pos[3])
                if new_state not in processed_states:
                    states_to_process.append(new_state)

    def export_to_csv(self, filename="chopsticks_states.csv"):
        """Export the generated states to a CSV file"""
        with open(filename, 'w') as f:
            # Write header
            f.write('Nodes,Directed Adjacent Nodes,"Game State
(LeftCurrent, RightCurrent, LeftNext, RightNext)"\n')
```

```
            # Write each state
                for state, node_id in sorted(self.states.items(),
key=lambda x: x[1]):
                # Get next possible states
                current_left, current_right, next_left, next_right =
state
                next_positions = self.get_next_states(current_left,
current_right, next_left, next_right)

                # Convert next positions to node IDs
                    adjacent_nodes = set()  # Use a set to avoid
duplicates
                for next_pos in next_positions:
                            next_state = (next_pos[0], next_pos[1],
next_pos[2], next_pos[3])
                    if next_state in self.states:
                        adjacent_nodes.add(self.states[next_state])

                # Sort the adjacent nodes numerically and write the
row
                                adjacent_str = ",".join(map(str,
sorted(adjacent_nodes))) if adjacent_nodes else "-1"
                f.write(f'{node_id},"{adjacent_str}","{state[0]},{s
tate[1]},{state[2]},{state[3]}"\n')

# Generate and export the states
game = ChopsticksGame()
game.generate_all_states()
game.export_to_csv()

# Print some statistics
print(f"Total number of unique states: {len(game.states)}")
```

There are 583 unique nodes representing game states. However, only the first 50 nodes are displayed above for brevity. Readers can run the provided code to generate and view the complete list of nodes.

```
Nodes,Directed Adjacent Nodes,"Game State (LeftCurrent,
RightCurrent, LeftNext, RightNext)"
1,"2,3,4,5","1,1,1,1"
2,"6,7,8,9,10,11","2,1,1,1"
3,"10,11,12,13,14,15","1,2,1,1"
4,"3,16,17,18","1,1,0,2"
5,"2,19,20,21","1,1,2,0"
6,"22,23,24,25,26,27,28","3,1,2,1"
7,"26,27,28,29,30,31,32","1,3,2,1"
8,"6,33,34,35,36,37","2,1,2,1"
9,"14,36,37,38,39,40","1,2,2,1"
10,"41,42,43,44","1,1,0,3"
11,"45,46,47,48","1,1,3,0"
12,"7,35,49,50,51,52","2,1,1,2"
13,"15,38,51,52,53,54","1,2,1,2"
14,"25,55,56,57,58,59,60","3,1,1,2"
15,"30,58,59,60,61,62,63","1,3,1,2"
16,"64,65,66,67","0,3,1,1"
17,"68,69,70","0,2,0,2"
18,"68,71,72","0,2,2,0"
19,"66,67,73,74","3,0,1,1"
20,"70,75,76","2,0,0,2"
21,"72,75,77","2,0,2,0"
```

```
22,"78,79","0,1,3,1"
23,"80,81,82,83","2,4,3,1"
24,"22,84,85,86,87,88,89","3,1,3,1"
25,"24,29,87,89,90,91","2,2,3,1"
26,"50,92,93,94,95","2,1,0,4"
27,"34,49,92,96,97,98","2,1,2,2"
28,"33,93,96,99,100","2,1,4,0"
29,"56,101,102,103,104,105,106","3,1,1,3"
30,"57,61,91,104,106,107","2,2,1,3"
31,"108,109","0,1,1,3"
32,"81,110,111,112","2,4,1,3"
33,"113,114,115,116,117,118","4,1,2,1"
34,"119,120,121,122,123,124","2,3,2,1"
35,"26,28,125,126,127,128","2,2,2,1"
36,"7,34,93,129,130,131","2,1,0,3"
37,"6,49,93,132,133,134","2,1,3,0"
38,"58,60,135,136,137,138","2,2,1,2"
39,"116,139,140,141,142,143","4,1,1,2"
40,"144,145,146,147,148,149","2,3,1,2"
41,"127,137,150,151,152,153,154","1,3,1,1"
42,"153,155,156,157,158","0,4,1,1"
43,"159,160,161,162","0,3,0,2"
44,"160,163,164,165","0,3,2,0"
45,"153,157,158,166,167","4,0,1,1"
46,"85,103,128,138,152,153,154","3,1,1,1"
47,"161,162,168,169","3,0,0,2"
48,"164,165,169,170","3,0,2,0"
49,"123,124,171,172,173,174","3,2,2,1"
50,"117,118,175,176,177,178","1,4,2,1"
…
```

This code is used to visualize the graph from the data in the csv fille.

