Graph and Logical Implication-Based Patterns in the Design of a Minesweeper Engine

Haegen Quinston - 13523109[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]haegenquinston@gmail.com [2]13523109@stei.itb.ac.id

***Abstract*—Minesweeper, a classic logic-based video game, challenges players to deduce mine locations on a grid using numerical clues. Automating Minesweeper gameplay presents a unique problem-solving domain that blends logical reasoning and computational efficiency. This paper introduces a novel Minesweeper engine designed to autonomously solve boards using graph theory and logical implications as its foundation. The engine identifies patterns and derives decisions by treating each cell as a node in a graph, connected by logical dependencies. Advanced techniques, including propositional logic, rules of inference, and pattern recognition, allow the engine to replicate human-like reasoning while handling the inherent uncertainty of Minesweeper's gameplay. The paper demonstrates the engine's ability to solve grids of varying complexities, offering insights into automated decision-making systems.**

***Keywords*—graph theory, logical implication, engine, solve grids**

I. Introduction

Minesweeper, initially introduced with Windows 3.11 in 1990, has captivated players with its blend of logic, strategy, and unpredictability. The game's core mechanics require players to deduce the locations of hidden mines using numerical clues that indicate adjacent mine counts [1]. Despite its simplicity, Minesweeper presents significant challenges for automation due to its nondeterministic polynomial (NP) complexity and the need to make decisions under uncertainty.

This research focuses on the development of an automated Minesweeper engine capable of solving boards autonomously. Unlike traditional approaches aimed at optimizing human efficiency, such as minimizing clicks or maximizing chording efficiency, this work emphasizes designing a computational framework for logical deduction and decision-making. The engine employs graph theory to represent the grid as a network of logical relationships, where nodes represent cells, and edges represent adjacency connections. Logical implications and rules of inference underpin its ability to recognize patterns and deduce safe moves or mine placements.

The paper explores how this engine systematically identifies basic and advanced patterns, such as the "1-1" and "1-2" configurations, as well as complex formations like holes and triangles. These patterns, derived from the logical dependencies encoded in Minesweeper's rules, guide the engine's decision-making process. Furthermore, an educated guessing mechanism is implemented to address cases where logical deductions are insufficient, ensuring that the engine can complete games successfully even under uncertainty.

By integrating principles from graph theory and propositional logic, this Minesweeper engine serves as a robust framework for exploring automated reasoning in uncertain environments, offering potential applications in AI-driven problem-solving and logical decision systems.

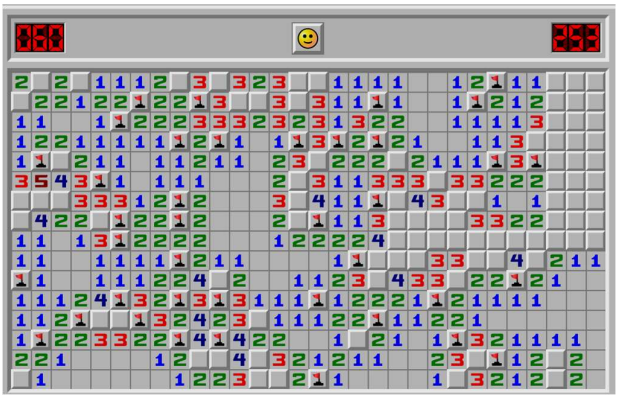

Fig. 1.1 An Expert level game of Minesweeper, the top right icon showing the time, and the top left icon showing the remainder of mines

II. Theoretical Foundations

A. Logic

Logic is based on the relationships between sentences or statements. Only sentences that can be evaluated as either true or false are considered, and these are referred to as propositions. A proposition is a statement that has a definite truth value, meaning it is either true (T) or false (F), but not both. [5]

Propositions can be categorized into several types:

a. Atomic proposition

An atomic proposition is a basic, indivisible statement that cannot be broken down further. It expresses a single idea and has a definite truth value, either true or false. Atomic propositions do not involve any logical operators, making them the fundamental building blocks of propositional logic. For example, “It is raining” is an atomic proposition because it represents a single fact. [5]

b. Compound proposition

A compound proposition is formed by combining two or more atomic propositions using logical connectives such as AND (∧), OR (∨), and NOT (¬). These propositions allow for the representation of more complex logical relationships. For instance, the statement "It is raining AND I will take an umbrella" combines two atomic propositions into one compound proposition. [5]

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

p	$\sim p$
T	F
F	T

Fig. 2.1 The truth table for conjunction (leftmost table), disjunction (center table), and negation (rightmost table) (Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

c. Implication

An implication is a specific type of compound proposition that describes a conditional relationship between two propositions. It is denoted using the logical operator IMPLIES (→) and is often expressed as "If *p*, then *q*." In this context, *p* is called the antecedent, and *q* is called the consequent. For example, the implication "If it rains, then I will take an umbrella" establishes a cause-and-effect relationship. The truth value of an implication depends on the truth values of its components and is typically true unless the antecedent is true while the consequent is false. [5]

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

Fig. 2.2 The truth table for an implication $p \rightarrow q$ (Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

d. Arguments & Rules of Inference

An argument in propositional logic is a sequence of propositions, called premises, followed by a conclusion. The goal of an argument is to demonstrate that the conclusion logically follows from the premises. An argument is considered valid if, whenever all the premises are true, the conclusion must also be true. [5]

Rules of inference are standardized logical patterns that ensure the validity of arguments. These rules provide a framework for deriving conclusions from given premises. Some common rules of inference include:

1) Modus Ponens: If p → q and p is true, then q must also be true.
2) Modus Tollens: If p → q and q is false, then p must also be false.
3) Disjunctive Syllogism: If p ∨ q and ¬ p (not p), then q must be true.

