Minesweeper Qt-Project
CMPE 230 - Systems Programming
02/06/2024

Yusuf Aygün **2020400033**

Contents

1 Introduction

The objective of this project is to create a single player Minesweeper game using Qt and C++. The goal of the game is to reveal all cells without triggering any mines. Each cell displays the number of adjacent mines, and players can flag suspected mines. The game board is initially filled with unrevealed cells, and the number of mines is set based on the game configuration.

The game features a configurable grid, typically starting with a 10x10 layout and 10 mines. The player is expected to click on cells to reveal their content. If a cell with no adjacent mines is revealed, all neighboring cells are recursively revealed. Right-clicking on a cell allows the player to flag it as a suspected mine, preventing accidental clicks on potential mines. If a mine is revealed, the game is lost. The game continues until all non-mine cells are revealed, resulting in a win.

The player can see their progress through a score displayed at the top of the screen, which shows the number of cells revealed. A hint button is available to suggest a safe cell to reveal, helping the player when they are unsure of the next move. The restart button resets the game, shuffling the mines and allowing the player to start a new game with fresh configuration settings.

The game ends with a win if all non-mine cells are revealed or a loss if a mine is triggered. A popup message indicates the game outcome, and all mines are revealed. The player can start a new game by closing the popup and clicking the "Restart" button.

The project includes several core components: the main window, custom buttons for flagging, and the logic for game initialization and hint functionality. The solution contains multiple source and header files, which will be explained in detail in the following chapters.

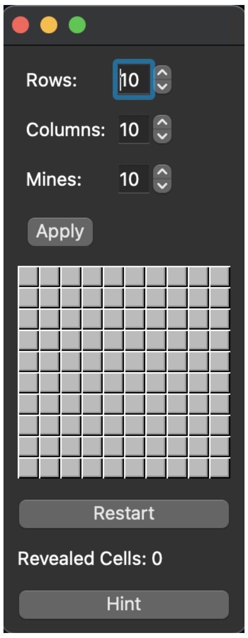
2 Program Interface

To run the program, ensure that Qt is installed on your computer. Open Qt Creator, click "Open Project," and select the `Minesweeper.pro` file. After the project is indexed, click the run button or press `Ctrl+R` to run the project. The game window will open, allowing you to play the game. You can close the window to quit the program.

3 Program Execution

3.1 Game Initialization

The game initializes with a default grid of 10x10 and 10 mines. The number of rows, columns, and mines can be configured via the UI.

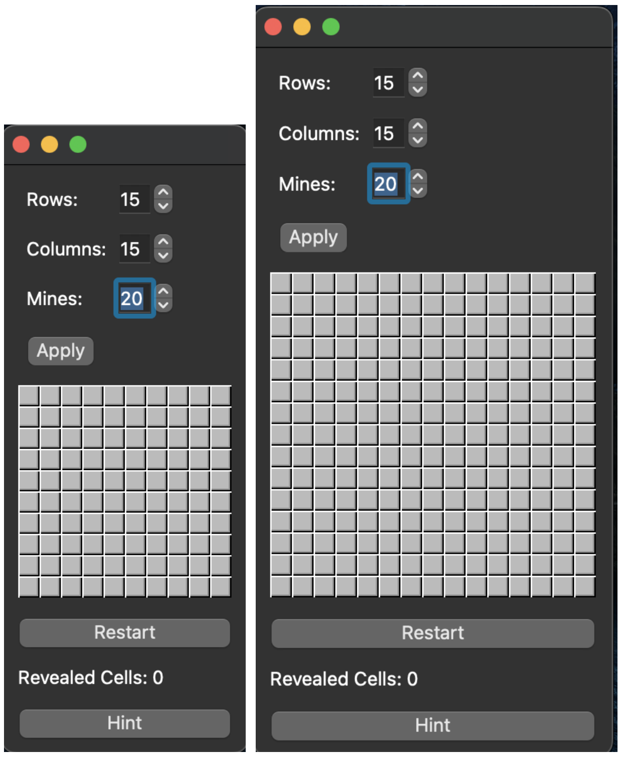

Rows: 10
Columns: 10
Mines: 10

Revealed Cells: 0

Figure 1: Main Screen

3.2 Settings Application

Users can apply new settings for the number of rows, columns, and mines. The game grid updates accordingly.


