Final Report

DSA IDEATHON

S.P.A.R.K.: Strategic Probabilistic Analysis of Real-time Minesweeper Kinetics

Presented to:

Dr. Suchetana Chakraborty

Mentor TA:

Dixit Dutt Bohra

Shubham Kumar

Team
Mine-sweeped
IIT Jodhpur

INDEX

PROBLEM STATEMENT

what we propose

The basic Minesweeper game requires players to clear a grid without detonating hidden mines by utilizing hints regarding the number of surrounding mines in each field. We propose improving the gameplay by computing and presenting the odds of mines in nearby unopened cells of the opened cells, resulting in a more data-driven approach to the game. This will gamer experience and will help a novice learn how to play the game.

DOMAIN MAPPING

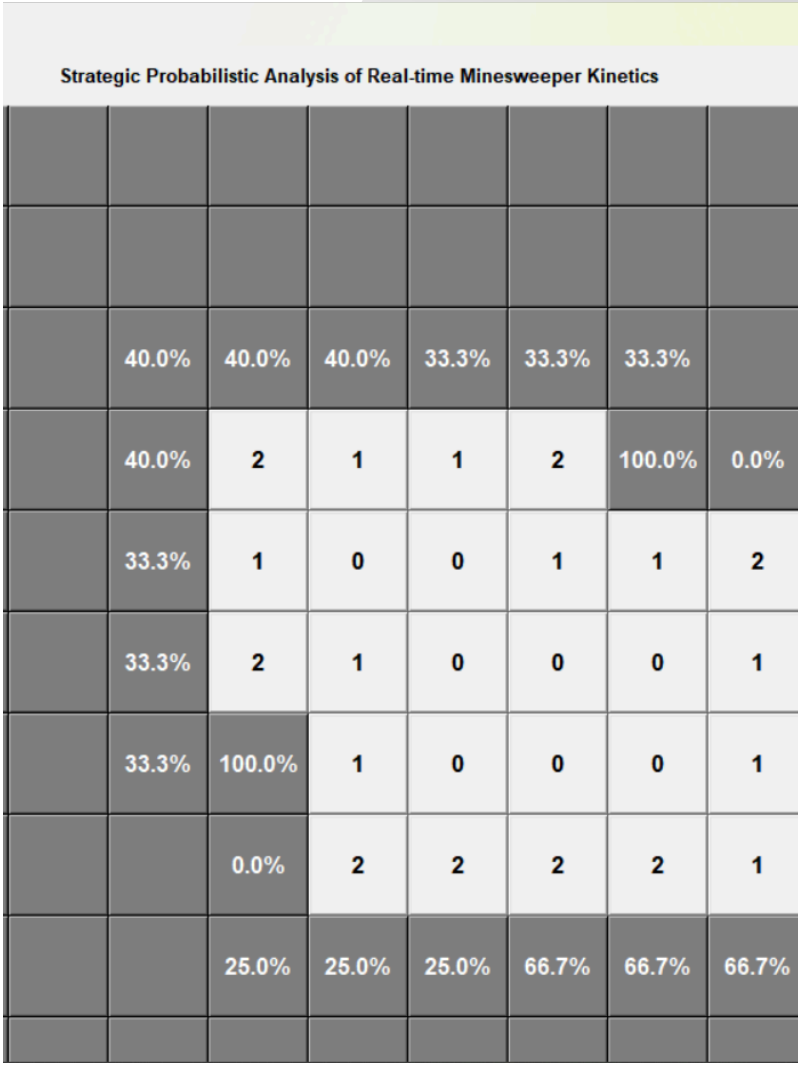
This problem falls under the domain of recreational computer games, often utilized for entertainment and cognitive skill development.

IMPORTANCE OF PROBLEM

Improves strategic decision-making in Minesweeper by providing additional probabilistic chances of mine occurence, enhancing user engagement and satisfaction.

Promise of Data-Driven Solutions

Data-driven solutions offer the potential to enhance gameplay by providing players with strategic insights based on probabilities and pattern recognition.



CHALLENGES

- Ensuring fair mine placement for balanced gameplay.
- Implementing efficient algorithms for revealing adjacent cells.
- Dynamically updating mine probability calculations for strategic decision-making.

CURRENT STATUS

what we researched

We searched the internet for various implementations of the minesweeper solver. To go about that, we first made the minesweeper game in C language. We looked at the GFG code for a minesweeper game and removed redundancies from the code. We implemented the logic for not keeping the mine at the first position and its 8 neighboring positions in this. PFA(click on it)

Existing Solutions

We searched a lot for the open source codes of the minesweeper solver. There were several sites and videos that claimed that their solution was closest to the optimal solution. But, minesweeper does not have an exact alogrithm because the problem itself is NP-complete, meaning that finding an optimal solution may require exponential time in the worst case. As a result, most Minesweeper solvers rely on heuristic approaches, which prioritize efficiency over optimality. The solutions we found on the internet were different, had no proper use of data structure or algorithm. Most were poorly written.

Let's explore the best possible solution that we found on YouTube: Github

We found that the code was too large.

