Game of Minesweeper with Quantum Safety Mechanism

Kaan Tekin **220011642**

Supervisor: Michael Young

School of Computer Science

University of St Andrews

This thesis is submitted in partial fulfillment for the degree of
MSc Information Technology with Management
at the University of St Andrews, School of Computer Science

August 2023

Abstract

This dissertation introduces a safety mechanism to the game of Minesweeper to save the player from losing by moving the mine away in the case of having to guess a move when no logical inference can be made. The name “Quantum Safety” leverages the principles in superposition from quantum mechanics. Superposition is a fundamental concept in quantum mechanics that refers to the ability of quantum systems to exist in multiple states simultaneously until being observed. Under the context of Minesweeper, the mine’s location remains in an indeterminate state until it is uncovered by the player. With the mechanism active throughout the game, the gaming experience is significantly improved due to the elimination of luck which renders it to be solely based on logical deduction. An artificial intelligence arbiter for the mines’ deducibility is devised to play the game alongside the player. To further enhance the gameplay experience, instructions are provided after a lost game to explain how the loss could have been prevented. The addition of the functionalities is an enhancement that aims to elevate the general gaming experience of Minesweeper.

Acknowledgements

First of all, I would like to express my gratitude towards my project supervisor Michael Young for his consistent support, understanding and insightful feedback week after week throughout the entire dissertation period. Working with him and under his guidance has been efficient, effective and just generally pleasant.

I would like to show love to my loved ones, my family and my partner, for their emotional support and motivation throughout this year of my postgraduate studies. My love for them is beyond what words could describe, they have made this year much easier for me.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is NN,NNN* words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

If there is a strong case for the protection of confidential data, the parts of the declaration giving permission for its use and publication may be omitted by prior permission of the Director of Postgraduate Teaching or Honours coordinator (as appropriate).

14 August 2023 Kaan Tekin

Table of Contents

Table of Figures

1 Introduction

This section introduces the main aims and objectives (Section 1.1) of this dissertation, as well as the final achievements (Section 1.2) to provide the reader with a general idea of what the work is about.

1.1 Aims and Objectives

Minesweeper is a strategic game that involves a lot of logical deductions, however oftentimes the element of luck might become a determining factor of the outcome of a game, when a player has to guess which cell to probe. This might ruin the gameplay experience because the previous effort could be then in vain.

Thereby, the **primary objectives** are:

1. Re-create a user-playable game of Minesweeper with basic functionalities such as probing, flagging and restarting.

2. Devise a quantum safety mechanism that algorithmically moves the un-deducible mine away and re-arrange the back end game board when the user decides to take a guess and clicks on the said mine. This mechanism is active throughout the game to protect the user every step of the way, which eliminates "luck" out of the equation.

3. An AI agent that plays the game alongside the user to determine whether each clicked mine is deducible.

The **secondary objective** of this dissertation is:

1. Create a post-game instructor mechanism that is triggered when a user actually logically loses the game. The instructor would review the current game and instruct the player how he/she could have deduced that the mis-clicked mine is indeed a mine. It explains to the user why it should not have been a lucky guess.

To sum up, the project aims to improve the gameplay experience of Minesweeper by eliminating the element of luck and by educating the player how the loss could have been prevented.

1.2 Achievements

This dissertation has constructed a playable game of Minesweeper with basic functionalities, implemented an AI agent that is able to play a full game of Minesweeper as well as to play with the user to continuously identify an un-deducible mine, and has devised a successfully working quantum safety mechanism that moves an un-deducible mine away when a player clicks on it. Furthermore, a step-by-step guide for how to prevent a loss has been created to assist and improve a player's capability after each lost game.

The work has provided two additional functionalities for the game of Minesweeper to enhance the general gaming experience. To the best extent of the author's knowledge, no prior work in such direction has been done, and all objectives set have been successfully accomplished.

1.3 Context Survey

In this chapter, background introduction on the history of development of Minesweeper as well as scientific research on the game are outlined.

1.3.1 Background of Minesweeper

The following sections provide a brief explanation of the origin history of Minesweeper (Section 1.3.1.1), the gameplay strategy (Section 1.3.1.2) and the first click safety mechanism (Section 1.3.1.3).

1.3.1.1 Origin of Minesweeper

Minesweeper was first released by Microsoft on October 8th, 1990 as part

of the Windows Entertainment Pack. The game quickly became popular due to its intuitive and addictive strategic gameplay. It has attracted millions of players since its commercial debut. Originally, the game was called Mine and was developed by two Microsoft employees, Curt Johnson and Robert Donner. It was fortunately selected to be part of the Entertainment Pack for Windows (Figure 1.1), which was set to be a marketing tactic to encourage purchase of Windows after Windows 3.0 came out. The game has been included in every version of Windows up until Windows 8, after which it became widely available on other operating systems as well.

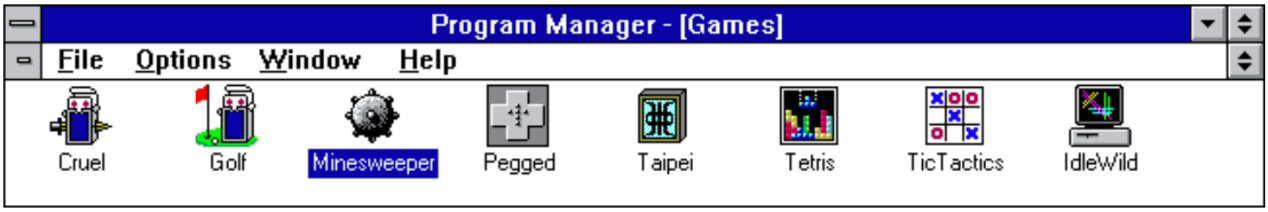

Figure 1.1: Original Entertainment Pack for Windows

1.3.1.2 Gameplay Strategy

The game's objective is to clear a minefield without detonating any mines. The game board can have different sizes and it is a grid of n x m square cells. Some cells are safe but some have mines hiding under them, the user will detonate the mine if the dangerous cell is probed (clicked on in this context), which leads to losing the game of Minesweeper (Figure 1.2). All the cells are initially unprobed (covered). After the player probes a cell, if there is a mine underneath, the player loses. Otherwise, the cell will display a number on the surface between 1-8, which indicates the number of mines among all its adjacent neighbor cells (Figure 1.3). If there is no mine nearby, the square will display nothing and automatically reveal all other adjacent cells with the same situation until it reaches the ones with numbers (Figure 1.4). Players must use logic and deduction to determine the locations of mines based on the numbers displayed on the

game board. Correctly opening all safe cells is the way the only win the game of Minesweeper (Figure 1.5).

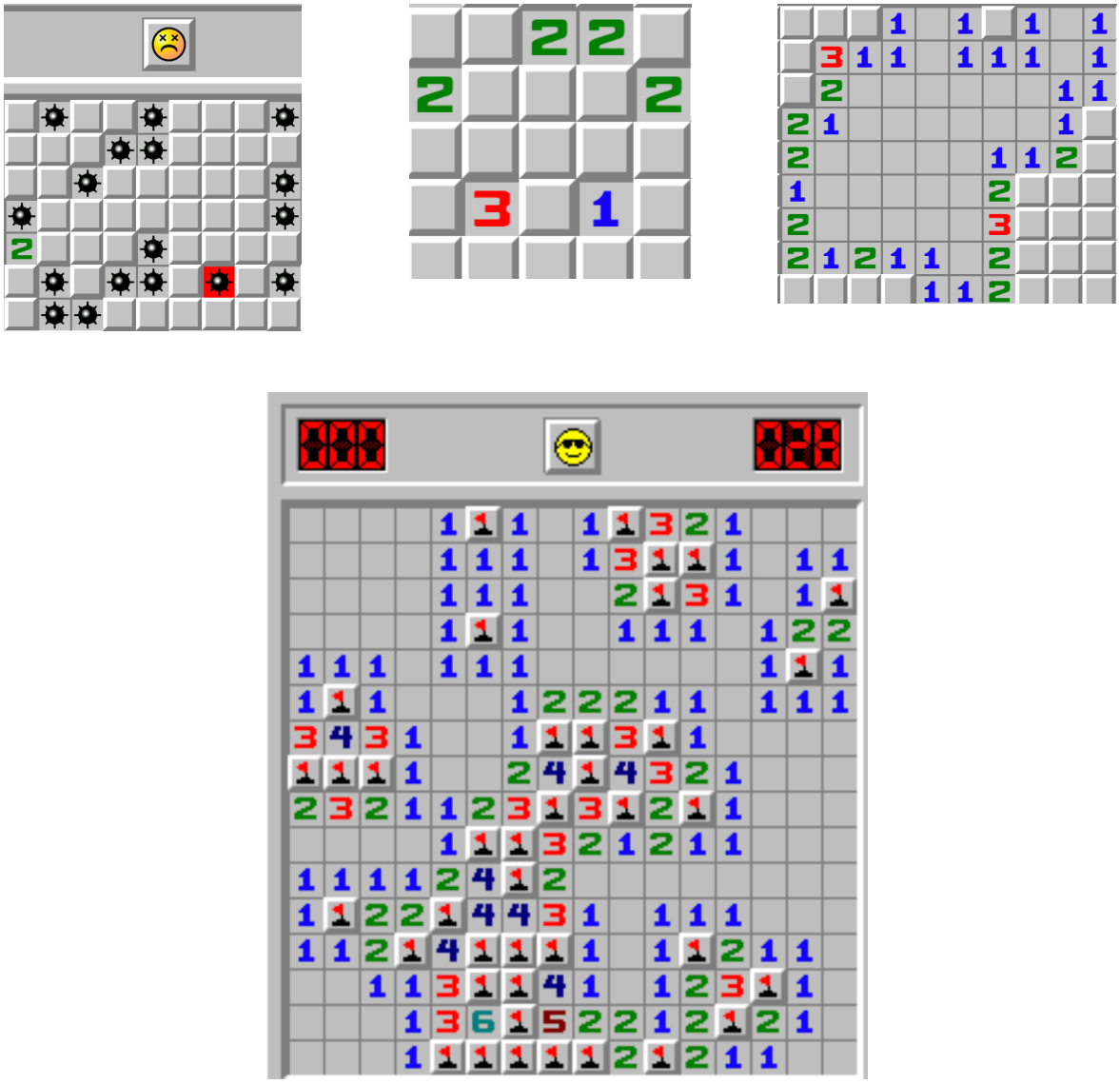

Figure 1.2 &1.3 & 1.4 & 1.5: Demonstration of Minesweeper game rules

To make the game more interesting, Windows also implemented other features such as first click safety (where the first probe is always safe), chording (pressing both buttons on the mouse to save time on revealing definitely safe squares) and flagging (marking a square as mine). However, for the sake of this project, only first click safety mechanism will be included as it aligns with the focal aim of the research.