B. Graph

A graph is defined as a pair of sets *G = (V, E)* where V is a set of vertices/nodes, and E is an *edge* consisting of unordered pairs of vertices. [6]

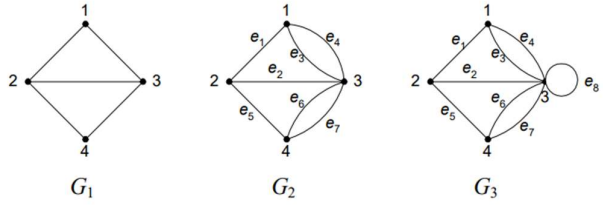

Fig. 2.3 (a) simple graph, (b) unsimple graph, (c) pseudo-graph (Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

Graphs can be classified into two types based on the presence of loops or multiple edges:

a. Simple graph

A simple graph is a graph that does not contain loops or multiple edges. An example is shown in Fig. 2.3(a).

b. Unsimple graph

An unsimple graph is a graph that contains either loops or multiple edges. Unsimple graphs can be further categorized as follows:

1. Multigraph

   A multigraph is a graph that contains multiple edges, which are two or more edges connecting the same pair of nodes. An example is shown in Fig. 2.3(b). [6]

2. Pseudograph

   A pseudograph is a graph that contains loops, which are edges that connect a node to itself. An example is shown in Fig. 2.3(c). [6]

Another way to classify graphs is based on the orientation of their edges. Using this approach, graphs are divided into two types:

a. Undirected graph

An undirected graph is a graph in which the edges have no directional orientation. In such a graph, the order of the nodes connected by an edge does not matter. For an edge connecting two nodes u and v in a graph, (u, v) is equivalent to (v, u). Examples are shown in Fig 2.3. [6]

b. Directed graph

A directed graph is a graph where each edge has a specific directional orientation. In this type of graph, the order of the nodes matters. For an edge connecting two nodes u and v, (u, v) and (v, u) represent distinct edges. In the case of the edge (u, v), u is called the origin node, and v is called the terminal node. Examples are shown in Fig 2.4. [6]

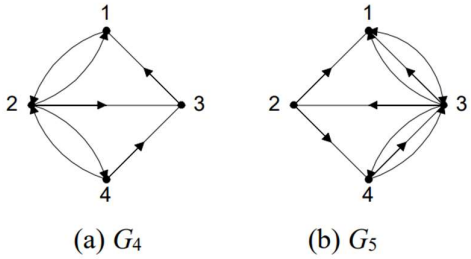

Fig. 2.4 Examples of directed graphs (Source: Matematika Diskrit Edisi 3 2010 (Dr. Rinaldi Munir))

C. Minesweeper

Minesweeper is a logic-based puzzle game that challenges players to uncover cells on a grid while avoiding hidden mines. Each revealed cell displays a number indicating the count of adjacent mines, or it may be blank if no adjacent mines exist. The objective is to identify all non-mine cells while employing logical reasoning to deduce safe cells and mark mine locations.

The complexity of Minesweeper stems from its reliance on deductive logic and decision-making under uncertainty. Addressing these challenges necessitates the use of structured patterns and logical implications as the foundation for systematic problem-solving. These patterns are derived from the relationships between numbers and their adjacent unrevealed cells, providing a logical framework for determining safe moves and identifying mines. [1]

Minesweeper patterns can be classified into basic and advanced categories. Basic patterns, such as B1-B2 or 1-1, are based on direct numerical relationships that enable the deduction of mine placements or safe cells. These patterns are rooted in propositional logic, where a clue serves as a premise leading to definitive conclusions about its neighboring cells. Advanced patterns, including holes and triangles, involve overlapping relationships among multiple numbers and their shared neighbors. These patterns are better conceptualized through graph theory, where the grid is represented as a network of nodes (cells) connected by edges (adjacency relationships). By treating the grid as a graph, solvers can apply rules of inference to navigate constraints, ensuring consistent and accurate deductions. [2]

a. Basic Patterns

1) B1-B2 Pattern

When a cell's number matches the count of its adjacent unrevealed cells (B1), all those cells must contain mines. Conversely, if the number matches the count of adjacent flagged cells (B2), all other unrevealed neighbors can be safely opened. [2]

Fig. 2.5 A B1 Pattern

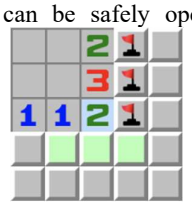

Fig. 2.6 A B2 Pattern

2) 1-1 and 1-2 Pattern

These patterns extend logical deductions. For example, in the 1-1 pattern, two adjacent "1" cells sharing unrevealed neighbors imply one mine in the shared cells. The 1-2 pattern further explores the relationship by identifying mine placements and safe cells beyond the shared neighbors. [2]

Fig 2.7 A 1-1 Pattern

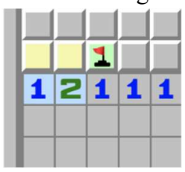

Fig. 2.8 A 1-2 Pattern

3) 1-2-1 and 1-2-2-1 Pattern

The 1-2-1 and 1-2-2-1 patterns extend the logic of the 1-2 pattern, applying it symmetrically from the left and right sides. [2]

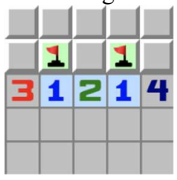

Fig. 2.9 A 1-2-1 Pattern

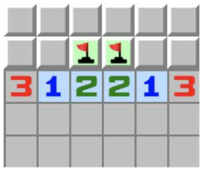

Fig. 2.10 A 1-2-2-1 Pattern

b. Reduction Patterns

Reduction patterns simplify complex arrangements by conceptually reducing numbers based on flagged cells. For instance, a "2" surrounded by one flagged cell can be reduced to "1," facilitating additional deductions. [2]

Fig. 2.11 A 1-2-1 reduction pattern

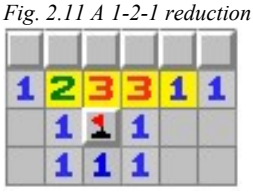

Fig. 2.12 A 1-2-2-1 reduction pattern

c. Holes & Triangles

1) Holes (H1, H2, H3)