Rows: 15
Columns: 15
Mines: 20

Revealed Cells: 0

Rows: 15
Columns: 15
Mines: 20

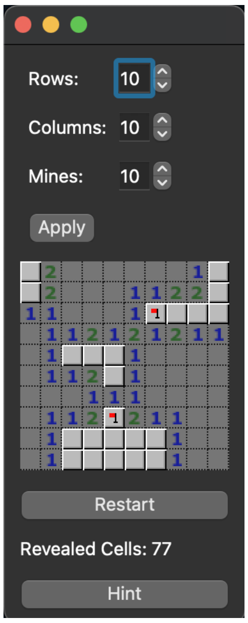
Revealed Cells: 0

(a) Before (b) After

Figure 2: Settings

3.3 Gameplay

The game allows left-click to reveal cells and right-click to flag suspected mines. If a mine is revealed, the game ends with a loss. Revealing all non-mine cells results in a win.



Rows: 10

Columns: 10

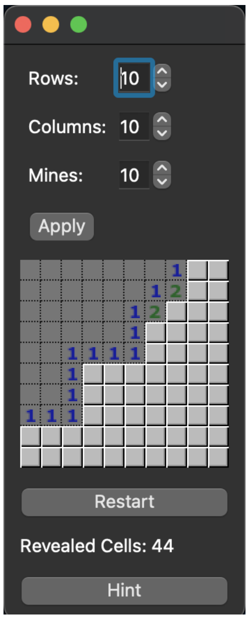
Mines: 10

Revealed Cells: 77

Figure 3: Game in Progress

3.4 Hint Functionality

The hint button suggests a safe cell to reveal. If pressed again, it reveals the suggested cell.



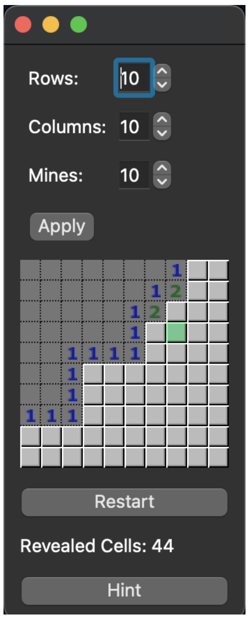
Rows: 10

Columns: 10

Mines: 10

Revealed Cells: 44

(a) Before Click



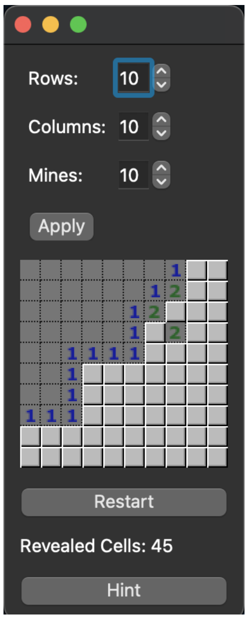
Rows: 10

Columns: 10

Mines: 10

Revealed Cells: 44

(b) After First Click



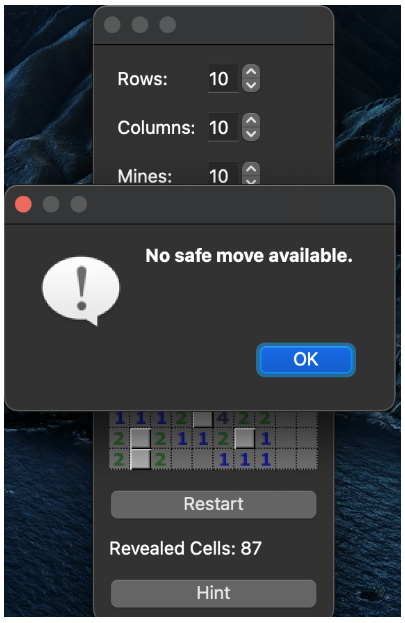
Rows: 10

Columns: 10

Mines: 10

Revealed Cells: 45

(c) After Second Click



Rows: 10

Columns: 10

Mines: 10

No safe move available.