```
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd

class ChopsticksGraphVisualizer:
    def __init__(self, filename="chopsticks_states.csv"):
        self.filename = filename
        self.graph = nx.DiGraph()

    def load_states_from_csv(self):
        """Load states from the exported CSV file and build the
graph."""
        df = pd.read_csv(self.filename)

        for _, row in df.iterrows():
            node_id = str(row['Nodes']).strip()
            adjacent_nodes_str = str(row['Directed Adjacent
Nodes']).strip("")
            adjacent_nodes = adjacent_nodes_str.split(',') if
adjacent_nodes_str != "-1" else []

            # Debugging print statements
            print(f"Node ID: {node_id}, Adjacent Nodes:
{adjacent_nodes}")

            # Ensure node_id is not empty or zero
            if node_id and node_id != "0":
                self.graph.add_node(node_id, label=node_id)
                for adjacent in adjacent_nodes:
                    if adjacent and adjacent != "0":
                        self.graph.add_edge(node_id, adjacent)

    def visualize(self):
        """Visualize the graph using NetworkX and
Matplotlib."""
        pos = nx.spring_layout(self.graph)  # Layout for nodes
        labels = nx.get_node_attributes(self.graph, 'label')

        plt.figure(figsize=(12, 8))
        nx.draw(
            self.graph, pos, with_labels=False, node_size=500,
node_color='skyblue',
            font_weight='bold', arrowsize=10, edge_color='gray'
        )
        nx.draw_networkx_labels(self.graph, pos, labels,
font_size=8)

        plt.title("Chopsticks Game State Graph")
        plt.show()

# Instantiate and visualize
visualizer =
ChopsticksGraphVisualizer("chopsticks_states.csv")
visualizer.load_states_from_csv()
visualizer.visualize()
```

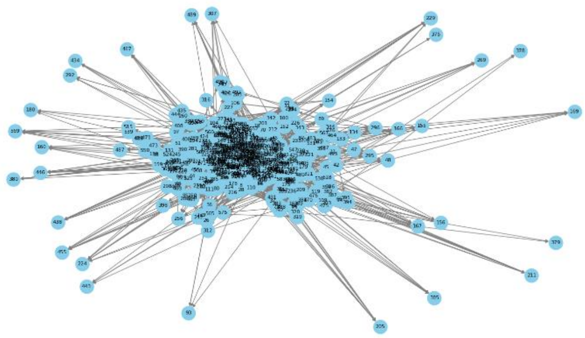


Fig. 3.1 Chopsticks Game Representation

The visualization appears cluttered due to the high density of nodes and edges. Readers are encouraged to run the provided code to generate and explore a clearer version of the graph.

```
from collections import defaultdict, deque

class ChopsticksGame:
    def __init__(self):
        self.states = {}
        self.node_counter = 1
        self.graph = defaultdict(list)

    def is_valid_hand(self, value):
        """Check if a hand value is valid (0-4)"""
        return 0 <= value <= 4

        def get_next_states(self, left_current, right_current,
next_left, next_right):
```

```
        """Generate all possible next states from current position"""
        next_states = []

        if left_current == 0 and right_current == 0:
            return next_states

        # Handle taps
        for tap_from in ['left', 'right']:
            for tap_to in ['left', 'right']:
                if tap_from == 'left' and left_current == 0:
                    continue
                if tap_from == 'right' and right_current == 0:
                    continue

                new_next_left = next_left
                new_next_right = next_right

                tap_value = left_current if tap_from == 'left' else right_current

                if tap_to == 'left':
                    new_value = next_left + tap_value
                    new_next_left = 0 if new_value >= 5 else new_value
                else:
                    new_value = next_right + tap_value
                    new_next_right = 0 if new_value >= 5 else new_value

                if self.is_valid_hand(new_next_left) and self.is_valid_hand(new_next_right):
                    next_states.append((new_next_left, new_next_right, left_current, right_current))