These patterns involve numerical clues forming "holes" in the grid. Shared cells between adjacent numbers satisfy mine requirements, allowing other cells to be safely opened. [4]

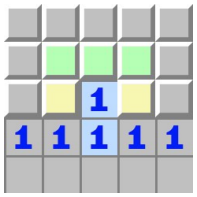

Fig 2.13 an H1 Pattern

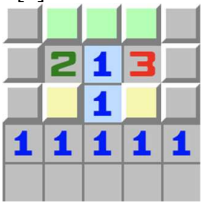

Fig 2.14 An H2 Pattern

Like the H1 and H2 patterns, on an H3 pattern, the bottom "1" confirms that the two yellow cells contain one mine. The top "1" is guaranteed to have its mine by these same yellow cells. All remaining cells can be safely opened. [4]

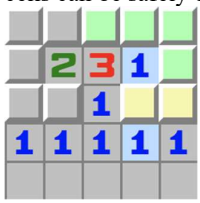

Fig 2.15 An H3 Pattern

2) Triangles (T1, T2, T3)

Similar to holes, triangle patterns involve groups of cells shared between adjacent numbers. Logical deductions based on overlaps enable solvers to safely open additional cells or mark mines. [4]

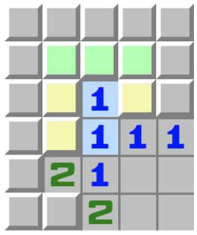

Fig 2.16 A T1 Pattern

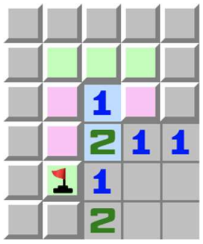

Fig 2.17 A T2 Pattern

In the T3 pattern, the "3" touches two mines in the purple cells. Since the "2" also touches these purple cells, the green-marked cells can be safely opened. [4]

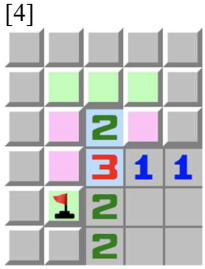

Fig 2.18 A T3 Pattern

d. Endgame – Mine counting

In the final stages of a minesweeper game, mine counting can be implemented to solve the placement of the remaining mines and help identify safe cells. This method leverages the remaining mine count, and the grouping of adjacent cells based on the remaining mines to ensure accurate deductions. [4]

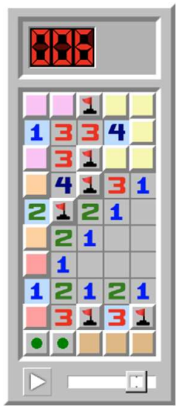

Fig 2.20 An implementation of mine counting

These theoretical patterns and logical principles serve as the foundation for the automated Minesweeper engine, allowing it to replicate human-like reasoning and efficiently solve the game while handling uncertainty.

III. Implementation Method

A. Minesweeper Module

1) The Grid

The `MinesweeperGrid` class represents the core structure of the Minesweeper board. It generates the underlying solution grid, including random mine placement and adjacency calculation. The grid serves as a fixed reference throughout the game and is primarily focused on generating and displaying the board, with no gameplay or interaction elements. Mines are represented as "M," while safe cells display the number of adjacent mines, forming the "answer key" for Minesweeper gameplay and simulation. By separating the solution grid from gameplay logic, the `MinesweeperGrid` ensures clarity and modularity in the program design.

```
class MinesweeperGrid:
    def __init__(self, rows, cols, mine_count):
        self.rows = rows
        self.cols = cols
        self.mine_count = mine_count
        self.board = [[0] * cols for _ in range(rows)]
        self.mines = set()
        self.place_mines()
        self.calculate_numbers()

    def place_mines(self):
        while len(self.mines) < self.mine_count:
            mine = random.randint(0, self.rows * self.cols - 1)
            self.mines.add(mine)

    def calculate_numbers(self):
        for mine in self.mines:
            x, y = divmod(mine, self.cols)
            self.board[x][y] = -1
            for dx in [-1, 0, 1]:
                for dy in [-1, 0, 1]:
                    nx, ny = x + dx, y + dy
                    if 0 <= nx < self.rows and 0 <= ny < self.cols and self.board[nx][ny] != -1:
                        self.board[nx][ny] += 1

    def display_board(self):
        for row in self.board:
            print(" ".join(str(cell) if cell != -1 else "M" for cell in row))
```