1.3.1.3 First Click Safety Mechanism

To ensure that the player does not lose on the first move, Microsoft implemented a first click safety mechanism to the game. This is of significance because it shares similar idea as the “quantum safety” in this project. There are two ways that have been commonly implemented to ensure safety on the first go (Damien, 2021):

1. Generating the game board after the player’s first move: In this approach, the game doesn't actually place the mines on the board until after the player makes their first move. Once the player has probed a cell, the game places the mines in the remaining cells, ensuring that the first clicked cell is always safe. This is simpler and is adopted by many modern Minesweeper games.

2. Generate as normal but shift the mine if first click is a mine: This is the original algorithm Microsoft adopted. If the player’s first click is on a mine, the mine is automatically shifted either to the upper-left corner of the board or to the nearest empty cell.

The second approach is already a primal version of quantum safety because moving the mine elsewhere requires dynamic change of the board itself. Upper-left corner is the default option because it causes the least impact on the board. However, this mechanism can be extended to span throughout the whole game, which requires constant dynamic shifting of the board to ensure a better gaming experience. Hence, the original algorithm serves as a helpful reference for the purpose of this research.

1.3.2 Related Work

Very few studies have been done on altering the built-in mechanisms of Minesweeper to improve users’ gaming experience, most academic and scholarly research related to Minesweeper tends to focus on AI algorithmic strategies for solving the game, solvability of the game, and probabilistic

reasoning. This section will sum up some of the findings and achievements in those research.

Chang (2022) analyzed the solvability of the game of Minesweeper after categorizing the game to be most of the time a problem of single-agent stochastic puzzle. It distinguished the definition of a “game” from a “puzzle” based on the winning rate of the proposed PAFG AI algorithm (Figure 1.6). The achieved result of 82% winning rate by the algorithm on a 9x9 board indicates that 82% of the time Minesweeper is stochastic and solvable, which is considered a “puzzle”, while the other 18% makes it a game that is unsolvable and could purely rely on luck. Chang (2022) acknowledged that guessing is an inevitable component during Minesweeper, and the previous result was achieved by the more advanced PAFG (“G” for Guessing Strategy using probability) solver rather than PAFR (“R” for Random Guessing).

$$p = \begin{cases} 1, & \text{deterministic puzzle} \\ 0 < p < 1, & \text{stochastic puzzle} \\ 0, & \text{game} \end{cases}$$

Figure 1.6: Difference between “puzzle” and “game”, where p represents possibility of finding a solution with single-agent AI (i.e. winning rate)

The early stage of AI attempts to solve Minesweeper only focus on making deterministic moves. Adamatzky (1997) modeled a cellular automation that defines a set of cell states and a transition function to analyze the board situation and make further moves. However, the limitation on the transition function causes the agent to be stuck in situations where guesses need to be made.

To resolve this issue, non-deterministic moves are included in later algorithms. Becerra (2015) analyzed and compared different standard algorithms using single point strategies with stochastic moves in Minesweeper.

The single point strategy focuses on one square at a time when the agent iterates through the cells that decides whether the cell falls under the condition (Figure 1.7) of AMN (All Marked Neighbors) or AFN (All Free Neighbors). In the circumstances where the agent is stuck, the author tested out two alternatives on making non-deterministic moves: random guesses and heuristics-based guesses. Becerra mathematically proved that corner squares and edge squares have the lowest probabilities of being a mine, which renders them the optimal choice for non-deterministic moves. As a result, Becerra concluded that the single point strategy suffers from short-sightedness as it fails to utilize useful information that could be obtained by other squares altogether. However, incorporating known information on the current board into guesses does lead to better performance than random uninformed guesses.

Figure 1.7: Instance of All Free Neighbors (Left) vs Instance of All Marked Neighbors (Right).

Chris Studholme (2000) approached Minesweeper as a constraint satisfaction problem (CSP), where the agent decides the next move based on the aggregate constraint that reflects the knowledge base of the current board. The number on a square limits the possible locations of mines in its neighbors. Once given enough constraints, the exact locations can be logically deduced. This situation is called finding the set of values (mine locations) that satisfies the constraints. This approach had also been included for comparison in Becerra's work, that demonstrates higher winning rates than the single point strategy.

Nayotama (2021) further improved the CSP approach by introducing backtracking algorithm. It replaces the original exhaustive search by only

exploring promising options. Nayotama implemented the Depth First Search backtracking algorithm (Figure 1.8) in a recursive manner, where the agent visits and assigns an unprobed cell to be mine/safe, then it goes on to explore the consequences of this assignment. If at any point of the traversal shows the inability to satisfy he constraint, the algorithm backtracks to make an alternative assignment. Otherwise, the constraints reflecting the knowledge of the board recursively get updated. The algorithm stops when it has exhausted all possible assignments.

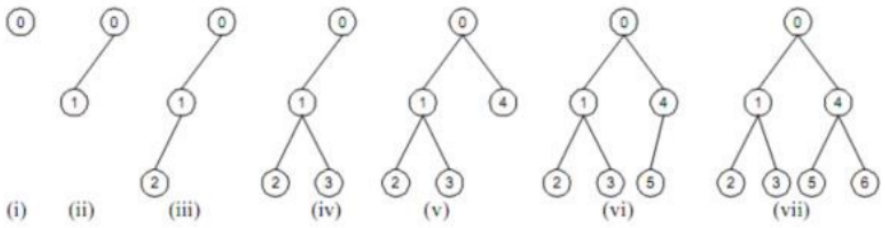

Figure 1.8: State space tree with Nayotama's DFS algorithm

These studies revolve around the vital element of guessing in the game of Minesweeper and concentrate on overcoming the negative impact of it on an AI agent's winning rate. It is evident that having to make uncertain choices significantly limits the winning experience of an AI agent, let alone human player, which is why completely wiping out the factor of luck by introducing the quantum safety mechanism is an avant-garde approach to enhance the gameplay experience for a user.

2 Project Design

This section explains the design planning (Section 2.1) before the implementation as well as showcases the general system structure of how the components are arranged (Section 2.2).

2.1 Component Planning

To ensure that the eventual artifact works as desired, the following components are fundamental to this project:

· **Game Maps/Boards:** Logically correct Minesweeper maps. Fixed boards for testing and a random board generator.

· **Graphical User Interface (GUI):** A working GUI with basic functionalities that replicates the original Minesweeper interface.

· **Artificial Intelligence Agent:** An AI arbiter for the deducibility of mines that is able to play a full game of Minesweeper on its own, as well as alongside the user, receiving his/her actions on the GUI.

· **Quantum Safety Algorithm:** A correct algorithm that detects and moves the mis-clicked un-deducible mines away unbeknownst to the user.

· **Post-game Instructor:** An algorithm that collects the steps taken until the user's losing click and demonstrates to the user.

2.2 System Architecture

The system architecture is shown below (Figure 2.1). Under /src, the Map class stores 10 fixed Minesweeper boards for testing and one method for generating random boards. The Main method reads the user command and retrieves the according game board from Map, then initiates an AI agent with said board. The Agent uses two libraries under /libs for logical deductions. At the instant of agent instantiation, a GUI delegate is initiated. A two-way

communication channel is built between the Agent and the GUI, which will be discussed in later sections.

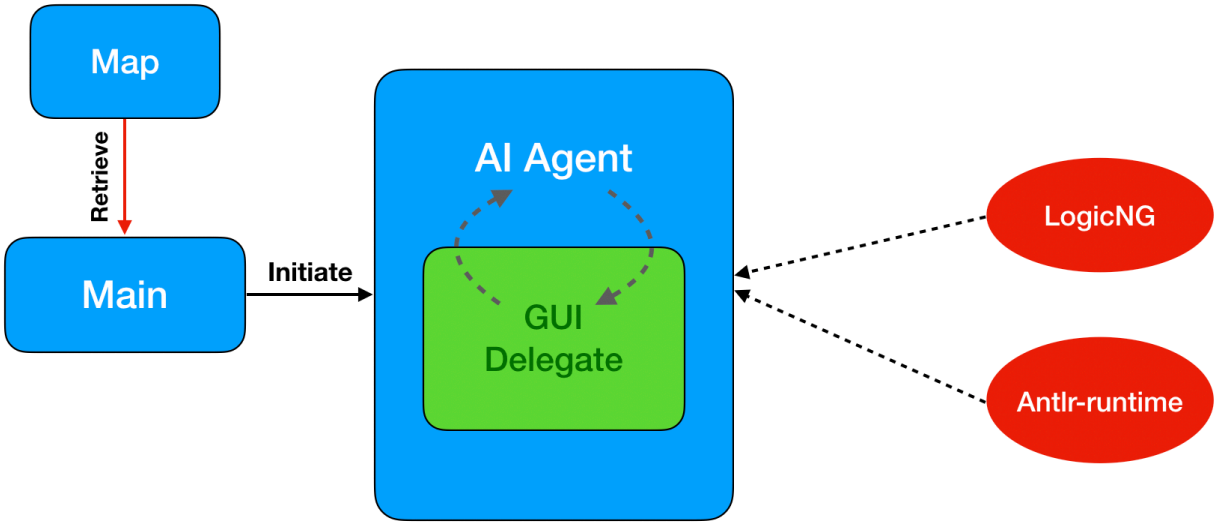

Figure 2.1: System Architecture of the work

3 Elements of Implementation

This section goes into details of how the required components are realized and implemented. It provides a clear outline and understanding of the construction of the entire system.

3.1 Minesweeper Artificial Intelligence Agent

The AI agent is designed to use the satisfiability test reasoning strategy (SATS) to play through Minesweeper, where it transforms its current partial view of the game board into a logic sentence, and uses satisfiability results to decide whether to probe, flag or skip a cell. The agent maintains its own undiscovered board on which it operates and retrieves values from the canonical board. In practice, the agent loops through each cell for appropriate actions and breaks when it reaches an impasse.

Agent's Dynamic Iteration: To enable gameplay with a dynamically changing map (its own map, because the agent as well as the player continuously makes progress), the agent encounters a new map view during each iteration, requiring the agent to reassess previously unresolved cells for potential probing. To achieve this, a Linked List is an ideal choice. The agent sets up a "frontier" linked list to store all the cells. As long as the frontier is not empty, the algorithms select a cell from the list to determine if it can be probed or marked. If there are no actionable steps, the cell is reinserted into the list, ensuring it reappears for consideration once further exploration of the map takes place. The frontier is also renewed when the progress is regressed during QSM, which is discussed in later sections.