        # Handle splits
        total = left_current + right_current
        for split_left in range(max(0, total - 4), min(4, total) + 1):
            split_right = total - split_left
            if (split_left != left_current or split_right != right_current) and \
               self.is_valid_hand(split_left) and self.is_valid_hand(split_right):
                next_states.append((next_left, next_right, split_left, split_right))

        return next_states

    def generate_all_states(self):
        """Generate all possible game states and their connections"""
        states_to_process = [(1, 1, 1, 1)]
        processed_states = set()

        while states_to_process:
            current_state = states_to_process.pop(0)
            if current_state in processed_states:
                continue

            processed_states.add(current_state)
```

```
            if current_state not in self.states:
                self.states[current_state] = self.node_counter
                self.node_counter += 1

            next_states = self.get_next_states(*current_state)

            for next_state in next_states:
                if next_state not in processed_states:
                    states_to_process.append(next_state)
                self.graph[current_state].append(next_state)

    def is_terminal(self, state):
        """Check if the state is terminal (game over)"""
        left_current, right_current, next_left, next_right = state
        return (left_current == 0 and right_current == 0) or (next_left == 0 and next_right == 0)

    def evaluate_state(self, state, moves):
        """Evaluate terminal states considering number of moves"""
        left_current, right_current, next_left, next_right = state

        if left_current == 0 and right_current == 0:
            return 1000 - moves if moves % 2 == 1 else -1000 + moves
        if next_left == 0 and next_right == 0:
            return -1000 + moves if moves % 2 == 1 else 1000 - moves
        return 0

    def minimax(self, state, depth, alpha, beta, maximizing_player, moves=0):
        """Minimax algorithm with alpha-beta pruning"""
        if depth == 0 or self.is_terminal(state):
            return self.evaluate_state(state, moves), None

        best_move = None
        if maximizing_player:
            max_eval = float('-inf')
            for next_state in self.graph[state]:
                eval_score, _ = self.minimax(next_state, depth - 1, alpha, beta, False, moves + 1)
                if eval_score > max_eval:
                    max_eval = eval_score
                    best_move = next_state
                alpha = max(alpha, eval_score)
                if beta <= alpha:
                    break
            return max_eval, best_move
        else:
            min_eval = float('inf')
            for next_state in self.graph[state]:
                eval_score, _ = self.minimax(next_state, depth - 1, alpha, beta, True, moves + 1)
                if eval_score < min_eval:
                    min_eval = eval_score
                    best_move = next_state
                beta = min(beta, eval_score)
                if beta <= alpha:
                    break
```

```
        return min_eval, best_move

    def find_shortest_winning_path(self, state):
        """Find the shortest path to victory"""
        queue = deque([(state, [], 0)])
        visited = {state: 0}

        while queue:
            current_state, path, moves = queue.popleft()

            left_current, right_current, next_left, next_right = current_state
            if left_current == 0 and right_current == 0 and moves % 2 == 1:
                return path[0] if path else None, moves
            elif next_left == 0 and next_right == 0 and moves % 2 == 0:
                return path[0] if path else None, moves

            for next_state in self.graph[current_state]:
                if next_state not in visited or visited[next_state] > moves + 1:
                    visited[next_state] = moves + 1
                    new_path = path + [next_state] if not path else path
                    queue.append((next_state, new_path, moves + 1))

        return None, float('inf')

    def analyze_position(self, left_current, right_current, left_next, right_next, depth=5):
        """Analyze position using minimax and shortest path"""
        self.generate_all_states()
        current_state = (left_current, right_current, left_next, right_next)

        minimax_value, minimax_move = self.minimax(current_state, depth, float('-inf'), float('inf'), True)

        shortest_move, moves_to_win = self.find_shortest_winning_path(current_state)

        print("\nPosition Analysis:")
        print(f"Current State: {current_state}")

        print("\n1. Minimax Analysis:")
        print(f"Best Move: {minimax_move}")
        print(f"Evaluation: {minimax_value}")

        print("\n2. Shortest Path Analysis:")
        if shortest_move:
            print(f"Best Move: {shortest_move}")
            print(f"Moves to win: {moves_to_win}")
        else:
            print("No guaranteed winning path found")

        if shortest_move:
            return shortest_move, f"Winning in {moves_to_win} moves"
        elif minimax_value > 0:
            return minimax_move, "Winning position (Minimax)"
        elif minimax_move:
            return minimax_move, "Best defensive move"
        return None, "No moves available"