Fig 3.1 The code for the class MinesweeperGrid

2) The Game

The MinesweeperGame class is a Python implementation that handles the interactive gameplay mechanics of Minesweeper. Utilizing the static solution grid provided by the MinesweeperGrid class, it creates a dynamic grid where all cells are initially hidden, represented by the symbol "*". Players interact with this grid through two primary actions: revealing cells or flagging suspected mines. When a cell is revealed, the grid updates to display the number of adjacent mines, an empty cell (if no adjacent mines exist), or a mine (which ends the game). Additionally, the class implements safe-zone propagation for zero-valued cells, ensuring recursive reveals of neighboring cells. It tracks the game state, including revealed cells, flagged cells, and win/loss conditions, making it a comprehensive representation of Minesweeper gameplay.

```python
from MinesweeperGrid import MinesweeperGrid

class MinesweeperGame:
    UNOPENED_CELL = '*'  # Symbol for unopened cells

    def __init__(self, rows, cols, mine_count):
        self.grid = MinesweeperGrid(rows, cols, mine_count)
        self.rows = rows
        self.cols = cols
        self.mine_count = mine_count
        self.visible_grid = [[self.UNOPENED_CELL for _ in range(cols)] for _ in range(rows)]
        self.revealed = set()
        self.flagged = set()
        self.game_over = False
        self.win = False

    def reveal_cell(self, x, y):
        if (x, y) in self.revealed or (x, y) in self.flagged or self.game_over:
            return
        if self.grid.board[x][y] == -1:
            self.game_over = True
            self.visible_grid[x][y] = 'M'
            print("Game Over! You hit a mine.")
            return
        self.revealed.add((x, y))
        self.visible_grid[x][y] = str(self.grid.board[x][y])
        if self.grid.board[x][y] == 0:
            self._reveal_safe_zone(x, y)
        self.check_win()

    def flag_cell(self, x, y):
        if (x, y) in self.revealed or self.game_over:
            return
        if (x, y) in self.flagged:
            self.flagged.remove((x, y))
            self.visible_grid[x][y] = self.UNOPENED_CELL
        else:
            self.flagged.add((x, y))
            self.visible_grid[x][y] = 'F'

    def _reveal_safe_zone(self, x, y):
        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                nx, ny = x + dx, y + dy
                if 0 <= nx < self.rows and 0 <= ny < self.cols and (nx, ny) not in self.revealed:
                    self.reveal_cell(nx, ny)

    def check_win(self):
        if len(self.revealed) == self.rows * self.cols - self.mine_count:
            self.win = True
            self.game_over = True
            print("Congratulations! You solved the Minesweeper grid.")

    def display_visible_grid(self):
        for row in self.visible_grid:
            print(" ".join(cell if cell != ' ' else self.UNOPENED_CELL for cell in row))
        print()
```

Fig 3.2 The code for Minesweeper Game

3) Moves

The Minesweeper engine automates gameplay using the methods $reveal\_cell(self, x, y)$ and $flag\_cell(self, x, y)$. These methods handle the fundamental operations of revealing a cell or marking it as a potential mine. Designed for full automation, these commands operate independently of human interaction and are integral to the solving logic of the engine. They enable the simulation of a player's decisions, efficiently guiding the engine through the game. Each move of the engine will be printed out on the terminal by order.

4) Simulator

To evaluate the engine's performance, a dedicated simulator was developed to automate and analyze multiple gameplay iterations across varying difficulty levels. The simulator allows the engine to play through a specified number of Minesweeper games for each difficulty—beginner, intermediate, and expert—recording the outcomes and calculating the win rate for each category.

```python
from MinesweeperEngine import MinesweeperEngine

def simulate_win_rate(difficulty, rows, cols, mines, iterations):
    wins = 0
    for i in range(iterations):
        engine = MinesweeperEngine(rows, cols, mines)
        engine.solve()
        if engine.game.win:
            wins += 1
        print(f"Game {i + 1}/{iterations}: {'Win' if engine.game.win else 'Loss'}")
    win_rate = (wins / iterations) * 100
    return win_rate

def main():
    difficulties = {
        "Beginner": (9, 9, 10),
        "Intermediate": (16, 16, 40),
        "Expert": (16, 30, 99)
    }
    print("Minesweeper Simulator\n")
    print("Choose your number of iterations:")
    iterations = int(input())
    print("Simulating Minesweeper Win Rates...\n")
    for difficulty, (rows, cols, mines) in difficulties.items():
        print(f"Simulating {difficulty} Level ({iterations} games)")
        win_rate = simulate_win_rate(difficulty, rows, cols, mines, iterations)
        print(f"{difficulty} Level Win Rate: {win_rate:.2f}%\n")

if __name__ == "__main__":
    main()
```

Fig 3.3 The code for the win-rate simulator.

B. Pattern-Based Algorithm

The foundation of Minesweeper pattern recognition lies in graph theory and logical implications. Each cell on the Minesweeper grid can be viewed as a node in a graph, with edges representing adjacency relationships. Logical dependencies between nodes (cells) are established through the numerical clues provided by the game. For instance, a cell marked "1" implies that exactly one mine exists among its neighbors. By representing these relationships as a graph, the engine can systematically analyze connections to deduce safe moves and mine placements.

```python
class MinesweeperEngine:
    def __init__(self, rows, cols, mine_count):
        self.game = MinesweeperGame(rows, cols, mine_count)

    def solve(self):
        print("Starting Minesweeper Solver...")
        while not self.game.game_over:
            move_made = self.simulate_solver()
            if not move_made:
                print("No safe moves detected, stopping.")
                break
            self.game.display_visible_grid()
        if self.game.win:
            print("Solver won the game!")
        else:
            print("Solver lost the game.")

    def simulate_solver(self):
        move_made = False
        processed = set()  # Track cells that have been processed

        # Step 1: Safe reveals
        for x in range(self.game.rows):
            for y in range(self.game.cols):
                if (x, y) in processed:  # Skip already processed cells
                    continue
                if self.game.visible_grid[x][y].isdigit():
                    number = int(self.game.visible_grid[x][y])
                    neighbors = self.get_neighbors(x, y)
                    unrevealed = [n for n in neighbors if self.game.visible_grid[n[0]][n[1]] == '*']
                    flagged = [n for n in neighbors if self.game.visible_grid[n[0]][n[1]] == 'F']

                    if len(unrevealed) > 0 and len(flagged) == number:
                        for nx, ny in unrevealed:
                            self.game.reveal_cell(nx, ny)
                            move_made = True
                        processed.add((x, y))  # Mark cell as processed

        # Step 2: Flag mines
        for x in range(self.game.rows):
            for y in range(self.game.cols):
                if (x, y) in processed:  # Skip already processed cells
                    continue
                if self.game.visible_grid[x][y].isdigit():
                    number = int(self.game.visible_grid[x][y])
                    neighbors = self.get_neighbors(x, y)
                    unrevealed = [n for n in neighbors if self.game.visible_grid[n[0]][n[1]] == '*']
                    flagged = [n for n in neighbors if self.game.visible_grid[n[0]][n[1]] == 'F']

                    if len(unrevealed) > 0 and len(unrevealed) + len(flagged) == number:
                        self.flag_cells(unrevealed)
                        move_made = True
                        processed.add((x, y))  # Mark cell as processed

        # Step 3: Apply patterns if no move has been made
        if not move_made:
            move_made = self.apply_patterns(processed)

        # Step 4: Probability-based guess as a last resort
        if not move_made:
            print("Making a probability-based guess.")
            self.probability_guess()
            move_made = True

        # Safeguard: If no progress is possible, stop processing
        if not move_made:
            print("No progress possible. Solver stopping.")
            self.game.game_over = True

        return move_made
```