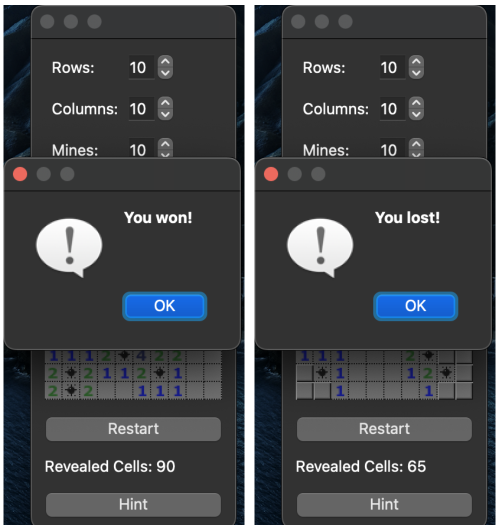
Revealed Cells: 87

(d) No Hint Found Failure

Figure 4: Hint Button

3.5 End Game Conditions

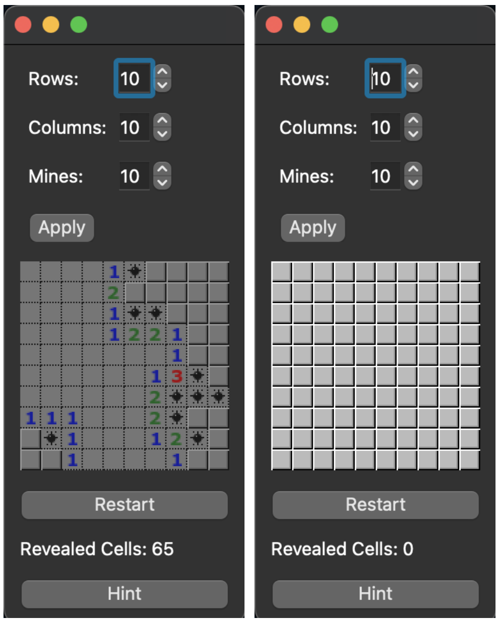
The game ends with a win or loss. A popup message indicates the result. The restart button is always available to start a new game.


Rows: 10
Columns: 10
Mines: 10
You won!
Revealed Cells: 90

Rows: 10
Columns: 10
Mines: 10
You lost!
Revealed Cells: 65

(a) Game Won (b) Game Lost

Figure 5: End Game


Rows: 10
Columns: 10
Mines: 10
Revealed Cells: 65

Rows: 10
Columns: 10
Mines: 10
Revealed Cells: 0

(a) Before (b) After

Figure 6: Restart Button

4 Input and Output

4.1 Input

The user interacts with the game through mouse clicks. Left-click reveals cells, while right-click toggles flags on cells. The number of rows, columns, and mines can be set through the UI.

4.2 Output

The game displays a grid of cells. Revealed cells show numbers indicating the number of adjacent mines, or an empty cell if no adjacent mines are present. Flagged cells are marked, and a score indicating the number of revealed cells is displayed.

5 Program Structure

The project consists of the following files:

- `Minesweeper.pro`
- `main.cpp`
- `mainwindow.h` and `mainwindow.cpp`
- `flagbutton.h` and `flagbutton.cpp`
- `resources.qrc`

5.1 Minesweeper.pro

This `.pro` file contains the commands necessary for building the project with qmake. It includes references to the header, source, and resource files required for the project. Qt Creator uses this file as a project description, allowing it to build and run the project efficiently.