Encoding knowledge base and query: The agent constructs the knowledge base from only outer cells (cells with covered neighbors, concept explained further in later sections), as further moves only use them as clues for deduction. The literal used to denote a mine being in position (i,j) is Dij, and

~Dij for the opposite. To encode the knowledge base and query for SAT algorithms, the agent follows a series of steps. At each iteration, it constructs a logic sentence representing the knowledge base of the current map view. To achieve this, the agent calls the `makeKB()` function and includes "&~Dij" (not Dij) and "&Dij" (Dij) in the logic sentence. This constructed logic sentence is then provided to the SAT solver as the query. The boolean results obtained from the SAT solver are sent back to a `probe()` function and stored in either `ifProbe` or `ifDanger` variables, which determine the subsequent actions.

The `makeKB()` function is responsible for generating the logical sentence. It first calls the `makeDNFClause()` function for each uncovered non-zero cell. The `makeDNFClause()` function produces the logical sentence in Disjunctive Normal Form for each cell. These individual clauses are then combined to form one final sentence representing the general knowledge base (DNF clauses are encoded by simply creating combinations of literals using appropriate operators).

Results of an Agent's Gameplay: The agent plays through the current version of the map as far as it can. At the end of the iteration, it either gets stuck where no further inference can be made, or completes and wins the entire game after discovering all the cells (Figure 3.1).

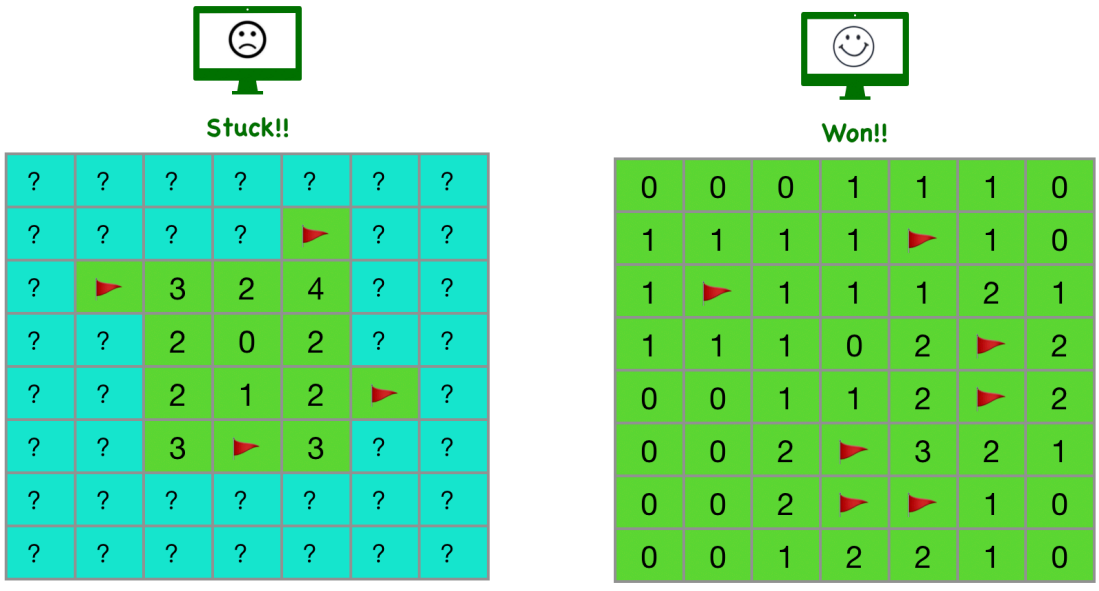

Figure 3.1: Demonstration of the agent gameplay results

3.2 How to identify an un-deducible cell?

The main characteristic of an un-deducible cell is that there is no way to logically step-by-step figure out whether this cell is a mine or not. Since the agent's task is play alongside with the user and deduce as far as it can get, an un-deducible cell would just be an undiscovered (covered) cell after the agent's operation. If the agent is able to finish the entire game, there will be no un-deducible cells because every single cell can be logically inferred and figured out by expanding from the "current" view of the user's board. Similarly, the agent would only make the guaranteed moves using the user's board as the basis and explore to the furthest. Therefore, when completing the game is impossible, all the unreachable cells by the agent would then fit the definition of un-deducible cells for the user (Figure 3.2). The agent is always ahead of the user and hence will be the arbiter of un-deducible cells.

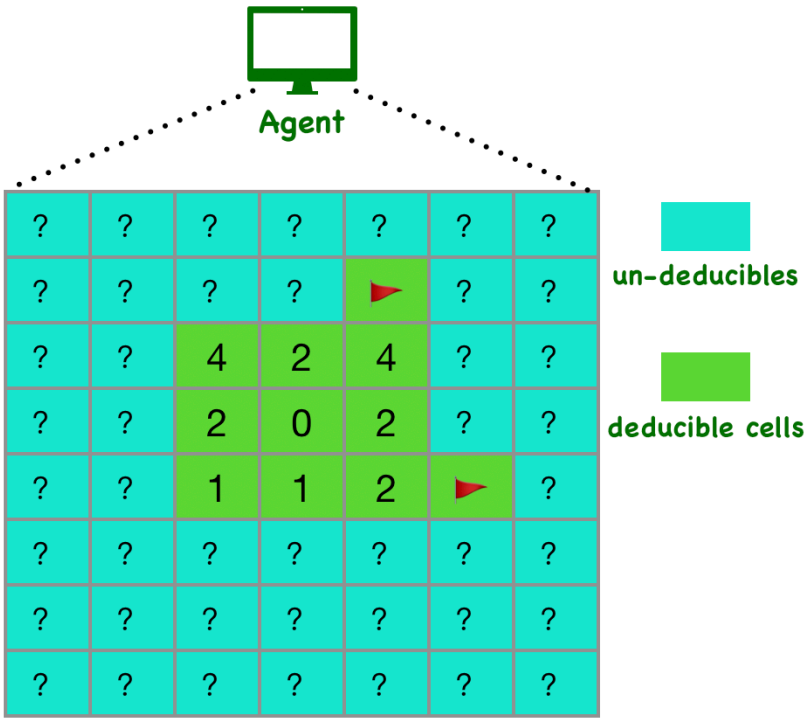

Figure 3.2: Un-deducible vs Deducible cells

In practice, a list will be in place to store all the un-deducible cells. Typically, if the agent gets stuck at some point and couldn't finish the game, a method is called to collect all uncovered cells from the agent's board to the list.

However, one crucial thing to note is that this list is susceptible to continuous updates as the user and agent progresses during a game. A cell might not be deducible at one point, but with a few more clues given after several steps, it could become deducible.

Since the agent tends to always be ahead of the user's progress, the only times the agent need to update the un-deducible list is when the player probes a cell that the agent hasn't already uncovered yet, as it provides more information and clue about the game board. When this happens, the agent continues from its previous progress to explore the map. Every time the agent got stuck in the game, the un-deducible list first clears itself completely and re-stores all uncovered cells at that moment. The list is also cleared when the agent wins the game. The operation workflow is summarized and demonstrated below:

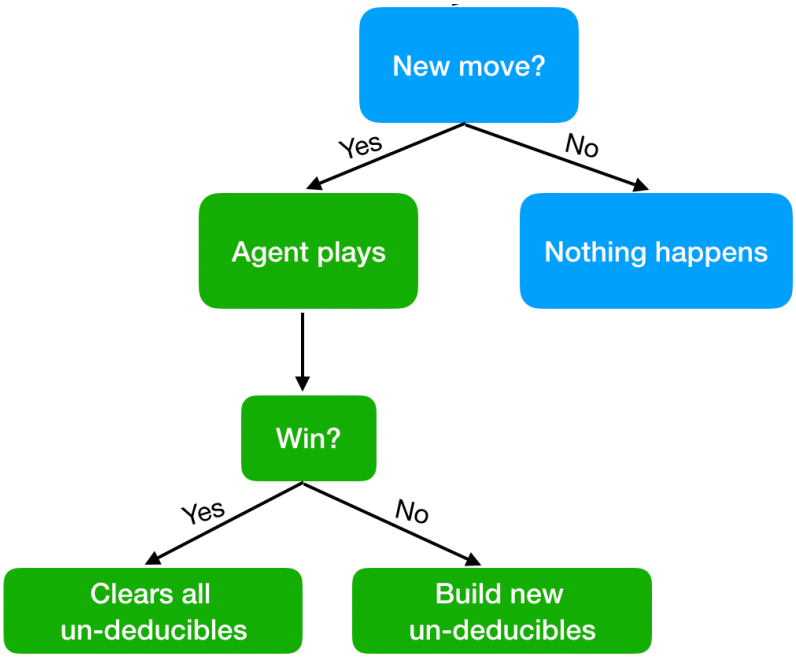

Figure 3.3: Operation workflow of building un-deducibles

3.3 Communication between Agent and GUI Delegate

A two-way communication must be established between the agent and the GUI for the project to work. This next section will explain the reason and implementation for the communication of each direction.

Agent to GUI: Although the agent tends to play at the backend alongside the user, there is one crucial scenario where the agent would have impact on the GUI delegate. The system is constructed following a Model-Delegate pattern where the GUI has no knowledge about a model that it reacts to accordingly when changes take place, and vice versa. The model in this case is the actual game board. Since the GUI delegate renders and displays numbers/mines according to the "board" model when a user clicks, the only scenario where changes can take place is precisely when the board is altered as a result of quantum safety mechanism. The agent is the entity that decides whether to activate quantum safety mechanism or not because the critical list of un-deducibles is stored and renewed in Agent. Once the agent identifies a probed cell to be un-deducible, it activates the mechanism and changes the board model, which GUI delegate would be notified of the change and receive the new model to display to the user for future steps.

To achieve this, the agent has the actual canonical gameBoard as a private attribute which it has authority over for altering it when needed. The GUI delegate is also conveniently set to be a private attribute inside agent which will be instantiated when an agent is created. The agent passes the canonical gameBoard as a parameter down to the GUI delegate, which it takes to be its private attribute in order to retrieve values from and display them to the user. Under this design, all the decisions and actualization of changes will happen at the back end unbeknownst to the user because the already-uncovered part is avoided while reconstructing the board, and only GUI delegate experiences the impact on the rest of the board, reacts swiftly and discreetly.

GUI to Agent: The communication channel in this direction is vital as the agent needs to know each step the user takes in order to play along. The GUI is responsible for identifying if the user clicked on a mine or a safe cell, then inform the agent of said click to trigger different methods accordingly.