Fig 3.4 The main code for the Minesweeper Engine class

The Minesweeper engine's logic is implemented through pattern recognition algorithms. These algorithms leverage adjacency rules and logical deduction to identify safe cells or mines. For instance, the **1-1 pattern** uses overlapping neighbor sets to determine specific moves.

```python
def detect_1_1(self, x, y, processed):
    number = int(self.game.visible_grid[x][y])
    neighbors = self.get_neighbors(x, y)
    unrevealed = [n for n in neighbors if self.game.visible_grid[n[0]][n[1]] == '*']
    flagged = [n for n in neighbors if self.game.visible_grid[n[0]][n[1]] == 'F']

    # Skip cells with no unrevealed neighbors
    if not unrevealed or (x, y) in processed:
        return False

    for nx, ny in neighbors:
        if self.game.visible_grid[nx][ny].isdigit():
            adjacent_number = int(self.game.visible_grid[nx][ny])
            adjacent_neighbors = self.get_neighbors(nx, ny)
            adjacent_unrevealed = [n for n in adjacent_neighbors if self.game.visible_grid[n[0]][n[1]] == '*']
            # Compute shared and unique cells
            shared_unrevealed = [n for n in unrevealed if n in adjacent_unrevealed]
            unique_unrevealed = [n for n in adjacent_unrevealed if n not in unrevealed]
            adjacent_flagged = [n for n in adjacent_neighbors if self.game.visible_grid[n[0]][n[1]] == 'F']

            # Skip if no actionable unique cells or all neighbors are processed
            if not shared_unrevealed or (nx, ny) in processed:
                continue

            # Ensure the pattern logic is valid
            if len(shared_unrevealed) == len(unrevealed) and adjacent_number - len(adjacent_flagged) == 1:
                # If no unique cells, mark as processed and skip further checks
                if not unique_unrevealed:
                    processed.add((x, y))
                    processed.add((nx, ny))
                    return False

                # Reveal unique cells
                for ux, uy in unique_unrevealed:
                    self.game.reveal_cell(ux, uy)
                processed.add((x, y))  # Mark as processed
                processed.add((nx, ny))
                return True
    return False
```

Fig 3.5 The code for the 1-1 pattern implementation

Outside of the 1-1 pattern implementation, the 1-2 pattern is also implemented, on the code below:

```python
def detect_2_1(self, x, y, processed):
    number = int(self.game.visible_grid[x][y])
    neighbors = self.get_neighbors(x, y)
    unrevealed = [n for n in neighbors if self.game.visible_grid[n[0]][n[1]] == '*']
    flagged = [n for n in neighbors if self.game.visible_grid[n[0]][n[1]] == 'F']

    # Skip cells with no unrevealed neighbors
    if not unrevealed:
        return False

    if number - len(flagged) == 2:  # "2" clue logic
        for nx, ny in neighbors:
            if self.game.visible_grid[nx][ny].isdigit():
                adjacent_number = int(self.game.visible_grid[nx][ny])
                adjacent_neighbors = self.get_neighbors(nx, ny)
                adjacent_unrevealed = [n for n in adjacent_neighbors if self.game.visible_grid[n[0]][n[1]] == '*']
                shared_unrevealed = [n for n in unrevealed if n in adjacent_unrevealed]
                unique_to_2 = [n for n in unrevealed if n not in shared_unrevealed]
                adjacent_flagged = [n for n in adjacent_neighbors if self.game.visible_grid[n[0]][n[1]] == 'F']

                # Ensure the adjacent "1" logic holds
                if adjacent_number - len(adjacent_flagged) == 1 and len(unique_to_2) == 1:
                    # Flag the unique cell for the "2"
                    for ux, uy in unique_to_2:
                        self.game.flag_cell(ux, uy)
                    processed.add((x, y))  # Mark as processed
                    processed.add((nx, ny))
                    return True
    return False
```

Fig 3.6 The code for the 1-2 pattern implementation

Several key features of the engine include:

1) Safe Reveals and Mine Flagging

   The engine identifies cells that can be safely revealed or flagged as mines based on numerical clues provided by adjacent cells. This involves checking the consistency of flagged cells and unrevealed neighbors against the number displayed in each cell. This safe revealing method is the implementation of the basic patterns.

2) Pattern Recognition

   The algorithm uses patterns such as "1-1" and "1-2" to deduce safe moves or mines. For example, in the "1-1" pattern, if two adjacent cells both indicate "1" and share a single unrevealed neighbor, that neighbor is flagged as a mine. Similarly, the "1-2" pattern uses overlapping neighbor sets to determine mines and safe cells.

3) Hole & Triangle Recognition

   Advanced patterns like the hole and triangle are also implemented within the 1-2 and the 1-1 pattern. By comparing the unrevealed neighbors of adjacent cells and their respective numerical requirements, the engine determines whether specific cells must be flagged or revealed.

4) Automation and Efficiency

   The engine loops through the grid, applying these rules iteratively until the game is solved or no further progress can be made. It stops when either a mine is revealed (loss condition) or all non-mine cells are successfully uncovered (win condition).

IV. Implementation Testing and Result

This section presents the testing process for the Minesweeper engine across three difficulty levels: beginner, intermediate, and expert. The results of each test are displayed and discussed, with further evaluation provided in Part B.

A. Unit Testing

1) Beginner Level Test

The first test was conducted on a beginner-level game featuring a 9x9 grid with 10 mines.