5.2 main.cpp

The `main.cpp` file is the entry point of the application. It initializes the Qt application and displays the main window. The main function creates an instance of the QApplication class, which manages application-wide resources and control flow. It then creates an instance of the MainWindow class, shows it, and enters the main event loop by calling `exec()`. This loop waits for user interactions and dispatches events to the appropriate widgets.

5.3 MainWindow Class

The `MainWindow` class is responsible for managing the overall game logic and user interface. It is defined in the `mainwindow.h` and `mainwindow.cpp` files. The class includes several private methods and slots that handle various aspects of the game, such as initializing the game grid, handling cell clicks, and providing hints to the player.

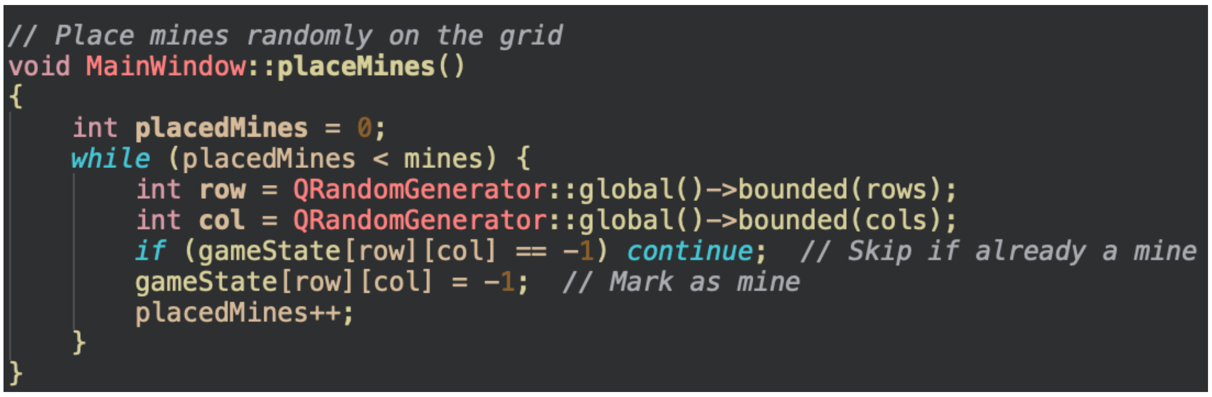
When the MainWindow is constructed, it calls the `initializeGame()` method. This method sets up the game by creating a grid of buttons, each representing a cell in the Minesweeper game. The grid layout is dynamically generated based on the specified number of rows and columns. Each button in the grid is an instance of the `FlagButton` class, which extends `QPushButton` to include flagging functionality.

The `FlagButton` class emits a `rightClicked()` signal when the right mouse button is pressed, allowing the main window to handle flagging operations. When a cell is left-clicked, the `handleCellClick()` slot is called. This method determines if the clicked cell contains a mine. If it does, the game ends, and a message box informs the player that they have lost. If the cell does not contain a mine, it is revealed, and the game state is updated accordingly.

Revealing a cell involves several steps. If the cell has no adjacent mines, all neighboring cells are recursively revealed. This is achieved by calling the `revealCell()` method, which checks the number of adjacent mines and updates the cell's display. If the cell has adjacent mines, it displays the number of mines. The `updateScore()` method is called to update the player's score based on the number of revealed cells.

5.3.1 Placing Mines Randomly

Before the game starts, mines are placed randomly on the grid. This is done by the `initializeGame()` method. The method uses a random number generator to place a specified number of mines on the grid without any predetermined pattern. Each cell's mine status is tracked using a two-dimensional array. Care is taken to ensure that no two mines occupy the same cell, and the placement process continues until all mines are placed.



```cpp
// Place mines randomly on the grid
void MainWindow::placeMines()
{
    int placedMines = 0;
    while (placedMines < mines) {
        int row = QRandomGenerator::global()->bounded(rows);
        int col = QRandomGenerator::global()->bounded(cols);
        if (gameState[row][col] == -1) continue;  // Skip if already a mine
        gameState[row][col] = -1;  // Mark as mine
        placedMines++;
    }
}
```