In the scenario where a safe cell is clicked, the agent receives the coordinate of the cell and discern if it's a new move that could progress the agent's game further. If yes, the agent probes the cell itself and update the list of un-deducible cells. If no, simply update the version of its knowledge of the current view from the user's perspective (reasons explained in later section).

In the scenario where a mine is clicked, the agent receives the cell and decides if it's un-deducible. If yes, re-construct the map for the GUI. If no, inform the GUI to let the user lose the game.

An extra scenario that is enabled by the "reset" functionality provided in the GUI also needs to inform the agent, as the agent is the only one capable of making changes to the canonical gameBoard.

Regardless, the agent needs to be informed of each and every step of the user's gameplay. To establish this channel of communication, the observer pattern is the most suitable candidate. By creating an Observer interface and letting Agent implement it, the GUI delegate can then add the agent as an observer so that it can trigger the correct methods in Agent. The notifier methods in GUI delegate ought to match the listener methods in Agent in cases of mine clicked, cell clicked and reset clicked.

Reaching this step, I have successfully established an effective two-way communication channel (Figure 3.4) between the two entities as a groundwork for this project.

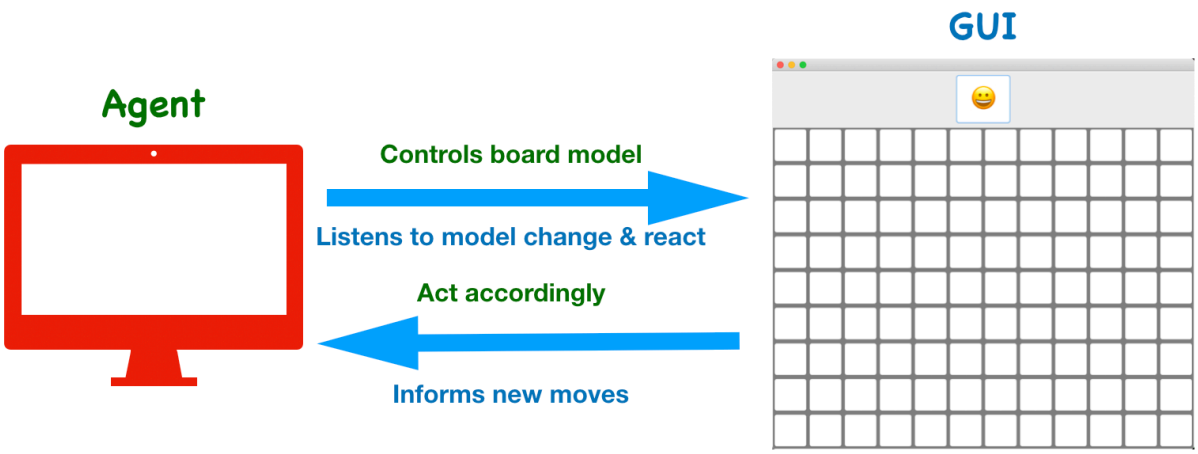

Figure 3.4: Two-way communication channel between Agent & GUI

3.4 Quantum Safety Algorithm

The quantum safety algorithm is one that moves the mine away to a vacant cell when an uneducable mine has been clicked by the user to save the user from losing the game. The design and implementation process for this core algorithm has progressed through **three** stages for refinement. This section will first lead by answering core concerns for such algorithm which would aid the potential audience for this report to segue into the actual demonstration of each stage of the progression as well as the explanation of the principles and details in design.

Q1. What is the major concern for this algorithm?

The general rule of thumb is to make sure that everything is happening at the backend unbeknownst to the user. The general purpose of introducing quantum safety mechanism to the game of Minesweeper is to improve the gameplay experience for the player. Therefore, with this in mind, it stands axiomatic that no matter how much the board is changing, the user should always be oblivious of it, which means the board from the user's perspective (GUI board) ought to never be contaminated by operations at the back. For the rest of this paper, the principle of avoiding contaminating the user's board would be referred to as the **Non Contamination Principle** (NCP).

Q2. Where to move the mine?

The classic Microsoft Minesweeper had kept the implementation of quantum safety on first click the same throughout generations. The idea being that the first row is always kept more vacant than the rest and hence could be used as a go-to 'asylum' for the moved mine. This doesn't require much computation as it's a one-off operation. Also, since most users start the game at cells in the middle, maneuvering mines in the top corner wouldn't contaminate much of the board.

This implementation would not be applicable for the task at hand however, because the mechanism would be active during the entire gameplay and could

be triggered at every step. First problem with this is that, soon the game would run out of space on the the top row and would have to pursue vacancies from rows that following, potentially ending up with a map that has most of the cells on the top and few at the bottom (which worsens the user experience). The real problem with this is the unnecessary waste of computation. Moving the mine to a corner far away does not solve any core problem because the NCP still has be dealt with. The algorithm still has to find a way to plant mines without changing the already uncovered cells by the user. But now in addition to that, the algorithm needs to satisfy the partial board around the affected corner too. Unless the algorithm somehow intelligently finds the 'optimal' cell in the corner, which itself might incur even more computation.

Therefore, since there is only one concern that matters, the best practice is to directly deal with the NCP and start with maneuvering the mines within the proximity of the user-uncovered areas.

3.4.1 General Mine-Moving Algorithm

Step 1. Regress agent's progress: Note that the AI agent and the user have two separate views of the board. The AI plays on its own board and is always ahead of the player's progress, this makes the agent's view and knowledge of the current game wider than the user because it has probed more cells. However, to strictly follow the NCP, the agent needs to bring itself back to the user's level, not only because it could correctly move the mines, but also that it would have to re-play the game because the previous version of game is already altered and abandoned. In practice, this is realized by replacing the agent's list of uncovered cells with the user uncovered cells.

Step 2. Find and collect all the "Outer Cells": An outer cell is one that still has covered neighbors, where an inner cell is one that all its neighbors are already probed (Figure 3.5). Consider the Non Contamination Principle, the agent should always avoid changing any uncovered cells' numbers when re-

arranging mines. This means that all inner cells are out of question as they are immutable and only outer cells requires extra attention. Therefore, the second step is to go through the board and collect the coordinates of all cells that are uncovered yet still have covered neighbors.

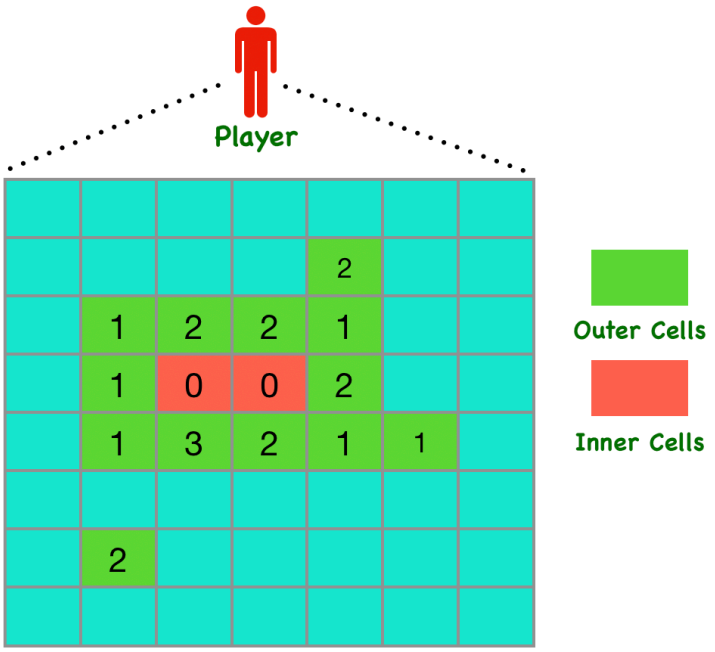

Figure 3.5: Definition of Outer cells vs Inner cells

Step 3. Re-arrange mines around the outer cells: This is the most crucial part of the algorithm. The core idea is to plant mines around each outer cell based on its numeric value, while avoiding planting on the mis-clicked mine (the one that triggered the quantum safety mechanism). The only thing here to mind is to not contaminate other uncovered cells while satisfying the outer cell at hand. This is made sure by checking if adding one more mine exceeds any of the neighbor outer cell's numeric value.

Step 4. Re-construct the rest of the board: Once the area of significance has been taken care of, the rest of the board can simply be reconstructed however one likes as it does not affect the user's gameplay experience. The implementation is direct, re-initialize the rest of board and randomly plant the rest of the mines on the board assign numbers to the remaining cells based on the number of neighbor mines.

Step 4 further discussion: When learning about the last step, one could potentially question the computation of re-creating the entire map and be motivated to seek for a solution that minimizes the area of effect, i.e. only alter the cells that are necessary and keep the rest the same. Before delving into this notion, a visualization of the potential area of effect when moving a mine would be helpful. Observe the two boards in Figure 3.6, the yellow part represents the originally visible uncovered cells from the user. By moving away the mine on the top right corner, the operation ends up altering 17 cells, with the area of effect highlighted in color orange. At first sight, the effect does look minimized, however, the actual effect caused by moving a mine is uncertain and difficult to define depending on the miscellaneous circumstances. In practice, it would require another helper algorithm to first discover and define the area of effect in a given case, and then operate within it. Such algorithm however could inversely incur more computation. Re-constructing the entire board could be processed in the matter of milliseconds and be completely unnoticeable from a player's perspective. Both the space and time complexity is O(rows*columns), which is realistically insignificant for computers because there would never be a board with even 1000 rows or columns as it is meant to be played by humans. In short conclusion, pursuing an intuitively more efficient algorithm is not only unnecessary for its trivial improvement, but could induce more computation for suitable-for-human-sized boards.

Step 5. Renew agent's board and gaming progress: Until Step 4, the re-mapping of the game board has been completed. However, this last step holds immense significance because the agent has to now play on an essentially new board in order for correctly activating the quantum safety mechanism in the future. By assigning all the covered cells from the user's perspective back to unknown and adding them back to the frontier, these cells will face a new round of logical deduction by the agent based on the new board. Last but not least,

remove all the current un-deducibles. A new list might be created if needed in this new agent gameplay.

To conclude, this 5-step process forms the general basis of the core mine-moving algorithm for Quantum Safety Mechanism. The pseudo-code is provided in Appendix B.