Data Structure for Grid Representation

The code uses a two-dimensional array to represent the Minesweeper grid. While this is a common approach, it can be inefficient for larger grids as it requires O(n^2) space complexity, where n is the size of the grid. For very large grids, this can lead to memory issues.

Edge Detection and Counting Algorithm

The algorithm for detecting and counting edges relies on nested loops to iterate over each cell in the grid. While this approach works, it has a time complexity of O(n^2), where n is the size of the grid. This can be inefficient for very large grids.

Probability Calculation Approach

The code uses a recursive approach to generate all possible mine arrangements for open edges and then calculates probabilities based on these arrangements. While this approach is theoretically sound, it can be computationally expensive for grids with many open edges

IDEA

what we propose

Now, we understood that while the 1400+ lines of code in the repository attached in the previous page gave very good results, it was slow for larger game boards. It was not reliable and we had to click on “get probabilities” every time and hence the gamer experience was not intuitive.

Our Pitch

Data Structure for Grid Representation

We will implement a new data structure, namely, “Cell”.

The `Cell` data structure provides a structured and organized approach to represent individual cells within the Minesweeper game board. By encapsulating cell-related data and functionality within a single class, such as the cell's status (open or covered), neighboring cells, probabilities, and mine presence, the code achieves better modularity, readability, and efficiency. This allows for more straightforward management and manipulation of individual cells, facilitates optimized algorithms for tasks like mine placement, probability calculation, and cell revealing, and ultimately enhances the overall performance and maintainability of the Minesweeper game implementation.

Edge Detection and Counting Algorithm

In our data structure, we have the provision to store the neighbor cells. Hence, we don’t have to find it again and again during probability calculations and hence it saves runtime. Also, we have provision for open status of mine. So, if the an unopened box has a box next to it with open status=1, it will will be part of the edge and this way, it can be used in the calculation of outer probabilities of the code.

Probability Calculation Approach

Basically what we propose is that, we find all the unopened cells, next to an opened cell. This is the outer loop. For each element in the outer loop, we find all the neighboring opened cells. In the list of those neighboring cells, we assign max probability that each opened cell provides to the unopened cell. The probability is calculated as no. of mines surrounding the opened box (or mine count of the opened cell) divided by the no. of unopened cells linked to the opened box. However, we have to give provision for reducing probabilities of surrounding cells if a confirmed mined is observed. For that we implement a min function and impose it on the cell in the mine-board. Further, to remove bugs, we have implemented functions to impose 0% on cells that are unopened, next to an opened 0 and has more than 0% probability.

INTRODUCTION

to S.P.A.R.K.

Minesweeper: S.P.A.R.K (Strategic Probabilistic Analysis of Real-time Minesweeper Kinetics) is our Python implementation of the classic Minesweeper game with added strategic and probabilistic analysis features. This report outlines the development journey, addressing initial challenges and the transition to Python for ease of development.

Initial Challenges

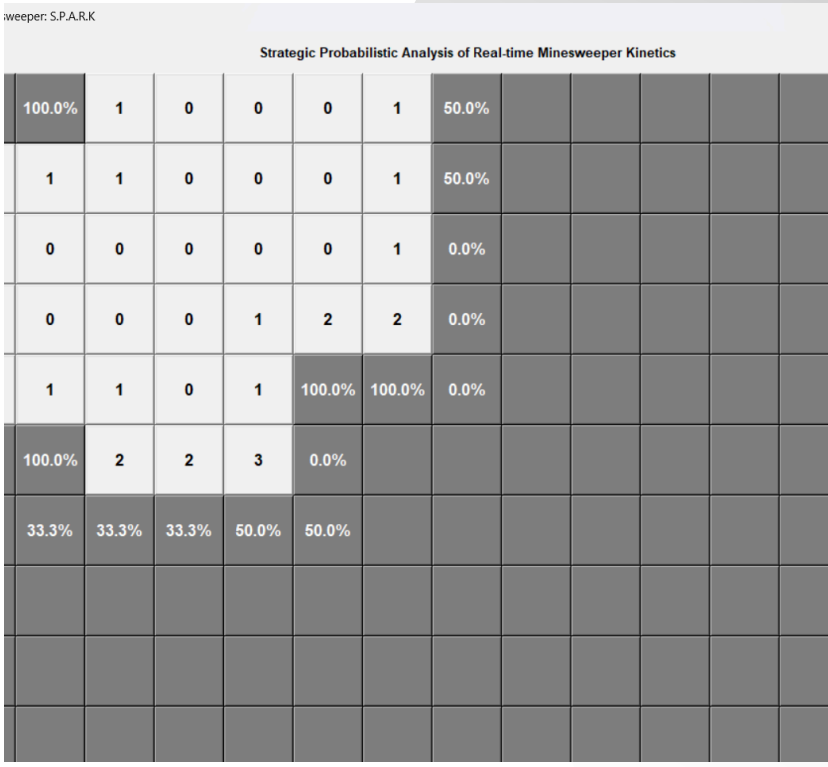
In the initial phases of development, our focus was on creating the Minesweeper game. We encountered several challenges and bugs, one of which was mine placement. Originally, we randomly placed mines on the board. However, if the player randomly selected a mine on their first move, it resulted in an unfair dis-advantage.

```
Enter the size of the board (m): 10
Enter the number of mines: 10
Initial State of Board (Mines are denoted by '*
    0  1  2  3  4  5  6  7  8  9
0   -  -  -  -  -  -  -  -  -  -
1   -  -  -  -  -  -  -  -  -  -
2   -  -  -  -  -  -  -  -  -  -
3   -  -  -  -  -  -  -  -  -  -
4   -  -  -  -  -  -  -  -  -  -
5   -  -  -  -  -  -  -  -  -  -
6   -  -  -  -  -  -  -  -  -  -
7   -  -  -  -  -  -  -  -  -  -
8   -  -  -  -  -  -  -  -  -  -
9   -  -  -  -  -  -  -  -  -  -
Enter your first move (row, column): 4 6
Current Status of Board :
    0  1  2  3  4  5  6  7  8  9
0   -  -  -  -  -  -  -  -  -  -
1   -  -  -  -  -  -  -  -  -  -
2   -  -  -  -  -  -  -  -  -  -
3   -  -  -  -  -  1  1  2  -  -
4   -  -  -  -  -  0  0  1  -  -
5   -  -  -  -  -  0  0  1  -  -
6   -  -  -  -  -  -  -  -  -  -
7   -  -  -  -  -  -  -  -  -  -
8   -  -  -  -  -  -  -  -  -  -
9   -  -  -  -  -  -  -  -  -  -
Enter your move (row, column) ->
```