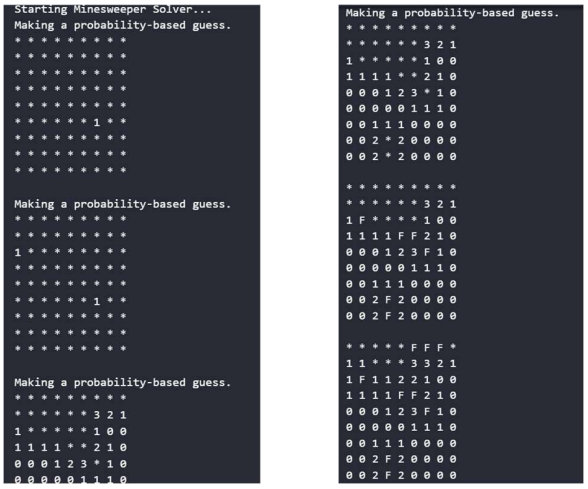

Fig 4.1 The 1st and 2nd stages of the beginner-level test sample

As shown in Figure 4.1, the engine begins with a random guess for the first move, a necessary step given the lack of initial information. Subsequently, the engine systematically flags mines and opens safe cells based on logical deductions, progressing steadily until the sixth move.

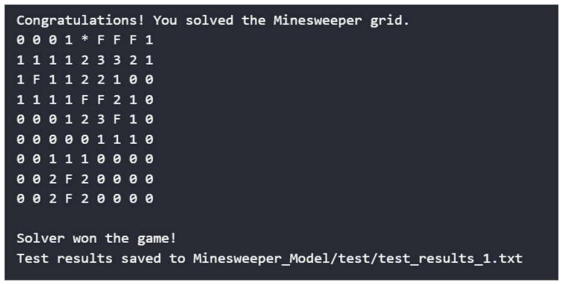

Fig 4.2 Final result of the beginner-level test

By the seventh move, as illustrated in Figure 4.2, the engine successfully completed the game with a win. The results of this test were saved to test_results_1.txt.

```
Minesweeper Engine Test Results
================================
Grid Size: 8x8
Number of Mines: 10
Game Outcome: Win

Final Game State:
0 1 F 1 0 1 F 2
0 1 1 1 0 1 2 F
2 2 2 1 1 0 1 1
F F 2 F 1 0 0 0
3 3 3 2 2 1 0 0
1 F 2 2 F 1 0 0
1 1 2 F 3 2 1 0
0 0 1 1 2 F 1 0

Solution Grid:
0 1 M 1 0 1 M 2
0 1 1 1 0 1 2 M
2 2 2 1 1 0 1 1
M M 2 M 1 0 0 0
3 3 3 2 2 1 0 0
1 M 2 2 M 1 0 0
1 1 2 M 3 2 1 0
0 0 1 1 2 M 1 0
```

Figure 4.3: Saved output of the beginner-level test in the text file

2) Intermediate Level Test

The second test was conducted on an intermediate-level game, using a 16x16 grid with 40 mines.

```
1-1 pattern detected at (7, 7).
1-1 pattern detected at (8, 0).
2-1 pattern detected at (13, 10).
0 0 0 0 1 F 1 0 1 1 1 0 0 0 1 F
0 0 0 0 1 1 1 0 1 F 2 1 0 0 1 1
1 2 2 1 0 0 1 1 2 2 F 2 1 1 0 0
1 F F 1 0 0 1 F 1 1 1 2 F 2 1 0
2 4 3 3 1 1 1 1 1 0 0 2 3 F 1 0
F 2 F 2 F 1 0 0 0 0 0 1 F 2 1 0
1 3 2 3 1 2 1 1 1 1 1 1 1 1 1 1
0 1 F 1 0 1 F 2 2 F 1 0 0 0 1 F
1 2 2 2 1 2 3 * * 2 2 2 3 2 2 1
* * 1 * * * * * * 1 1 F F F 2 1
* * * * * * * * * 3 4 4 4 3 F 1
* * * * * * * * * F F F 1 1 1 1
* * * * * * * * * * 4 2 2 1 1 0
* * * * * * * * * * 2 0 1 F 1 0
* * * * * * * * * F 1 0 2 2 2 0
* * * * * * * * * * 1 0 1 F 1 0
```

Figure 4.4: The engine demonstrating the use of 1-1 and 1-2 patterns during the intermediate-level test

As shown in Figure 4.4, the engine effectively employed advanced patterns such as 1-1 and 1-2 to identify mines and open safe cells. This strategic approach enabled the engine to complete the game successfully.

```
Minesweeper Engine Test Results
================================
Grid Size: 16x16
Number of Mines: 40
Game Outcome: Win

Final Game State:
0 0 0 0 1 F 1 0 1 1 1 0 0 0 1 F
0 0 0 0 1 1 1 0 1 F 2 1 0 0 1 1
1 2 2 1 0 0 1 1 2 2 F 2 1 1 0 0
1 F F 1 0 0 1 F 1 1 1 2 F 2 1 0
2 4 3 3 1 1 1 1 1 0 0 2 3 F 1 0
F 2 F 2 F 1 0 0 0 0 0 1 F 2 1 0
1 3 2 3 1 2 1 1 1 1 1 1 1 1 1 1
0 1 F 1 0 1 F 2 2 F 1 0 0 0 1 F
1 2 2 2 1 2 3 F 3 2 2 2 3 2 2 1
* 1 1 F 2 3 F 4 F 1 1 F F F 2 1
1 1 1 1 2 F F 4 2 3 4 4 4 3 F 1
1 2 1 1 1 3 F 2 2 F F F 1 1 1 1
F 2 F 1 1 2 2 1 2 F 4 2 2 1 1 0
1 2 1 1 1 F 1 0 2 2 2 0 1 F 1 0
0 0 1 1 2 1 1 0 1 F 1 0 2 2 2 0
0 0 1 F 1 0 0 0 1 1 1 0 1 F 1 0
```