game = ChopsticksGame()
best_move, strategy = game.analyze_position(1, 1, 3, 4) #initial state
print(f"\nFinal Recommendation:")
print(f"Best move: {best_move}")
print(f"Strategy: {strategy}")
```

This code is to simulate a current state in a game, and it will find the next best move to make or the next best possible state. There are two algorithms used to find the best move, the first one is using bfs algorithm to find the shortest path to victory, and the second one is minimax algorithm to find the best move based on the evaluation points.

IV. Results and Analysis

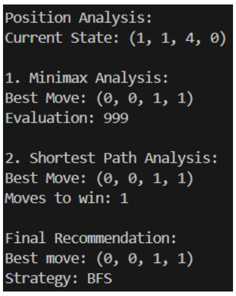


Fig. 4.1 Experiment Results 1

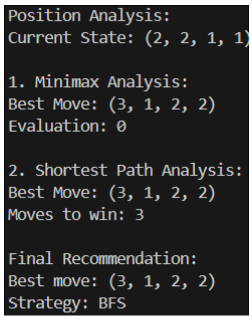


Fig. 4.2 Experiment Results 2

In this study, the optimal strategies for the Chopsticks game were identified using two graph-based approaches: the Minimax algorithm with alpha-beta pruning and Breadth-First Search for finding the shortest path to victory. The entire game was modeled as a directed graph, where nodes represent game states and edges represent possible moves. A total of 583 unique nodes were generated, representing all possible combinations of hand points.

The Minimax algorithm evaluated each state by simulating all possible outcomes up to a specified depth. It identified moves that maximized the player's chance of winning while accounting for the opponent's best responses. For example, starting from the initial state (1, 1, 1, 1), the algorithm suggested moves that led to either an immediate win or a path with minimal risk.

Using BFS, the shortest path to a winning state was determined by minimizing the number of moves required. The algorithm efficiently found paths that guaranteed victory if executed correctly. Starting from state (1, 1, 4, 0), BFS identified a path with a length of 1 moves to secure a win (refer to Fig. 4.1).

V. Discussion

The results highlight the effectiveness of modeling the Chopsticks game as a directed graph. By using Minimax, players can make decisions that account for both offensive and defensive strategies. This approach, however, is computationally intensive and depends on depth-limited searches, which may not explore all possible future outcomes. Conversely, BFS guarantees finding the shortest path to victory but assumes that the opponent plays suboptimally, which limits its applicability in real-world scenarios with skilled players.

The Minimax strategy offers a flexible framework for evaluating multiple moves and their long-term consequences, making it ideal for strategic depth. In contrast, BFS is best suited for scenarios where immediate results are prioritized. The choice of strategy depends on the player's goal, whether to maximize long-term advantage or secure a quick win.

VI. Conclusion

This research demonstrates that graph-based approaches are powerful tools for optimizing gameplay strategies in the Chopsticks game. By modeling game states as nodes and transitions as directed edges, both Minimax and BFS algorithms can effectively guide decision-making. Minimax provides a comprehensive analysis by considering all possible outcomes, while BFS offers a fast solution for finding guaranteed winning sequences.

Future work could explore hybrid strategies that combine the strengths of both methods, balancing computational efficiency with strategic depth. Additionally, incorporating probabilistic modeling to handle uncertainties in opponent behavior would further enhance strategy formulation.

VI. Acknowledgment

The author extends heartfelt gratitude to God for providing wisdom, perseverance, and opportunity to complete this paper successfully. Sincere appreciation is all extended to Mr. Dr. Ir. Rinaldi Munir, M.T., as the lecturer of the IF1220 Discrete Mathematics course.

References

[1] Munir, Rinaldi. 2024. "Graf (Bagian 1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf (accessed on 6 December 2024).

[2] Munir, Rinaldi. 2024. "Graf (Bagian 2)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf \ (accessed on 6 December 2024).

[3] Breadth First Search or BFS for a Graph. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/ (accessed on 8 December 2024).

[4] Minimax Algorithm in Game Theory. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/ (accessed on 8 December 2024).

[5] Difference between BFS and DFS. https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/ (accessed on 8 December 2024).

Statement

I hereby declare that this paper is my own work, not a paraphrase or translation of someone else's paper, and not plagiarism.

Bandung, 8 January 2025

Dave Daniell Yanni 13523003