1	1	1	1	1	2	t
t	1	1	t	2	3	t
2	1	1	1	2	t	2
1	0	0	0	1	1	1
0	1	1	1	0	1	1
0	1	t	2	1	2	t
0	1	1	2	t	3	2
0	0	0	2	3	4	t
0	0	0	1	t	t	2

1	1	1	t	1	2	1
t	1	1	2	2	t	t
2	1	0	1	t	3	1
1	0	0	1	1	0	0
0	1	1	1	0	1	1
0	1	t	2	1	2	t
0	1	1	2	t	3	2
0	0	0	2	3	4	t
0	0	0	1	t	t	2

Figure 3.6: Demonstration of affected cells after moving a mine

3.4.2 Mine-Moving Algorithm with Heuristics

Reaching this far, the general algorithm has been formulated and works effectively on most occasions, but consider the following two scenarios ("t" represents the mines being planted, red cross represents the mis-clicked cell that should be avoided):

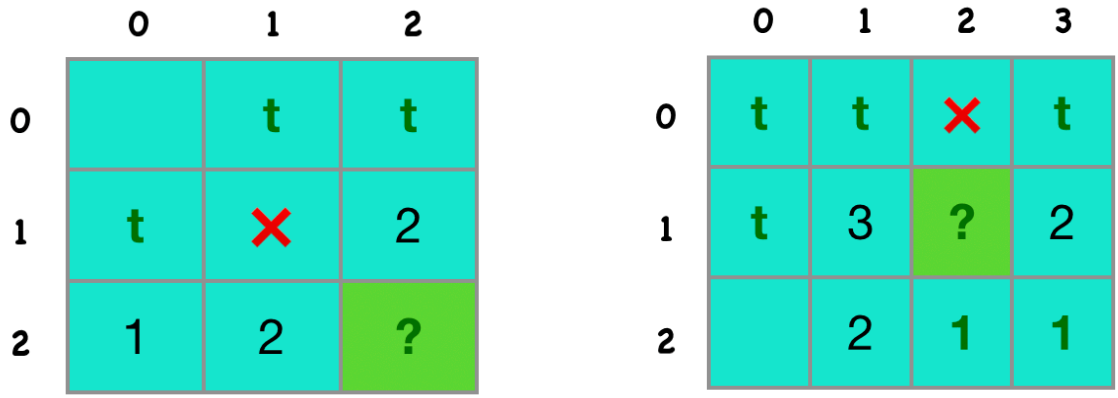

Figure 3.7: Scenarios where the previous algorithm doesn't work

For the first board where (2,0), (2,1) and (1,2) are uncovered, according to the 5-step strategy, assuming the algorithm starts with (1,2) and places two mines on (0,1) and (0,2). Then, it moves onto (2,0) and placed a mine on (1,0). Finally, it moves to (2,1) where one more mine needs to be planted to satisfy its number. Unfortunately, (1,1) needs to be avoided and (2,2) cannot be planted either as it would contaminate (1,2) because it is already satisfied. Now the algorithm is left with no vacant cells to plant the mine and reaches a dead end.

Similarly, for the second situation where (1,1) is picked and satisfied first, it leaves the algorithm stuck with (1,3) because (1,2) cannot be planted with mine following the NCP, and no more vacant cells can be found.

Both situations indicate that an improved version of the algorithm is needed. Consider the common mistake the algorithm made in both scenarios, it only focused on satisfying the one cell at hand without thinking more for other nearby outer cells. Had the agent chosen to first plant the mine at (2,2) and (1,2) first respectively for board 1 and 2, the conflict would have been resolved. It is

worth noting that both (2,2) and (1,2) have the most neighbors. (2,2) is in direct contact with 2 cells and (1,2) is adjacent to three cells at once. The more outer cells a cell is touching, the more constraint it has, which would make it harder to satisfy especially if being dealt with later. Proceeding in this direction, an improved algorithm would introduce a specific order when planting mines around an outer cell rather than randomly.

Hence, the level of constraint (i.e. number of neighbor outer cells) could be used as a heuristic that determines a sequence of mine-planting. Changes only need to take place in Step 3, where a helper method that constructs an ordered list (called `popularCellRank`, example in Figure 3.8) of cell coordinates based on the number of adjacent outer cells is provided for the algorithm to follow. By doing this, mines will always be planted to first satisfy the cell at hand, while also satisfying the most amount of other outer cells it can. This significantly decreases the occurrence of out-of-vacancies situation and also saves operation time to a degree because the upcoming outer cells in the loop are being pre-satisfied.

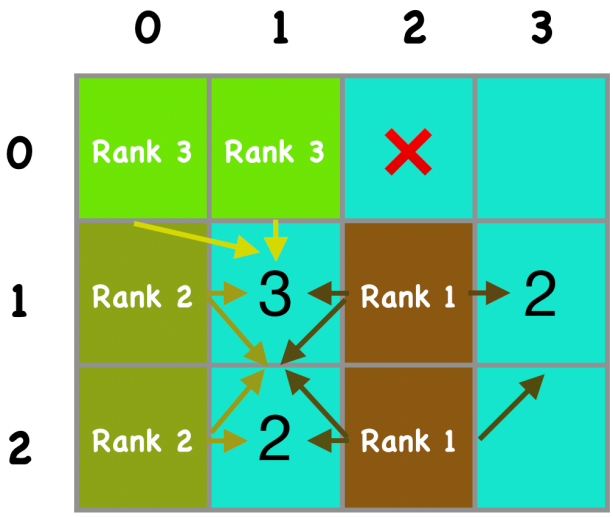

Figure 3.8: Idea of Popular cell rank

3.4.3 Recursive Mine-Moving Algorithm

`PopularCellRank` as a heuristic has shown promising improvement, but it still does not guarantee a solution 100% of the time. Observe the following scenario (Figure 3.9):

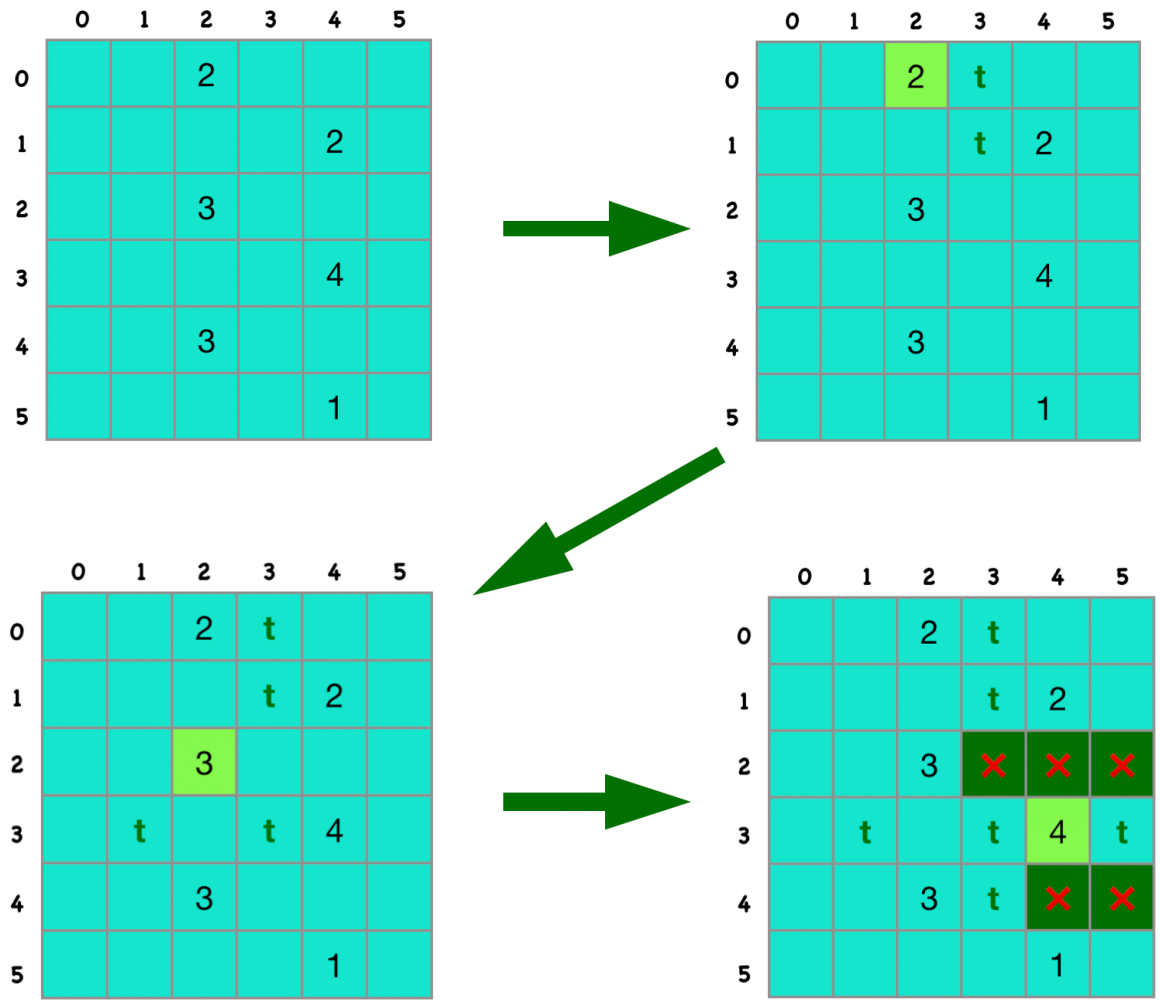

Figure 3.9: Scenario where the heuristic algorithm fail

The algorithm starts with (0,2) and proceeds downwards, satisfying outer cells along the way using the heuristic until it reaches (3,4). All the available covered cells left are not plant-safe according to the NCP, and hence the algorithm reaches the state of out-of-vacancies again. Something went wrong in the process and hence the lack of robustness of the heuristic surfaced.

The essence of this problem is that since the algorithm has no knowledge of the spaces around the subsequent outer cells, it cannot be certain about where

to plant mines around the cell at hand. There's always a possibility of running out of vacancies at some point.

Hence, a new algorithm would need to have some backtracking mechanism in place, where if a placement reaches a dead end at some point during the process, the algorithm goes back to the last or even earlier steps to adjust until a solution is found. This is realized by using recursion.

The recursive mine-moving algorithm adopts the idea of depth first search. Since the only concern is the arrangement of outer cells, a particular method `placeMines()` is singled out to take care of the process. It starts with the first outer cell in the list and generates a list of all possible ways to allocate mines in its covered neighbors. Then a for loop is run on this list, where for each possible placement, allocate it if it doesn't violate the NCP and recursively call `placeMines()` on the next outer cell in line that repeats the process. It continues this process until it has tried to place the mines for all outer cells. If at any point the placement of mines doesn't lead to a valid game state, the algorithm recognizes and backtracks to the previous outer cell and tries a different placement of mines. This backtracking process involves removing the mines placed for the current recursion, in which special attention needs to be paid to not remove coinciding mines placed during previous placement for previous outer cell from earlier level of recursion. They will be dealt with in their own level of recursion.