Figure 7: `placeMines` Method

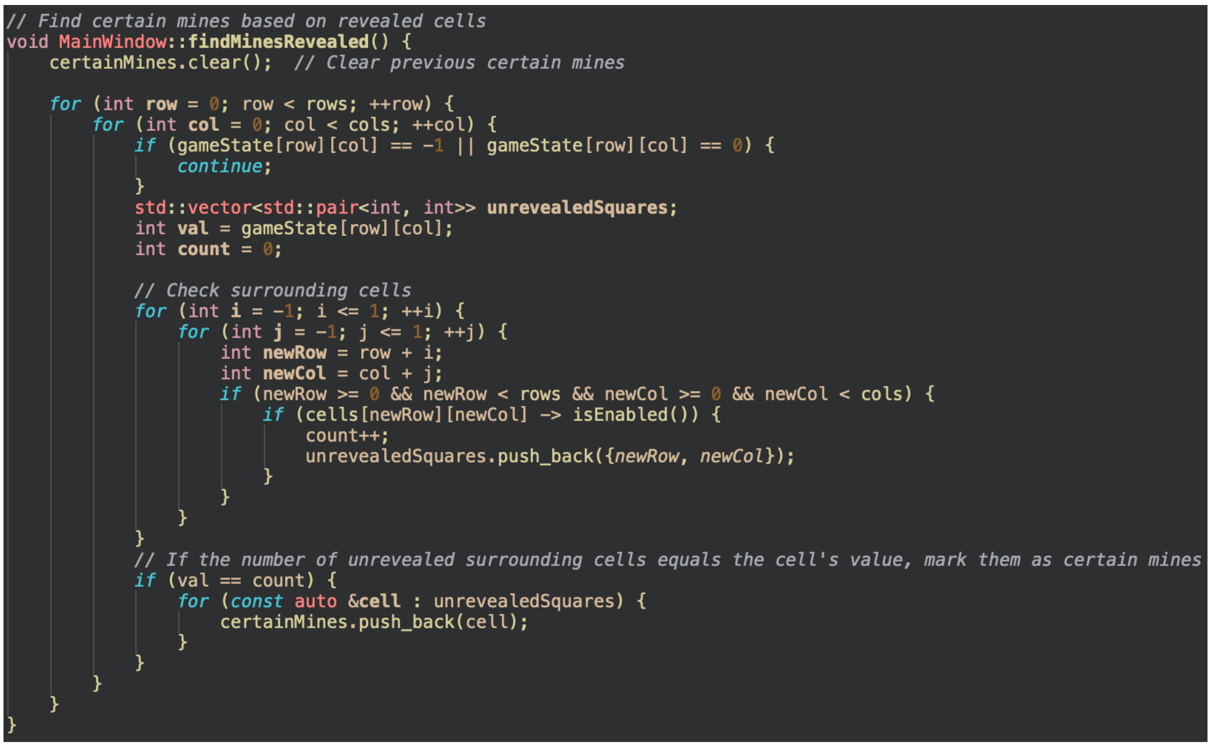
5.3.2 Adjusting Grid Sizes

The grid size is adjustable based on the player's preference or the game's configuration settings. The `initializeGame()` method reads the desired number of rows, columns, and mines from the configuration and dynamically adjusts the grid layout. This flexibility allows for various game difficulties and ensures the game can scale appropriately to different grid sizes.

5.3.3 Using Revealed Information for Hints

When a cell is revealed, its number indicates how many adjacent cells contain mines. The `revealCell()` method updates the grid based on this information. The game logic also keeps track of all revealed cells and their numbers, using this data to provide accurate hints.