Figure 4.5: Saved output of the intermediate-level test in the text file

3) Expert Level Test

The third test was conducted on an expert-level game, featuring a 16x30 grid with 99 mines. Due to the inherent randomness of the first move, several initial attempts resulted in failure. However, on one occasion, the engine demonstrated its capability by successfully completing the game.

```
Minesweeper Engine Test Results
================================
Grid Size: 16x30
Number of Mines: 99
Game Outcome: Win

Final Game State:
0 1 F F F F 2 1 1 0 0 1 1 2 2 2 1 0 0 0 2 F 2 0 0 0 0 0 0 0
1 2 3 4 F 3 3 F 2 0 1 2 F 2 F F 2 1 1 1 4 F 3 1 2 2 1 1 1 1
1 F 1 1 1 1 2 F 2 1 2 F 3 4 4 4 F 1 1 F 4 F 3 1 F F 1 1 F 2
2 2 2 0 0 0 1 1 1 1 F 3 3 F F 3 2 1 1 1 3 F 3 2 2 2 2 2 3 F
1 F 2 1 1 1 1 0 0 1 1 1 2 F 4 4 F 1 0 0 0 1 2 F 1 0 0 1 F 3 2
3 4 F 1 2 F 2 1 1 1 1 1 2 2 2 F 3 2 0 0 0 0 1 1 1 0 0 1 1 2 F
F F 3 2 3 F 2 1 F 1 1 F 1 2 4 F 2 0 0 0 0 0 0 1 1 1 0 0 1 1
F F 3 2 F 2 1 1 1 1 1 1 1 1 F F 4 2 1 1 1 2 1 2 F 1 0 0 0 0
2 2 3 F 4 2 2 1 1 0 0 0 0 2 3 4 F F 2 3 F 4 F 2 1 1 0 0 1 1
1 1 3 F 4 F 2 F 1 0 1 2 3 3 F 2 2 3 F 4 F F 2 1 0 0 0 0 1 F
F 2 3 F 3 2 3 2 1 0 1 F F F 3 2 1 1 2 F 3 2 2 1 2 2 2 1 1 1
1 3 F 4 2 2 F 1 0 0 1 2 4 3 3 F 1 0 2 2 2 1 2 F 2 F F 2 1 0
0 2 F 3 F 2 2 2 1 1 1 2 2 F 2 2 3 2 2 F 1 1 F 2 3 3 4 F 2 1
0 1 1 2 1 1 1 F 2 2 F 2 F 2 2 2 F F 5 4 3 3 3 3 2 F 3 2 3 F
0 0 0 0 0 1 2 3 3 F 2 2 1 2 2 F 5 F F F 2 F F 3 3 3 F 2 1
0 0 0 0 0 1 F 2 * 2 1 0 0 1 F 2 3 F 4 3 2 2 2 3 F 2 F 2 1 0

Solution Grid:
```

Figure 4.6: Saved output of the expert-level test in the text file

B. Win Rate Simulation

To evaluate the engine's overall performance, its win rate was assessed across all three difficulty levels using a simulation. Each difficulty level was simulated 100 times to determine the engine's success rate. The detailed results of these simulations are presented below.

```
Congratulations! You solved the Minesweeper grid.
2 F 2 0 0 0 0 0 0
2 F 2 0 0 0 0 0 0
1 2 2 1 1 2 2 1 0
0 1 F 2 2 F F 1 0
0 1 2 F 3 3 2 1 0
0 0 1 2 F 1 0 1 1
0 0 0 2 2 2 0 1 F
0 0 0 1 F 1 0 2 2
0 0 0 1 1 1 0 1 F

Solver won the game!
Game 100/100: Win
Beginner Level Win Rate: 68.00%
```

Fig 4.7 The engine's win rate of Beginner games are about 68%

```
Making a probability-based guess.
Game Over! You hit a mine.
* * * 2 * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * M * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * 3 * * * *
* * * * * * * * * * * * * * * *
* 3 * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * *

Solver lost the game.
Game 100/100: Loss
Intermediate Level Win Rate: 40.00%
```

Fig 4.8 The engine's win rate of Intermediate games are about 40%

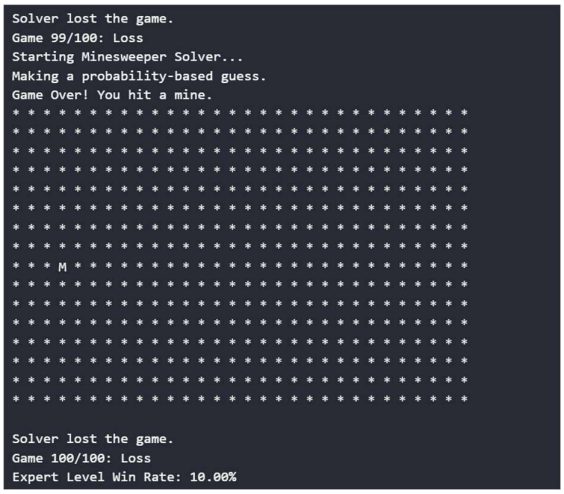

Fig 4.9 The engine's win rate of Expert games are about 10%

The analogy of each engine's win rate will be discussed in the section below.

C. *Result Discussion & Analysis*

The Minesweeper engine's win rates, which stand at approximately 68% for beginner-level games, 40% for intermediate, and 10% for expert, reflect the intrinsic challenges posed by the game's structure and randomization. The disparity in success rates can be attributed to the increasing complexity of logical deductions required and the heightened risk of unavoidable random guesses as the difficulty level rises.

In beginner games, the smaller grid and fewer mines offer a relatively higher probability of making accurate deductions and fewer instances of forced guesses. Intermediate games have a higher density of mines and a larger grid, increasing the likelihood of scenarios requiring guesses due to insufficient clues. Expert-level games further amplify these challenges, often presenting situations where logical deduction is insufficient, forcing the engine to rely heavily on probabilistic guessing.