To address this, we brainstormed and implemented a solution. We modified the mine placement logic to ensure that there were no mines on the first cell clicked by the player and its surrounding eight cells. This adjustment aimed to provide players with a fair chance, removing the luck factor associated with initial mine placement. Our development initially took place in C, where we encountered these challenges.

Transition to Python

The transition to Python was driven by its ease of GUI development and accessibility. Leveraging Python's libraries, particularly tkinter, allowed for seamless integration of graphical elements into the Minesweeper game. Additionally, Python's flexibility, readability and general simplicity of doing business was very helpful to our endeavors.

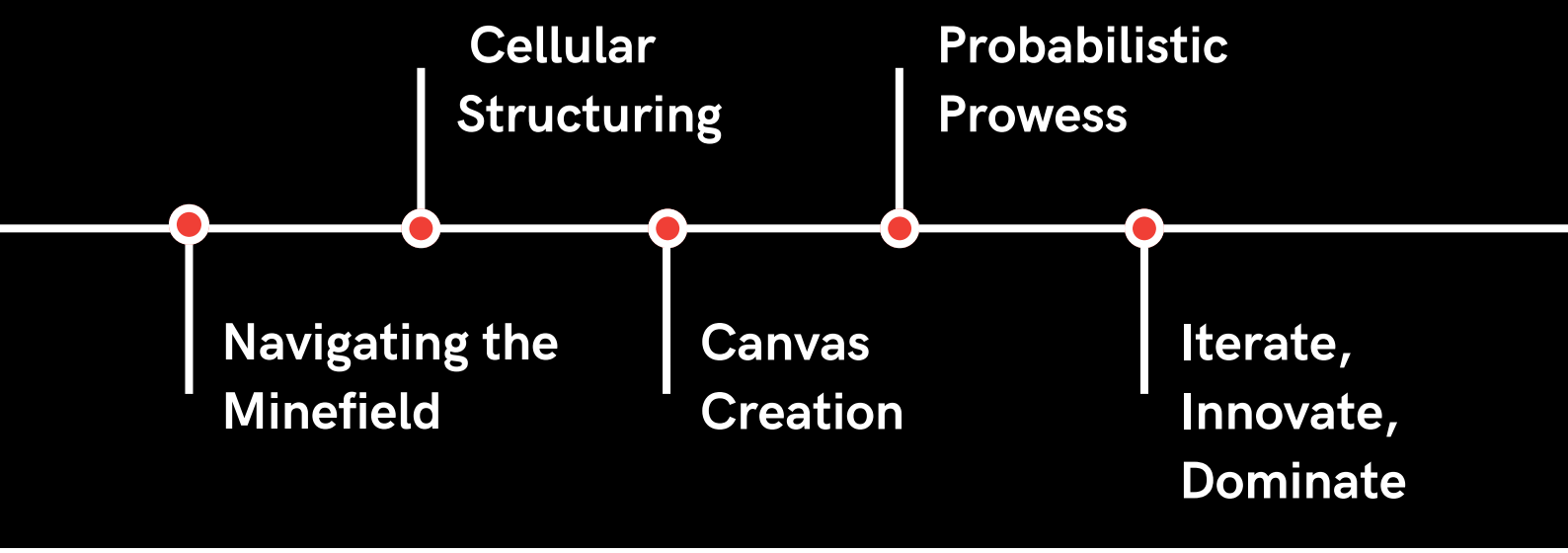


The codebase utilizes Python's object-oriented approach to represent individual cells on the Minesweeper grid- “Cells”, which has various parameters like open status, neighbour minds,etc. It dynamically generates the game board, ensuring fair mine placement by avoiding mines on the first clicked cell and its surrounding eight cells. We then employed our own algorithm, which is explained later, to calculate the likelihood of mines in unopened cells, based on its neighboring opened cells, with probabilities displayed dynamically during gameplay.