To provide hints, the `findMinesRevealed()` method analyzes the revealed cells and calculates the probabilities of adjacent unrevealed cells containing mines. And the `findSafeCells()` method identifies cells that are guaranteed to be safe based on logical deduction by using the numbers on revealed cells. For example, if a revealed cell with the number 1 has only one adjacent unrevealed cell, that cell must contain a mine, and if there is another revealed cell with number 1 adjacent to that cell, all other adjacent cells of the cell with number 1 are considered safe. This logic helps the `findHint` method in deciding which cell to reveal next as a hint.

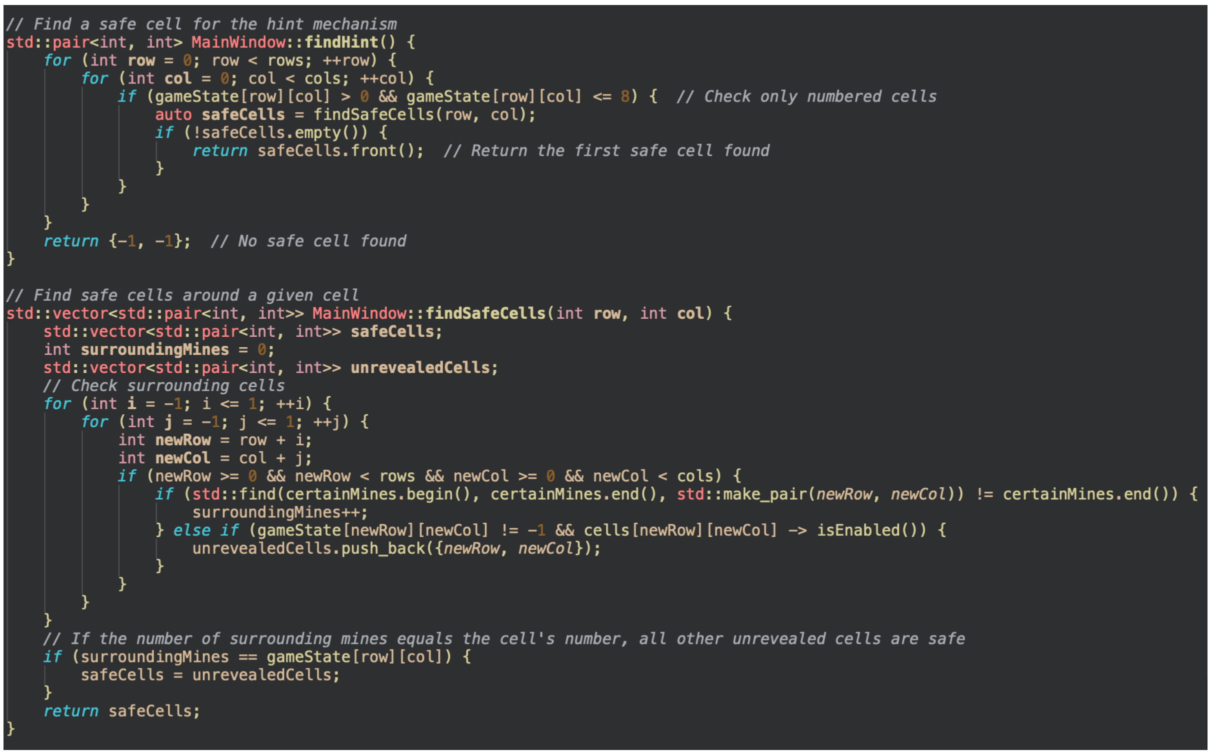


```cpp
// Find certain mines based on revealed cells
void MainWindow::findMinesRevealed() {
    certainMines.clear();  // Clear previous certain mines

    for (int row = 0; row < rows; ++row) {
        for (int col = 0; col < cols; ++col) {
            if (gameState[row][col] == -1 || gameState[row][col] == 0) {
                continue;
            }
            std::vector<std::pair<int, int>> unrevealedSquares;
            int val = gameState[row][col];
            int count = 0;

            // Check surrounding cells
            for (int i = -1; i <= 1; ++i) {
                for (int j = -1; j <= 1; ++j) {
                    int newRow = row + i;
                    int newCol = col + j;
                    if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols) {
                        if (cells[newRow][newCol] -> isEnabled()) {
                            count++;
                            unrevealedSquares.push_back({newRow, newCol});
                        }
                    }
                }
            }
            // If the number of unrevealed surrounding cells equals the cell's value, mark them as certain mines
            if (val == count) {
                for (const auto &cell : unrevealedSquares) {
                    certainMines.push_back(cell);
                }
            }
        }
    }
}
```