In this way, the algorithm explores all possible combinations of mine placements for the outer cells, and either finds a valid placement for all the mines or determines that no valid placement exists under the given constraints. Note that a non-existent solution does not exist under this context because if moving away a mine is impossible, it implies that mis-clicked mine's location must be fixed at the first place and hence axiomatically deducible. Therefore, it is an exhaustive search of valid placement which guarantees a solution. The conceptual workflow of the algorithm is visualized in Figure 3.10.

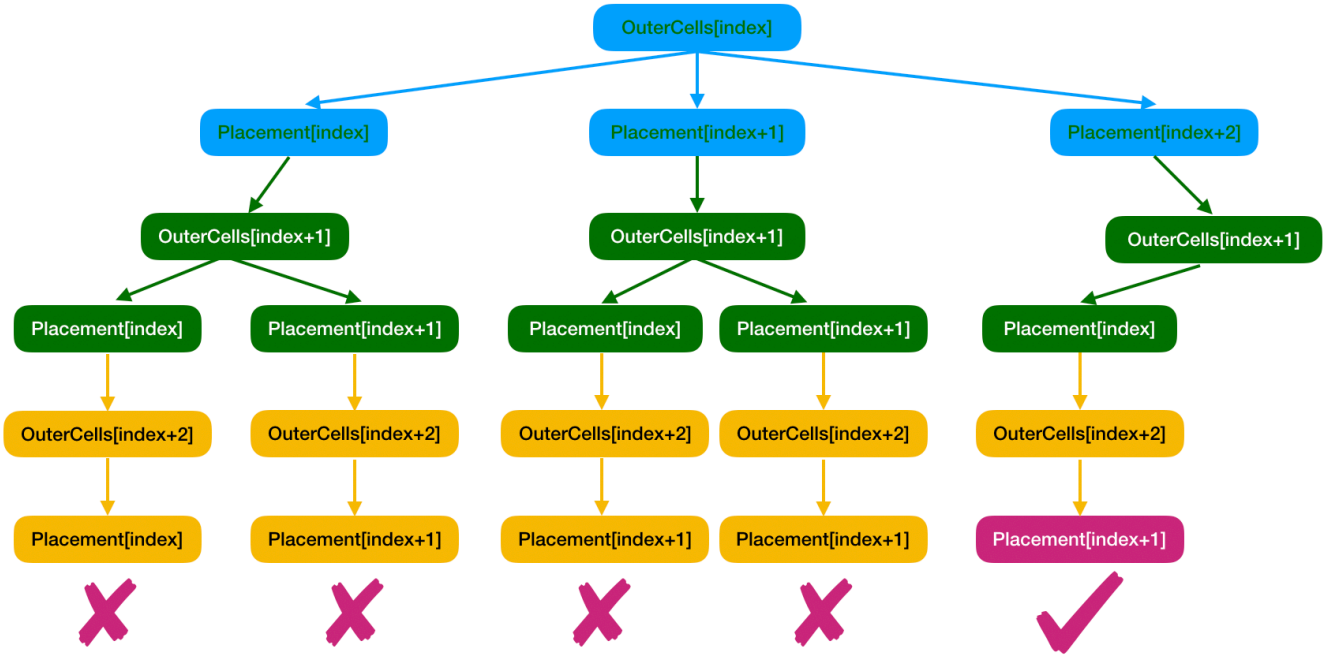

Figure 3.10: Conceptual workflow of recursive mine-moving algorithm

The structure of the visualization isn't completely accurate that represents the final algorithm. The final algorithm has one attempt at optimization of computation applied, which will be discussed further in the next section. The pseudo-code can be viewed in Appendix C.

3.5 AI Post-Game Instructor

After the player loses a game, a popup window will appear asking whether the player would like to see further steps that could be deduced based on the current board (Figure 3.11).

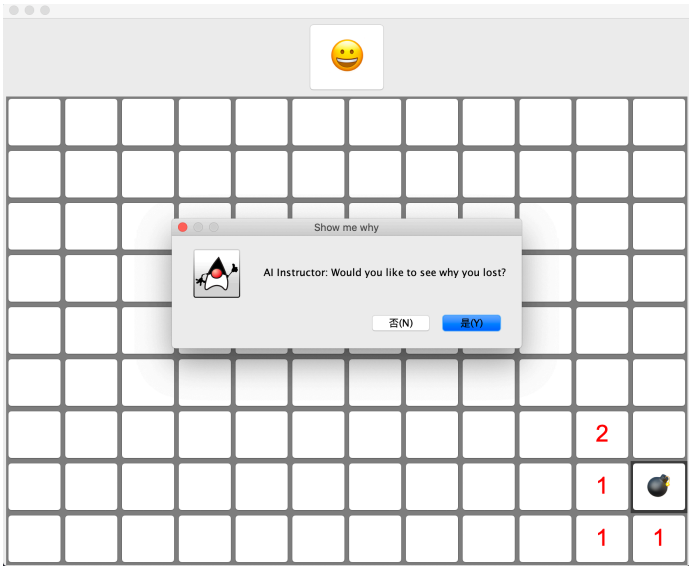

Figure 3.11: Selection to enable post-game instructor

When selected yes, the user is able to view the further logical deductions one step at a time denoted in different colors (Figure 3.12).

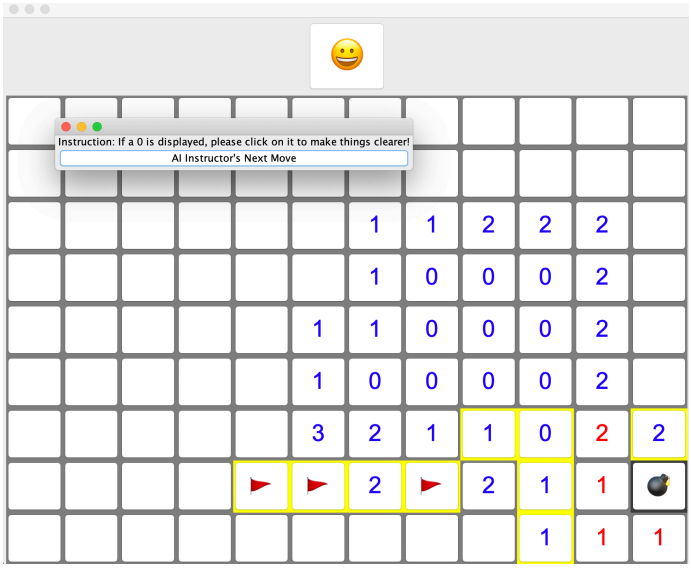

Figure 3.12: Post-game instructor with several steps shown

The said further logical steps, in practice, are the recorded steps made by the AI agent at the back end. While the agent is playing through the board, a

`path` list records the coordinate of each cell the agent made a valid action on, together with an action index (0 for flagging, 1 for probing). Since Java's ArrayLists are inherently ordered, all the moves are stored in the same sequence the agent makes. When the player loses on the GUI end, the delegate retrieves the `path` list and cleanses it by eliminating all user-uncovered cells. This leaves the delegate with a list of all the sequential further logical steps. The GUI then displays the appropriate actions (based on the action index) upon the corresponding cells one by one, a process controlled by the user's click of a "Next Move" button.

The reason that the recorded steps from the agent can be directly utilized is because the agent always progresses from the same starting line as the user. All the steps that the agent takes are strictly a derivation from the user's perspective of the uncovered board. The agent always plays ahead of the user as far as it can by making sequential logical moves. When Quantum Safety Mechanism is triggered, the agent's progress is regressed and is brought back to the new starting line same as the user, during which the content of the `path` list is discarded and restarted. This is because all previous moves are no longer relevant and only new moves from that point on would be considered relevant "further logical steps".

With the help of the AI instructor, the user could not only view the steps taken to deduce the mis-clicked mine and learn why he/she lost the game, if the user would like to see how far the AI agent has gotten (i.e. how much of the board could be deduced from the current situation), he/she could keep clicking "Next Move" until the recorded steps are exhausted. This provides the player with a quantified and visualized perspective of the progress made in the current game.

4 User Evaluation

This section discusses the process and results of evaluations from real players on the artifact. Ethical considerations are addressed (Section 4.1) and the feedbacks and corresponding actions taken are also mentioned (Section 4.2).

4.1 Ethical Considerations

In order to conduct the evaluation, an ethical application artifact form was submitted to the ethics committee (Appendix A).

The objective of the evaluation was to gather input on the improved artifact of Minesweeper. To participate, individuals had to be students studying CS or IT at the University of St Andrews, and they were to be recruited through direct personal contact. There were no ethical considerations in the process as the sole task was to act as a normal Minesweeper player that experiences the game and provides feedback.

4.2 User Feedback

Throughout the course of user evaluation, the general feedback has been satisfactory. However, miscellaneous problems were found, suggestions were given and were later dealt with.

System error: The mine-moving algorithm has consistently evolved as a result of user testing. The aforementioned corner cases that lead to disruption to the game has been discovered in this process, where out-of-vacancies situation occur when re-arranging the mines next to the outer cells. The AI agent also demonstrated faulty behavior on boards at more difficult levels, where it falsely believed that it had completely the game when it hadn’t. Updating the un-deducible cells under various circumstances were also challenged. This evaluation process had been valuable as it demonstrates the completely

different perspectives between the game engineer and the real player during gameplay, given the amount of unforeseen situations.

Enhancement on User experience: The initial version of the GUI did not support flagging a cell, as re-creating the full game wasn't the focal point of the dissertation. However, it was brought to attention that the missing of flag functionality significantly deteriorates the gameplay experience because the player has to personally remember which cells he/she deemed to be a mine. This becomes exhausting towards the end, for the large amount of mines flagged inside their heads. The aim of the work is to enhance the general gaming experience, and the effect of quantum safety mechanism cannot afford to be overshadowed. Hence, flagging was added using a right click. In addition, the application was solely deployed on one machine running on MacOS, where a right-click is realized by a single click on the trackpad using two fingers (Figure 4.1). This action isn't familiar to everyone, so accidental probing of a cell occurs more often than it should. Therefore, the game is designed in a way that one could ignore and still proceed in the game (re-flagging the accidentally probed mine) despite the announcing of loss.