The random distribution of mines inherently limits the engine's ability to find a guaranteed solution, as the game's progression is influenced by the availability and clarity of numerical hints. This underscores the probabilistic nature of Minesweeper, where even an advanced logical engine cannot always overcome the uncertainties of initial moves and late-game decision-making.

Interestingly, while the engine's win rates may initially appear modest, a comparison with the author's personal Minesweeper statistics offers a compelling perspective on its performance.

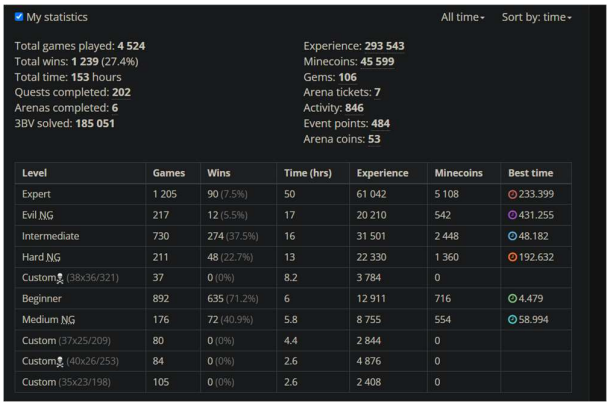

Fig 4.10 The author's personal Minesweeper win rate

As an experienced Minesweeper player, the author's win rates are reflective of substantial skill, particularly on advanced variations like Evil NG and Hard NG. However, for consistency in analysis, this comparison focuses on the Beginner, Intermediate, and Expert difficulty levels. On beginner games, the author's win rate of 71.2% narrowly exceeds the engine's 68%, demonstrating a slight 3% advantage. Conversely, in intermediate games, the engine achieves a superior win rate of 40% compared to the author's 37%, indicating its capacity to handle increased complexity more effectively. In expert games, the engine also outperforms with a win rate of 10%, surpassing the author's 7.5%.

These results underscore the engine's capability as a Minesweeper solver, exceeding expectations and proving competitive even against an experienced human player. The engine's success can be attributed to its consistent application of logical deductions and its immunity to human limitations, such as cognitive fatigue or biases in probabilistic decision-making. Moreover, when evaluated against global averages—49.8% for beginner, 24.1% for intermediate, and 2.8% for expert—the engine's performance is particularly noteworthy, significantly outperforming most players at all levels. This demonstrates the robustness of the implemented algorithms and highlights their potential as an effective Minesweeper-solving tool capable of rivaling both casual and skilled human players.

V. CONCLUSION

The development of a Minesweeper engine utilizing graph and logical implication-based patterns has proven to be not only feasible but highly effective, exceeding initial expectations. The engine employs a recursive approach, systematically analyzing each cell to determine optimal moves through a progression of techniques. Starting with basic pattern recognition and flagging, it advances to more sophisticated strategies such as 1-1 and 1-2 reduction patterns. This systematic methodology has resulted in win rates of 68%, 40%, and 10% for the Beginner, Intermediate, and Expert difficulty levels, respectively.

While the development process faced challenges—including issues like infinite loops and the complexity of implementing advanced algorithms such as endgame dependency chains—these obstacles were addressed effectively. The results demonstrate that the integration of graph theory through the neighboring system and logical implication in pattern

recognition can significantly enhance the engine's ability to solve Minesweeper puzzles. This project underscores the potential of computational logic in tackling probabilistic games and lays the groundwork for further improvements in algorithm design.

VI. Appendix

For those interested in exploring the implementation details or conducting further experiments, the full codebase is available on GitHub at:
https://github.com/haegenpro/Minesweeper_Model.

VII. Acknowledgment

The author would like to express the deepest gratitude to the Lord Almighty for His guidance, wisdom, and blessings throughout the development of "Graph and Logical Implication-Based Patterns in the Design of a Minesweeper Engine". It is through His grace that challenges were overcome, clarity was achieved in solving complex problems, and the work was successfully completed. His presence has been a constant source of strength and inspiration, enabling perseverance during the most demanding phases of this endeavor. For this, the author is profoundly thankful.

The author would also like to extend heartfelt gratitude to all those who contributed to the preparation of this paper, including:

1. Dr. Rinaldi Munir, M.T., lecturer of the K1 IF1220 Discrete Mathematics course, for his invaluable guidance and the knowledge imparted during the lectures,
2. The author's parents, for their unwavering support, encouragement, and motivation throughout this journey, and
3. The author's friends and peers, for their insightful feedback and encouragement during the drafting process of this paper.

Their contributions and support have been instrumental in the successful completion of this work.

References

[1] How-To Geek. "30 Years of Minesweeper: Sudoku with Explosions." How-To Geek, https://www.howtogeek.com/693898/30-years-of-minesweeper-sudoku-with-explosions . Accessed 20 Nov. 2024.
[2] Minesweeper Strategy. "Advanced Patterns." Minesweeper Strategy, https://minesweepergame.com/strategy/patterns.php . Accessed 20 Nov. 2024.
[3] Trudeau, Richard J. 1994. *Introduction to Graph Theory.* Dover Publications.
[4] Minesweeper Online. "Patterns." Minesweeper Online, https://minesweeper.online/help/patterns. Accessed 25 Dec 2024.
[5] Munir, Rinaldi. 2024. "Logic". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/01-Logika-2024.pdf . Accessed 20 Nov 2024.
[6] Munir, Rinaldi. 2024. "Graf (Bag.1)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf . Accessed 20 Nov 2024.
[7] Munir, Rinaldi. 2010. "Matematika Diskrit Edisi 3". Penerbit Informatika Bandung.

Statement of Originality

I hereby declare that this paper I have written is my own work, not an adaptation or translation of someone else's paper, and not plagiarism.

Bandung, 27th December 2024

Haegen Quinston
13523109