Figure 8: `findMinesRevealed` Method



```cpp
// Find a safe cell for the hint mechanism
std::pair<int, int> MainWindow::findHint() {
    for (int row = 0; row < rows; ++row) {
        for (int col = 0; col < cols; ++col) {
            if (gameState[row][col] > 0 && gameState[row][col] <= 8) {  // Check only numbered cells
                auto safeCells = findSafeCells(row, col);
                if (!safeCells.empty()) {
                    return safeCells.front();  // Return the first safe cell found
                }
            }
        }
    }
    return {-1, -1};  // No safe cell found
}

// Find safe cells around a given cell
std::vector<std::pair<int, int>> MainWindow::findSafeCells(int row, int col) {
    std::vector<std::pair<int, int>> safeCells;
    int surroundingMines = 0;
    std::vector<std::pair<int, int>> unrevealedCells;
    // Check surrounding cells
    for (int i = -1; i <= 1; ++i) {
        for (int j = -1; j <= 1; ++j) {
            int newRow = row + i;
            int newCol = col + j;
            if (newRow >= 0 && newRow < rows && newCol >= 0 && newCol < cols) {
                if (std::find(certainMines.begin(), certainMines.end(), std::make_pair(newRow, newCol)) != certainMines.end()) {
                    surroundingMines++;
                } else if (gameState[newRow][newCol] != -1 && cells[newRow][newCol] -> isEnabled()) {
                    unrevealedCells.push_back({newRow, newCol});
                }
            }
        }
    }
    // If the number of surrounding mines equals the cell's number, all other unrevealed cells are safe
    if (surroundingMines == gameState[row][col]) {
        safeCells = unrevealedCells;
    }
    return safeCells;
}
```

Figure 9: `findHint` and `findSafeCells` Methods

The `showHint()` method then highlights a safe cell for the player to reveal, guiding their next move. This feature is particularly useful in complex scenarios where the next move is not immediately obvious to the player.

5.3.4 End Game Conditions

The game ends when the player reveals a mine or successfully reveals all non-mine cells. If a mine is revealed, the game ends with a loss, and all mines are displayed. If all non-mine cells are revealed, the game ends with a win. In either case, a popup message informs the player of the result. The restart button is always available, allowing the player to reset the game and start a new session with a different mine layout.

5.4 FlagButton Class

The `FlagButton` class, defined in `flagbutton.h` and implemented in `flagbutton.cpp`, extends `QPushButton` to add flagging functionality. It overrides the `mousePressEvent()` method to emit a custom `rightClicked()` signal when the right mouse button is pressed. This signal is connected to the `handleCellRightClick()` slot in the `MainWindow` class, which toggles the flag state of the clicked cell. Flagged cells are visually marked, helping the player keep track of suspected mines.

5.5 resources.qrc

The `resources.qrc` file contains references to any additional resources needed for the game, such as images or icons. It uses the Qt Resource System to embed these resources directly into the executable, making them easily accessible at runtime. This ensures that all necessary assets are bundled with the application, providing a seamless user experience.

5.6 Game Logic Flow

The overall game logic is managed by the `MainWindow` class, with the following key steps:

1. **Game Initialization**: The game is initialized by creating a grid of `FlagButton` instances. Each button is connected to the appropriate slots for handling left and right clicks.

2. **Placing Mines Randomly**: Before the game starts, mines are placed randomly on the grid using a random number generator, ensuring no two mines occupy the same cell.