Throughout gameplay, the probabilities of mines in unopened cells are continuously updated and displayed on the game screen. This dynamic updating process encompasses all adjacent cells to the opened cells, providing players with real-time insights into potential mine locations.

By continuously updating probabilities and performing strategic analysis, the algorithm aids players in decision-making throughout the game. Players can use this information to make informed choices about which cells to uncover, reducing the risk of clicking on a mine. This strategic analysis persists until the game concludes, either by triggering a game over condition, such as clicking on a mine, or successfully uncovering all cells on the board.

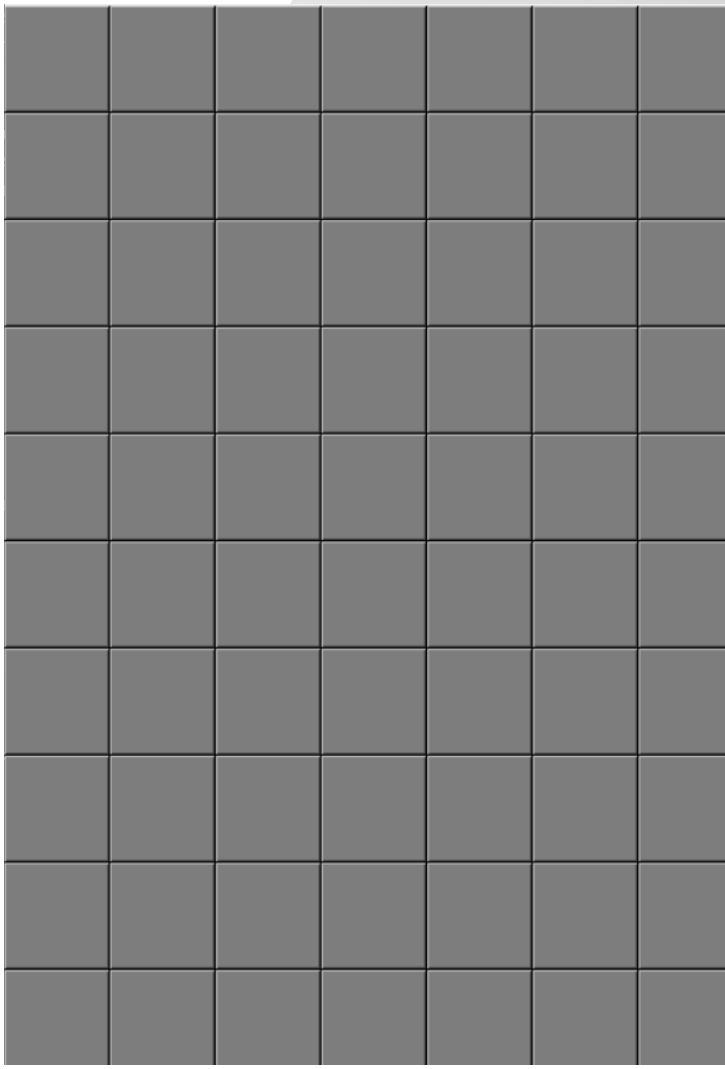
Operational RoadMap



- **Navigating the Minefield:** Overcome early challenges by ensuring fair mine placement and introducing dynamic gameplay features to maintain engagement and excitement.
- **Cellular Structuring:** Develop a comprehensive blueprint for each cell, defining unique attributes and functionalities to facilitate efficient interaction within the game environment.
- **Canvas Creation:** Craft a dynamic game board interface, meticulously painting each cell to create a level playing field accessible to all players, regardless of skill level or experience.
- **Probabilistic Prowess:** Engineer an advanced algorithm to calculate mine probabilities with precision, guiding players through the complexities of uncertainty and strategic decision-making.
- **Iterate, Innovate, Dominate:** Embrace a cycle of continuous refinement and rigorous testing, striving for excellence with each iteration to empower players on their journey to Minesweeper mastery.

DATA STRUCTURE

what we used



In this Minesweeper implementation, a key aspect of the design is the data structure used to represent the game board and individual cells. Let's explore this in-depth:

Grid of cells

The fundamental data structure in this Minesweeper game is a grid of cells, where each cell represents a single unit on the game board. The grid is implemented as a two-dimensional array, with each element representing a cell object.

Cell Object:

Each cell object encapsulates various attributes and functionalities, allowing it to store information about its state and interact with neighboring cells. Here are the key attributes of a cell object:

- **Open Status:** Indicates if the cell has been revealed or not.
- **Probability Array:** Stores probabilities of containing a mine, updated dynamically.
- **Neighbour Blocks:** Contains references to neighboring cells for game mechanics.
- **Row and Column Indices:** Positions the cell in the grid.
- **Inner Run Probability:** Probability of the cell being a mine in a specific scenario.
- **Reduced Probability:** Minimum probability of being a mine based on neighbors' probabilities.