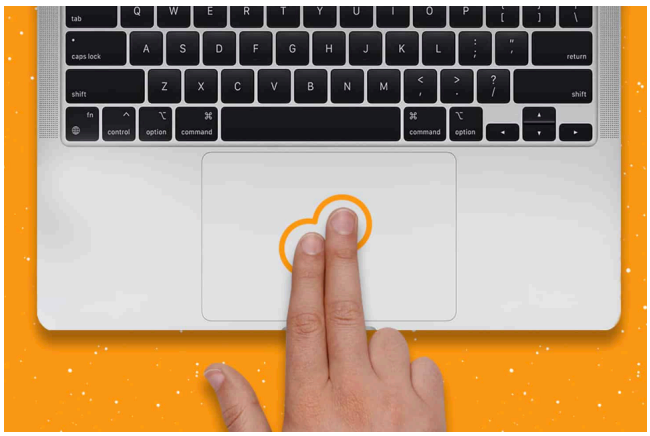

Figure 4.1: Gesture for right clicking on Mac

5 Discussion

With all components in place and in action, the general workflow of the entire project can be summarized as follow (Figure 5.1):

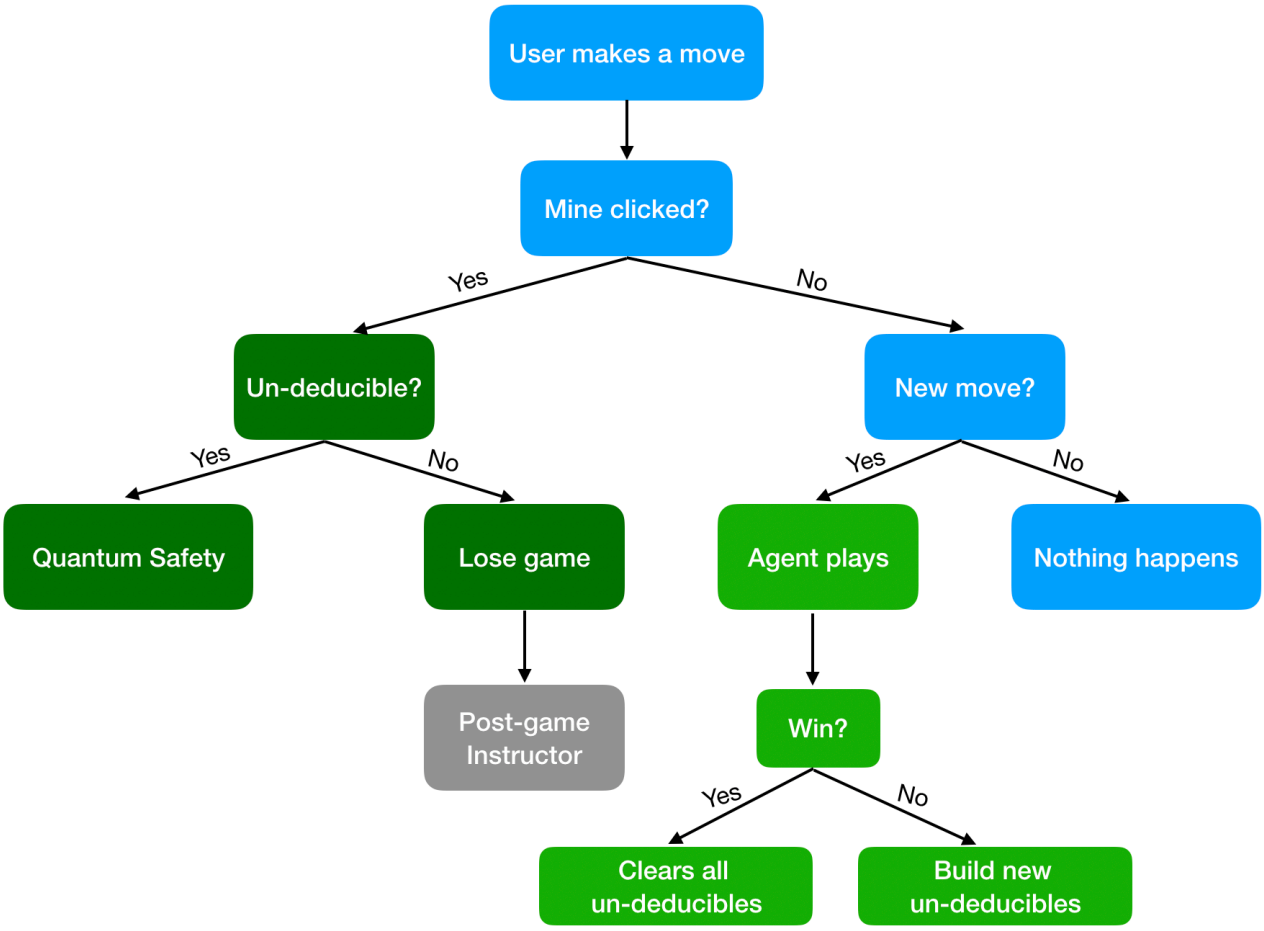

Figure 5.1: General workflow of the project as a whole

This section will mainly focus on the discussion on limitations from two major components and how they potentially pose threat to the system - the recursive mine-moving algorithm (Section 5.1) and the AI agent (Section 5.2).

5.1 Limitations of Mine-Moving Algorithm

The final mine-moving algorithm that guarantees a solution is in essence an exhaustive search of one placement that maintains the valid state of the entire game. It searches through all combinations of possible placements of all

outer cells until a solution is found. Therefore, in the worst case scenario, the algorithm might traverse through all the possibilities. The computation could potentially be severely costly. The time complexity of such algorithm is exponential O(m^n), with m being the number of placement options and n being the number of outer cells. Here's the breakdown of the time complexity:

The first recursion starts with the first outer cell, assuming it has m possible placements of mines, which is m leaf cells on the first level of the conceptual tree (Figure 3.7). For each branch cell, another m possible placements of mines are put in line from the second outer cell, which gives m^2 *leaf cells. Assuming each outer cell unanimously has m placements, the eventual number of leaf cells is indeed* m^n*. Note that m is the combination of k cells with p mines:*

$$\boldsymbol{k! / (q! \times (k - q)!)}$$

As costly as it might be, in a realistic setting, the odds of having to traverse through all possible placements with a large base is statistically insignificant. Most computation load incurred can be dealt with in trivial time. However, a decent optimization strategy could still be applied to improve the algorithm.

The largest combination base possible here is C(8,4) = 70, if the algorithm starts with a cell with value 4, the first level of the traversal tree will have 420 branches. Assume that somewhere in the list of outer cells, there is a cell with value of 1, which poses the most constraints upon its surroundings. The probability of the depth first search traversal reaching this cell and failing, thus having to backtrack is very high. A lot of traversals from the first level branches would be wasted. Therefore, starting with satisfying cells with lower values could not only significantly decrease the size of the tree on shallow levels, but also starkly increase the chance of finding a solution earlier, thus saving computation load. In practice, the list of outerCells is sorted in ascending order based on the values on the cells. Although the combination base decreases again

after 4 and C(8,8) has the lowest, the odds of having cells with such high values are statistically low and hence negligible.

To conclude, the recursive mine-moving algorithm is computationally intensive and has exponential time complexity. Even though the natural odds of a bad scenario is low and an optimization strategy is set to further avoid it, as the size of the board increases and the game gets more difficult with more mines, such scenario might occur occasionally. According to the workflow diagram, the GUI doesn't display a value until the quantum safety procedure is finished. This means that a lengthy delay in response after an un-deducible mine click is possible in the worst scenario, which poses threat to the gameplay experience. Future studies could concentrate on devising a comprehensive optimization strategy, perhaps combining the aforementioned heuristics, to improve the algorithm's efficiency in games of all sizes.

5.2 Limitations of AI Agent

Similar to the limitations of the algorithm, the AI requires more time to explore the game board to the best of its abilities as the board size increases. When a user makes a new move, the agent re-builds the logic sentence that represents the knowledge base of the entire board (Figure 5.2), and processes a heavy load of satisfiability test on it. As the board expands, the length of the logical sentence and the amount of computation increases linearly (Figure 5.3). The amount of computation that needs to be made also increases with the level of complication of the current board's circumstance.

As the cell value will only be displayed on the GUI after the agent's gameplay, another potential threat of delay in frontend response is introduced. The delay will likely lengthen as the game size expands and diminish the user experience to certain extent. A possible way to improve the current model would be to save a partial knowledge base logic representation and make

necessary changes to it when new moves are made, through which time can be saved from re-constructing an entire knowledge base with repeated parts every time. Future studies could also introduce multi-threading to the system to minimize delay and enhance gameplay experience.

```
Knowledge Base:
((D08&D010&D18&~D19&~D110)|(D08&D010&~D18&D1
9&~D110)|(D08&~D010&D18&D19&~D110)|(~D08&D01
0&D18&D19&~D110)|(D08&D010&~D18&~D19&D110)|(
D08&~D010&D18&~D19&D110)|(~D08&D010&D18&~D19
&D110)|(D08&~D010&~D18&D19&D110)|(~D08&D010&
~D18&D19&D110)|(~D08&~D010&D18&D19&D110))&((
D010&~D110&~D111)|(~D010&D110&~D111)|(~D010&
~D110&D111))
```

Figure 5.2: Knowledge base representation example with 2 outer cells

```
Knowledge Base:
((D30&~D31&~D50)|(~D30&D31&~D50)|(~D30&~D31&D50))&((D30&D31&~D32&~
D50)|(D30&~D31&D32&~D50)|(~D30&D31&D32&~D50)|(D30&~D31&~D32&D50)|(
~D30&D31&~D32&D50)|(~D30&~D31&D32&D50))&((D31&~D32&~D33)|(~D31&D32
&~D33)|(~D31&~D32&D33))&((D32&~D33&~D34)|(~D32&D33&~D34)|(~D32&~D3
3&D34))&((D34&~D35&~D36&~D46&~D55&~D56)|(~D34&D35&~D36&~D46&~D55&~
D56)|(~D34&~D35&D36&~D46&~D55&~D56)|(~D34&~D35&~D36&D46&~D55&~D56)
|(~D34&~D35&~D36&~D46&D55&~D56)|(~D34&~D35&~D36&~D46&~D55&D56))&(D
50)&(D55)&(D50)&(D50)&(D55)&((D55&~D56)|(~D55&D56))&((D55&~D56&~D5
7)|(~D55&D56&~D57)|(~D55&~D56&D57))&((D56&~D57&~D58&~D68)|(~D56&D5
7&~D58&~D68)|(~D56&~D57&D58&~D68)|(~D56&~D57&~D58&D68))&(D68)&((D6
8&~D69)|(~D68&D69))&((D68&D69&~D610&~D710&~D810)|(D68&~D69&D610&~D
710&~D810)|(~D68&D69&D610&~D710&~D810)|(D68&~D69&~D610&D710&~D810)
|(~D68&D69&~D610&D710&~D810)|(~D68&~D69&D610&D710&~D810)|(D68&~D69
&~D610&~D710&D810)|(~D68&D69&~D610&~D710&D810)|(~D68&~D69&D610&~D7
10&D810)|(~D68&~D69&~D610&D710&D810))&((D710&~D810)|(~D710&D810))
```

Figure 5.3: Knowledge base representation example with 15 outer cells

6 Conclusion

The overall goal of this project was to introduce a Quantum Safety Mechanism to the game of Minesweeper that aids the play throughout the game by moving the mine away when the click is logically un-deducible, thus eliminating the element of luck out of the equation. Of the objectives outlined in the beginning, the project was successful in completing all primary and secondary objectives. The final products of the work are this dissertation report and the complete playable Minesweeper with Quantum Safety Mechanism and post-game AI instructor.