3. **Adjusting Grid Sizes**: The grid layout is adjusted dynamically based on the desired number of rows, columns, and mines, allowing for varying game difficulties.

4. **Cell Click Handling**: When a cell is left-clicked, the `handleCellClick()` method checks if the cell contains a mine. If a mine is revealed, the game ends with a loss. If the cell is safe, it is revealed, and the game state is updated. The method also handles recursive revealing of adjacent cells with no mines.

5. **Using Revealed Information for Hints**: The game uses revealed cell information to calculate the probabilities of adjacent unrevealed cells containing mines. The `findHint()` method identifies cells that are guaranteed to be safe, and the `showHint()` method highlights the safest cell for the player to reveal next.

6. **Flagging Cells**: Right-clicking a cell toggles its flag state. The `handleCellRightClick()` method updates the cell's appearance and prevents it from being revealed until the flag is removed.

7. **Score and Status Updates**: The player's score is updated based on the number of revealed cells. The `updateScore()` method updates the displayed score. The `checkGameStatus()` method checks if the game has ended in a win or loss and displays the appropriate message.

8. **End Game Conditions**: The game ends when the player reveals a mine or successfully reveals all non-mine cells. A popup message informs the player of the result, and the restart button allows for a new game session.

6 Improvements and Extensions

While the project is functionally robust, there are areas where it could be visually and functionally enhanced. The user interface could be made more appealing with better graphics and animations. Implementing a modern and visually appealing design would significantly improve the user experience.

6.1 Additional Features

Several features could be added to enhance the gameplay experience:

- **Timer**: Introducing a timer to track how long it takes for a player to complete a game could add an extra layer of challenge and competition.
- **Multiplayer Mode**: Adding a multiplayer mode where players can compete against each other to solve the Minesweeper grid the fastest would enhance the game's social and competitive aspects.
- **Variable Grid Sizes and Level System**: Providing options for different grid sizes and difficulty levels (e.g., easy, medium, hard) would cater to a wider range of players, from beginners to experts. Also, implementing a level system where players can progress through increasingly difficult levels would add depth to the gameplay.
- **Enhanced Hint System**: Improving the hint system to provide more intelligent hints based on the game state could help players when they are stuck and enhance their strategic thinking.

6.2 Weak Points

Despite its strengths, the project has some areas that could be improved. The current design is functional but lacks aesthetic appeal. Investing time in enhancing the visual design and user interface would make the game more attractive to players. Additionally, while the core gameplay mechanics are solid, adding more features and customizability would significantly enhance the overall gaming experience.

By addressing these areas and implementing the suggested features, the Minesweeper game can be transformed into a more engaging and visually appealing application that offers a richer and more varied gameplay experience.

7 Difficulties Encountered

7.1 Technical Challenges

- **Qt Installation:** Installing and configuring Qt was challenging.
- **Bug Fixing:** Debugging issues related to signal-slot connections and UI adjustments required significant effort.
- **New Technology:** Learning and using Qt and C++ for OOP was a steep learning curve.

7.2 Implementation Challenges

- **Hint Mechanism:** Implementing the hint mechanism to accurately determine safe cells was difficult. I worked hard to figure out an algorithm to find a safe cell for game to reveal hints, and implementation of that algorithm was also challenging.

- **UI Adjustments:** Ensuring proper alignment and resizing of cells and buttons for a good user experience was difficult since this project is my first experience designing a game by using QT.
- **Widget Handling and Overall QT Experience:** Handling overall game logic flow and thinking about widgets when creating the grid was a challenging, but also a very educational task for me.

8 Conclusion

The Minesweeper game was successfully implemented with all required functionalities. The game mechanics, hint functionality, and user interface were thoroughly tested to ensure a smooth and enjoyable gameplay experience. This project provided valuable experience in using Qt and C++ for systems programming and OOP.

9 Appendices

The source code is provided as an extra ZIP file because of its size.