INTERACTION AND ANALYSIS

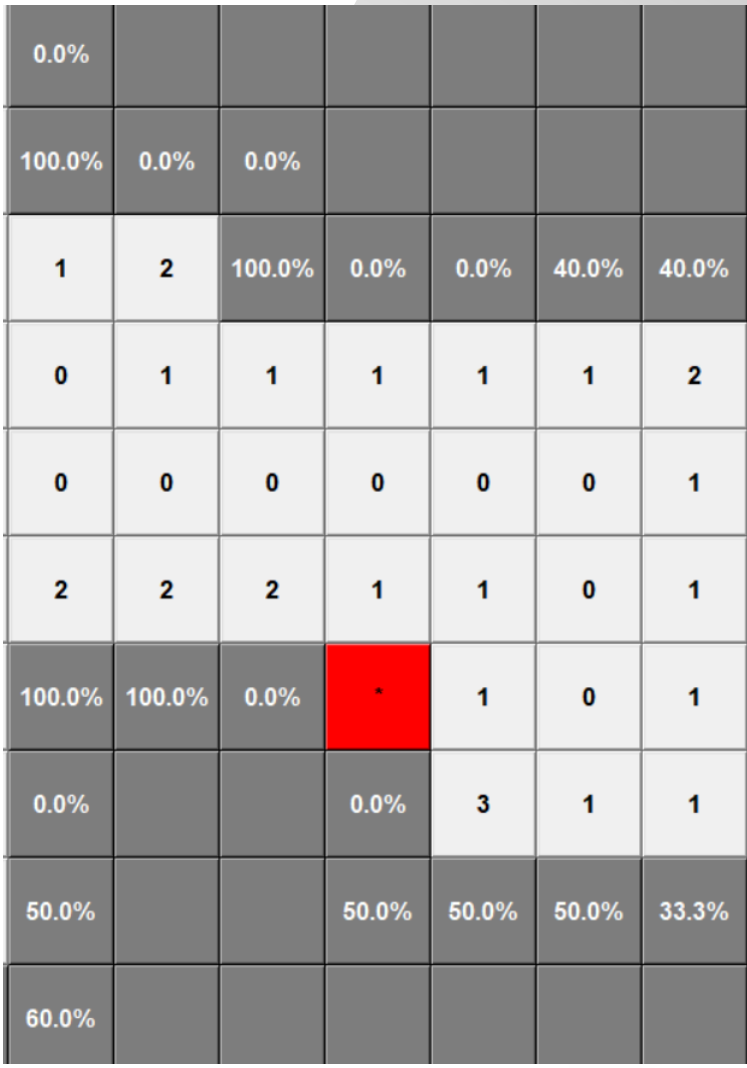
The data structure facilitates various interactions and analyses during gameplay:

Cell Reveal:

When players click on a cell, it is revealed, updating its open status and displaying its content. This interaction allows players to uncover the game board and progress through the game.

Mine Placement:

The grid structure ensures fair mine placement by strategically distributing mines across the game board. Mines are deliberately avoided near the first clicked cell and its surrounding cells, ensuring a balanced and challenging gameplay experience.



Probability Analysis:

Leveraging the grid of cells, the game dynamically calculates and updates probabilities for unopened cells based on neighboring opened cells. This probabilistic analysis provides players with strategic insights, enabling them to make informed decisions about which cells to uncover next. This algorithm we created is explained next

FUNCTION ANALYSIS

what we implemented

Each function in our Minesweeper implementation was developed with meticulous attention to detail and a deep understanding of player engagement and strategic gameplay. Let's explore that together:

__init__(self, master):

Initializes the Minesweeper game window and parameters. It sets up the game window and defines parameters like grid dimensions and mine count. Our approach ensured a welcoming entrance for players, establishing a polished interface to begin their Minesweeper journey.

create_game_board(self)

Generates the game board GUI by creating buttons representing cells and binding click events for reveal functionality. Our approach focused on creating an intuitive and visually appealing game board, prioritizing both functionality and aesthetics.

initialize_game(self, first_selected_cell):

Sets up the game board after the first cell is selected, defining neighbor blocks for each cell and ensuring fair mine placement. We aimed for a balanced starting point, allowing players to engage with the game without feeling unfairly disadvantaged.

place_mines(self, x, y):

Places mines on the game board by randomly selecting mine locations, avoiding placing mines near the first clicked cell. Our approach aimed for strategic mine placement, creating a challenging yet fair game board layout.

reveal_cell(self, x, y)

Reveals the content of a cell upon user click, handling cell reveal logic and updating the GUI. We focused on providing a seamless and engaging gameplay experience, ensuring that cell reveals were intuitive and responsive.

FUNCTION ANALYSIS

what we implemented

update_probabilities(self):

Updates probabilities of mines in unopened cells based on neighboring opened cells, enhancing strategic decision-making. Our approach involved implementing an algorithm that analyzed the game board dynamically, providing players with valuable insights into potential mine locations.

reveal_all_mines(self):

Reveals all mines on the game board, displaying all mine locations when the game concludes. We prioritized transparency and closure, ensuring that players had a clear understanding of their final game state.

remove_zero_mines(self):

Handles the removal of cells with zero mine probabilities, optimizing the game board display. We aimed for clarity and readability, ensuring that players could navigate the game board with ease.

count_unopened_adjacent(self, x, y):

Counts unopened cells adjacent to a given cell, aiding in gameplay analysis. Our approach involved providing players with valuable information to inform their strategic decisions, without overwhelming them with unnecessary detail.