The goal and the artifact is achieved by setting up the basic interactive frontend Minesweeper GUI and the backend board control. A two-way communication channel is built between the two ends. An artificial intelligence agent is created that is able to play a full game of Minesweeper on its own, which enables it to be the arbiter of the deducibility of mines alongside the player throughout an actual game. An iteratively refined mine-moving algorithm is devised to find a guaranteed new and safe placement of mines when the QSM is triggered at any point of a game. After the loss of a game, the post-game instructor uses recorded information provided by the AI agent to demonstrate further logically sound steps to the player on the GUI. The robustness and correctness of the system was tested and evaluated with real users, from whom feedbacks were collected and acted upon accordingly. The overall feedback was positive and showed that it was an improvement in gaming experience to the original version.

6.1 Limitations and Future Work

The current system, while soundly working, has demonstrated weakness in computation load and time complexity in games with larger and more complicated boards, mainly from the AI agent and the mine-moving algorithm.

Further studies in this direction can be pursued on optimization strategies to decrease the computation incurred in the process, as well as introducing multi-threading to the artifact to reduce delays in response during the gameplay and further enhance the gaming experience.

The proposed Quantum Safety Mechanism could also be referenced and incorporated into other applications of similar nature where player actions could lead to a failure state without the player having the necessary information to prevent it, such as Solitaire, Sudoku and Chess, etc. For instance, Sudoku involves logical deduction based on current information. If the player is about to place a number in a location that would make the puzzle unsolvable based on future placements, the Quantum Safety Mechanism could step in and reconfigure the remaining cells to ensure the puzzle stays solvable.

Lastly, if future interest in further elevating the difficulty of Minesweeper with Quantum Safety Mechanism is expressed by scholars, the theory of probability in Minesweeper suggested in [8] could be considered as an optional trigger for the QSM, where if the player clicks on an un-deducible yet probabilistically-high mine, he/she loses the game. The theory suggests that not all un-deducible situations have to be a 50-50 guess. If the remaining number of the mines and all possible arrangements that makes the current board sound (i.e. not breaching the NCP) are considered, cells hold different probabilities of being the potential mine. However, enabling this would require a mine counter implemented that is visible to the user on the GUI, a functionality the artifact in this work has left out, for him/her to deduce the probabilities. This work was not mentioned in early sections due to its low relevance to this dissertation's focal task, but it holds high significance to any future exploration regarding this topic direction. Furthermore, the post-game instructor could also include this and be designed to explain and demonstrate those probabilities and how they are calculated, which brings the gameplay experience to higher analytical level.

A Artifact Evaluation Form

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
ARTIFACT EVALUATION FORM

Title of project

Minesweeper with quantum safety

Name of researcher(s)

Kaan Tekin

Name of supervisor

Michael Young

Self audit has been conducted **YES** ☐

This project is covered by the ethical application CS15727.

Signature Student or Researcher

Print Name

Kaan Tekin

Date

2023/05/26

Signature Lead Researcher or Supervisor

Print Name

Michael Young

Date

2023/05/26

B Pseudo-code for Generic QSM

```
Public void reMap(int x, int y){

      // Regress agent progress
      reset uncovered -> userUncovered;

      // Get all outer cells
      ArrayList<ArrayList<Integer>> outerCells;
      for (i,j = 0; i,j<board.length; i,j++) {
            if (uncovered[i][j] && coveredNeighbor.size>0){
                  outerCells.add(i,j)
            }
      }

      // Copy content of gameBoard and reset it to all zeros
      char[][] temp <— gameBoard & reset gameBoard -> all '0';

      // Re-arrange mines around the outer cells
      for (each cell : outerCells) {
            avail = getNeighbors(cell.row, cell.col);
            int numMines = temp[cell.row][cell.col];
            while (numMines>0) {
                  if (not the mis-clicked mine) {
                        if (already a mine) {
                              numMines-1;
                        } else {
                              if (not violate NCP) {
                                    plantMine;
                                    numMines-1;
                              }
                        }
                  }
            }
      }

      Randomly place mines for rest of the board;
      Generate new numbers based on those mines;

      // Re-initialize the agent's board and frontier
      for (i,j = 0; i,j<board.length; i,j++) {
            if (!uncovered[i][j]){
                  if (agent uncovered prior to the reMap) {
                        frontier.add(i,j);
                        board[i][j] = '?';
                  }
            }
      }
      // Clear all previously undeducibles
      undeducibles.removeAll();

}
```

C Pseudo-code for Recursive QSM

```
Public void reMapRecursive(int x, int y){

      Regress progress;
      outerCells = getOuterCells();
      Copy content of gameBoard and reset to zeros;

      placeMines(temp, 0, outerCells, outerMineCount,x,y);

      Rearrange rest of the board;
      Re-initialize agent's board and frontier;
      Clear undeducibles;
}

// Finding a valid placement for one outer cell
Public boolean placeMines(temp, index, outerCells, count, x, y) {

      // Base case: if all outer cells have been handled
      if (outerCellIndex == outerCells.size()) {
          // Found a solution        // No solution
          if (totalMines == 0) {true;} else {false;}
      }

      cell = outerCells.get(index);
      int numMines = temp[cell.row][cell.col];
      avail = getNeighbors(cell.row, cell.col);
      // Gets all possible ways to place mines given the vacant slots
      possiblePlacements = generateAllPlacements(avail, numMines);

      for (each placement : possiblePlacements) {

            // Record mines planted in previous recursions in case of
               accidental removal
            if (cell in placement already is a mine) {oldMine.add(cell)}

            if (isValid(eachPlacement, x, y) && more mines await plant) {
                  plant mines using this placement;
                  // Move onto next recursion for next outer cell
                  if (placeMines(temp, index+1, outerCells, totalMines -
                  eachPlacement.size(), x, y)) {
                      return true; // Found a solution
                  }

                  // If next recursion failed, backtrack and erase the
                     mines in this recursion. Don't erase old mines
                  if (not oldMine) {reset the placed mines back to '?';}

            }
      }
      return false; // No solution for this cell under the current board
}
```

Glossary

NCP Non Contamination Principle

SATS Satisfiability Test Strategy

GUI Graphical User Interface

AI Artificial Intelligence

QSM Quantum Safety Mechanism

CSP Constraint Satisfaction Problem

AMN All Marked Neighbors

AFN All Free Neighbors

PAFG Primary and Advanced reasoning, First action and Guessing strategy

References

[1] Edwards, Benj (2021). "30 Years of 'Minesweeper' (Sudoku with Explosions)". *How-To Geek*. https://www.howtogeek.com/693898/30-years-of-minesweeper-sudoku-with-explosions/

[2] Unknown author (2022). “How To Play Minesweeper”. *Authoritative Minesweeper*. https://minesweepergame.com/strategy/how-to-play-minesweeper.php

[3] Damein Moore (2021). “Strategy”. *Minesweepergame*. https://www.minesweeper.info/wiki/Strategy.

[4] Chang Liu (2022). “A solver of single-agent stochastic puzzle: A case study with Minesweeper”. *ScienceDirect*. https://www.sciencedirect.com/science/article/pii/S0950705122002842#sec4

[5] Andrew Adamatzky (1997). “How cellular automaton plays minesweeper”. *ScienceDirect*. https://www.sciencedirect.com/science/article/pii/S0096300396001178

[6] Beccera, David J (2015). “Algorithmic Approaches to Playing Minesweeper”. *Harvard Library*. https://dash.harvard.edu/bitstream/handle/1/14398552/BECERRA-SENIORTHESIS-2015.pdf

[7] Chris Studholme (2000). “Minesweeper as a Constraint Satisfaction Problem”. *University of Toronto Project Report*. http://www.cs.toronto.edu/~cvs/minesweeper/minesweeper.pdf

[8] Sean Barrett (1999). “Advance Minesweeper Tactics”. *Nothings*. https://nothings.org/games/minesweeper/#:~:text=Minesweeper:AdvancedTactics

[9] Alammar, Mohammed M (2021). “A Minesweeper Algorithm for Improved Signal Area Estimation in Spectrum Aware Systems”. *28th International Conference on Telecommunications (ICT)* .https://ieeexplore.ieee.org/document/9511512?arnumber=9511512

[10] M Hamza Sajjad (2022). “Neural Network Learner for Minesweeper”. *Cornell University*. https://arxiv.org/abs/2212.10446

[11] Sinha, Yash Pratyush (2021). “Fast constraint satisfaction problem and learning-based algorithm for solving Minesweeper”. *Cornell University*. https://arxiv.org/abs/2105.04120

[12] Mehta, Anav (2021). “Reinforcement Learning For Constraint Satisfaction Game Agents”. *Cornell University*. https://arxiv.org/abs/2102.06019

[13] Unknown author (2023). “What Is Superposition & Why Is It Important?”. *Caltech Science Exchange*. https://scienceexchange.caltech.edu/topics/quantum-science-explained/quantum-superposition

[14] Unknown author (2023). “What Is Quantum Physics?”. *Caltech Science Exchange*. https://scienceexchange.caltech.edu/topics/quantum-science-explained/quantum-physics

[15] Kaye, R. (2000). “Minesweeper is NP-complete”. *Mathematical Intelligencer 22(2), 9-15.*

[16] Czengler (2022) “LogicNG - The Next Generation Logic Framework”. *LogicNG.* https://logicng.org/

[17] Baeldung (2020) “Java Bitwise Operators”. *Baeldung.* https://www.baeldung.com/java-bitwise-operators

[18] Le Berre (2023) “Solver Documentation”. sat4j.org. http://www.sat4j.org/maven233/apidocs/org/sat4j/minisat/core/Solver.html

[19] Mun See Chang (2023) “The Danger Sweeper Knowledge Base”. https://studres.cs.st-andrews.ac.uk/CS5011/Lectures/L12-Logic3.pdf

[20] Tang, Yimin (2018). “A Minesweeper Solver Using Logic Inference, CSP and Sampling”. *Cornell University*. https://arxiv.org/abs/1810.03151