ALGORITHM OVERVIEW

reveal_cell

Initialization Check

The function begins by checking if the list of mines is empty. If it is empty, indicating the first move of the game, it initializes the game by calling the initialize_game function with the coordinates of the first clicked cell.

Mine Detection

If the clicked cell contains a mine (as indicated by its coordinates being present in the list of mines), the function updates the appearance of the button to display a mine symbol and shows a "Game Over" message box. It then reveals all mines on the game board and waits for 2 seconds before destroying the game window.

Non-Mine Cell

If the clicked cell does not contain a mine, the function calculates the number of neighboring mines using the count_mines function.
The button representing the clicked cell is updated to display the number of neighboring mines, and its status is set to open. The cell's coordinates are removed from the set of covered cells.

Call to auto_clear

If the clicked cell has no neighboring mines (i.e., its count is 0), the auto_clear function is called to automatically reveal adjacent cells recursively.This is done so as to ensure all cells linked to zero are opened, and if any such cells is zero, then auto_clear is called for that cell too.

Call to update_probabilities

Finally, the update_probabilities function is called to update the probabilities of mines in unopened cells based on the newly revealed information, dynamically.

```
f reveal_cell(self, x, y):
  if not self.mines:
      self.initialize_game((x, y))

  if (x, y) in self.mines:
      self.buttons[(x, y)].config(text='*', bg='red', fg='black')
      messagebox.showinfo("Game Over", "You clicked on a mine!")
      self.reveal_all_mines()
      self.master.after(2000, self.master.destroy)  # Wait for 2000 milliseconds (2 secc

  else:
      num_mines = self.count_mines(x, y)
      self.buttons[(x, y)].config(text=str(num_mines), bg='SystemButtonFace', fg='black'
      self.covered.remove((x, y))
      self.cells[x][y].open_status = 1
      if num_mines == 0:
          self.auto_clear(x, y)
      self.update_probabilities()
```

ALGORITHM OVERVIEW

auto_clear

Cell Iteration

The auto_clear function iterates over all adjacent cells of the clicked cell, including diagonal neighbors, within the bounds of the game board.
It examines each neighboring cell to determine if it is covered and has not been revealed yet

Cell Reveal and Recursion

For each adjacent cell, if it's covered and has no neighboring mines, the function reveals it and recursively calls itself for further exploration. This process continues until all adjacent cells with no neighboring mines are revealed, efficiently clearing large contiguous areas.

Probabilistic Analysis Update

As each cell is revealed, its inner run probability is updated with a value of 0.0, indicating that no mines are present in its immediate vicinity.
The maximum probability of a mine for the revealed cell is set to 0, as there are no neighboring mines to influence its probability.

These updates contribute to the ongoing probabilistic analysis of the game board, ensuring that the probabilities of mines in unopened cells are accurately adjusted based on newly revealed information.

Efficient Board Exploration

By automatically revealing adjacent cells with no neighboring mines, the auto_clear function efficiently explores the game board, revealing large contiguous areas of empty cells with a single click. This allows the player to get efficient and more no of choices.

```
def auto_clear(self, x, y):
    for i in range(max(0, x-1), min(self.num_rows, x+2)):
        for j in range(max(0, y-1), min(self.num_columns, y+2)):
            if (i, j) in self.covered and self.buttons[(i, j)]['text'] == ' ':
                num_mines = self.count_mines(i, j)
                self.buttons[(i, j)].config(text=str(num_mines), bg='SystemButtor
                self.cells[i][j].inner_run_prob.append(0.0)
                self.cells[i][j].max_per_mine = 0
                self.covered.remove((i, j))
                self.cells[i][j].open_status = 1
                if num_mines == 0:
                    self.auto_clear(i, j)
```

ALGORITHM OVERVIEW

in update_probabilities

A more detailed overview of the "update probabilities" algorithm in the Minesweeper game:

Neighbor Identification:

The algorithm begins by identifying covered cells neighboring those that have been revealed. By iterating through each covered cell, it determines which cells are adjacent to the revealed ones.

Probability Computation:

For each covered cell adjacent to revealed cells, the algorithm computes the probability of that cell containing a mine. This calculation is based on the number of neighboring open cells and the number of mines among them.

Maximum Probability Update

Following the probability computation, the algorithm updates the maximum probability of mine presence for each covered cell. This maximum probability represents the highest likelihood of a mine being present in that cell.

Confirmed Mine Identification:

Cells with a maximum probability of 1 are identified as confirmed mines. These cells are deemed certain to contain mines based on the information available from neighboring revealed cells.

Non-Mine Probability Calculation:

For covered cells adjacent to confirmed non-mine cells, the algorithm calculates the probability of those cells not containing mines. This calculation considers the number of neighboring covered cells and the number of adjacent confirmed non-mine cells.

Probability Display Update:

Upon computing the probabilities for each covered cell, the algorithm updates the display accordingly. The calculated probabilities are shown on the corresponding buttons in the game interface. This aids players in making informed decisions about which cells to reveal next.

Probability Computation:

Throughout the process, the algorithm continuously monitors the game state. It checks if all mines have been correctly identified by the player. If the number of covered cells equals the number of mines, and all identified mines are correct, the game is won.

in update_probabilities

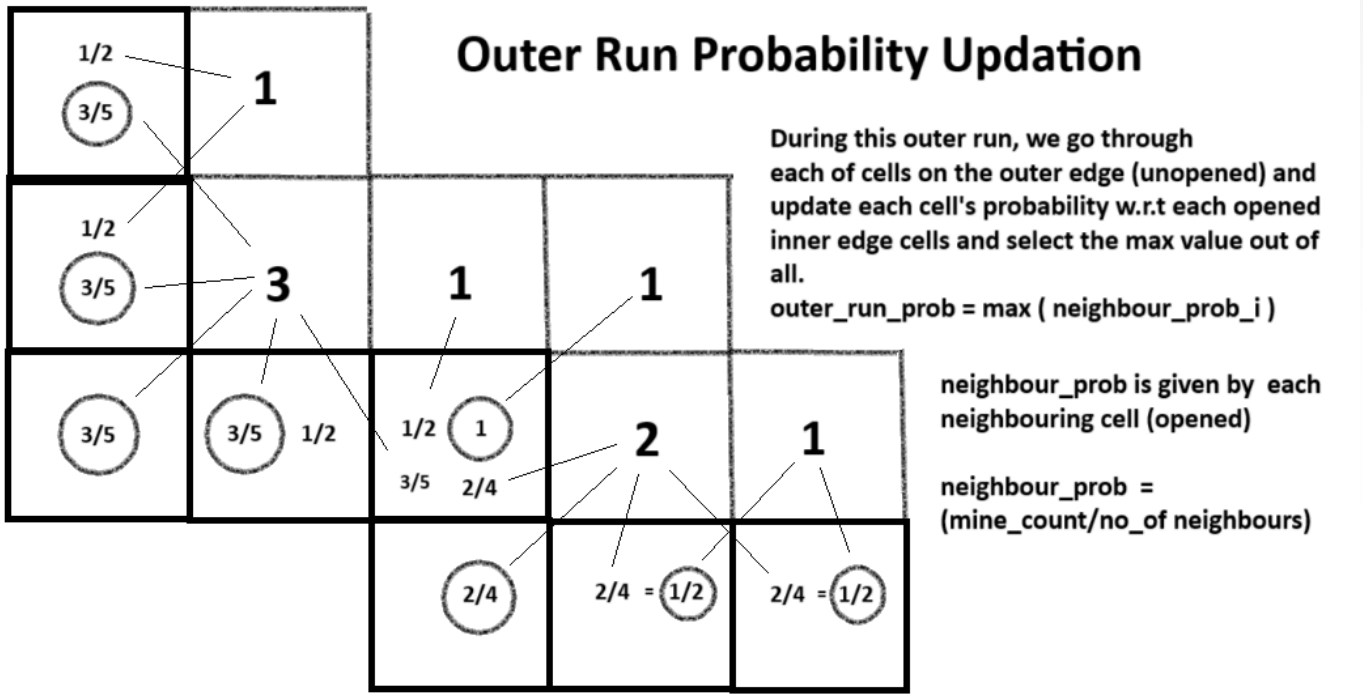
Win Condition Check:

Throughout the process, the algorithm continuously monitors the game state. It checks if all mines have been correctly identified by the player. If the number of covered cells equals the number of mines, and all identified mines are correct, the game is won.

Zero-Mine Cell Removal:

Finally, the algorithm removes covered cells with no neighboring mines from consideration. These cells have a probability of zero and are updated to reflect this information on the game board, assisting the player in focusing on areas with higher probabilities of containing mines.

Outer loop



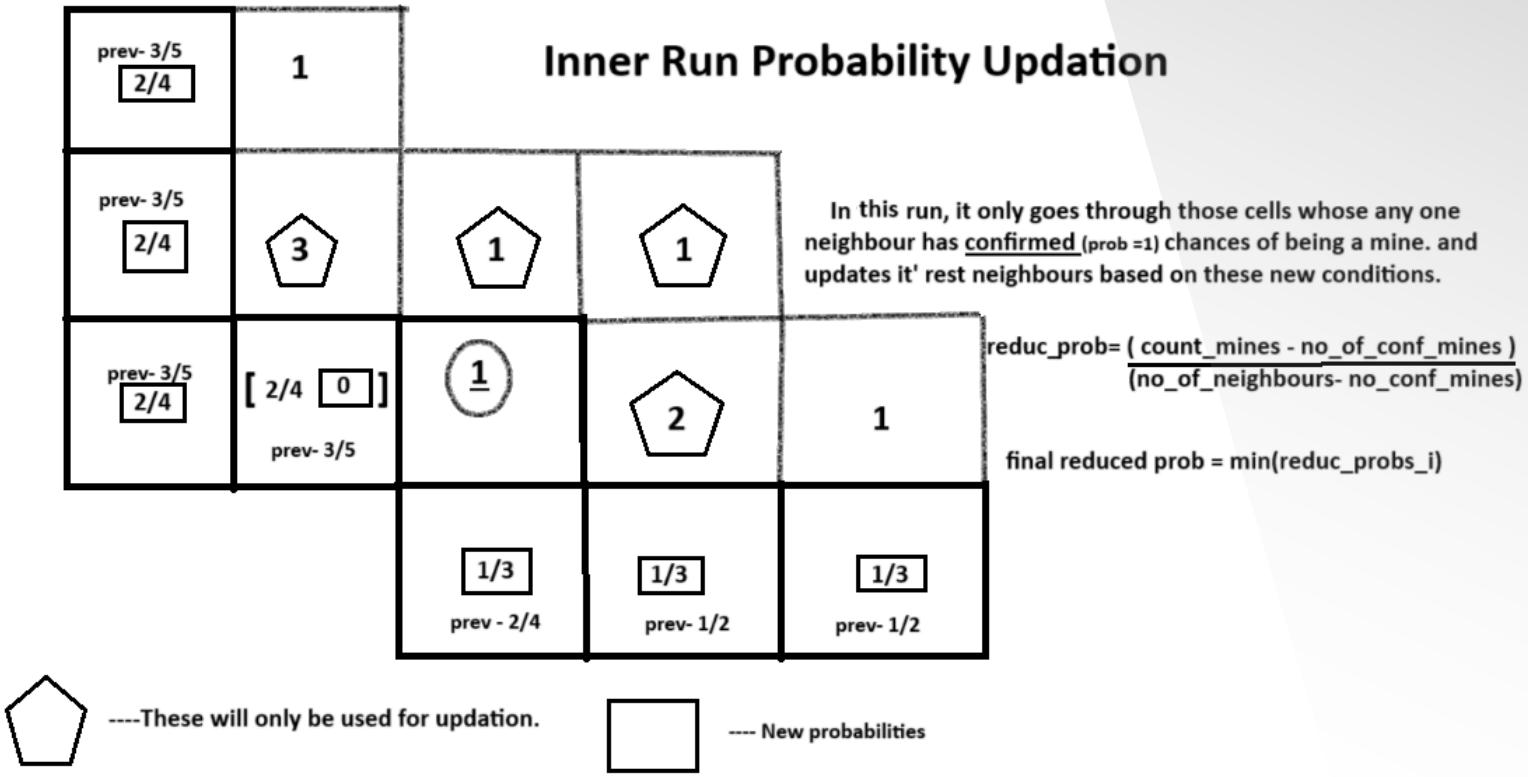
The outer run probability updation is responsible for updating the probability of each unopened cell based on the information from the opened neighboring cells. This process is illustrated in the image above.
During this outer run, the code iterates through each unopened cell on the outer edge (cells adjacent to at least one opened cell) and updates its probability based on the opened inner edge cells (neighboring opened cells).

The probability of an unopened cell is calculated as follows:
outer_run_prob = max(neighbour_prob_j)
Here, neighbour_prob_j is the probability value obtained from each neighboring opened cell, calculated as:
neighbour_prob = (mine_count / no_of_neighbours)

In other words, for each unopened cell, the code considers the probabilities provided by all its opened neighboring cells and selects the maximum value as the outer run probability for that cell.

For example, in the previous image, the unopened cell with a probability of 3/5 has three opened neighboring cells with probabilities 1/2, 3/5, and 2/4 (1/2). The maximum of these probabilities is 3/5, which becomes the outer run probability for that unopened cell.

Inner loop



The inner run probability updation is a more refined process that updates the probabilities of the remaining unopened cells based on the information from the confirmed mines and the newly updated probabilities from the outer run.

In this run, the code only considers those unopened cells whose any one neighbor has a confirmed probability of being a mine (prob = 1) and updates the rest of its neighbors based on these new conditions.
The probability for these remaining unopened cells is calculated as follows:
reduc_prob = (count_mines - no_of_conf_mines) / (no_of_neighbours - no_conf_mines)
.

Here, count_mines is the number of mines indicated by the opened neighboring cell, no_of_conf_mines is the number of confirmed mines among the unopened neighbors, and no_of_neighbours is the total number of unopened neighboring cells.
The final reduced probability for each unopened cell is calculated as:
final reduced prob = min(reduc_probs_j)
Where reduc_probs_j are the probabilities calculated for that cell from different opened neighboring cells.

n this image, the cell with a probability of 2/4 has one confirmed mine neighbor (prob = 1). The reduced probability for the remaining unopened neighbors is calculated as (2 - 1) / (4 - 1) = 1/3, which becomes the new probability for those cells. The code then iterates through all the unopened cells and assigns the minimum of these reduced probabilities as the final reduced probability for that cell.

This two-step process of outer run and inner run probability updation helps refine the probabilities of the unopened cells based on the available information, providing a more accurate representation of the likelihood of each cell containing a mine.

ALGORITHM OVERVIEW

remove_zero_mines

The remove_zero_mines function in the Minesweeper game operates by systematically analyzing covered cells on the game board to identify regions with zero mine probabilities.

Exploring Covered Cells

The function iterates through each covered cell on the game board, examining their potential for containing zero mines.

Probabilistic Analysis Update

For each covered cell, the function determines its adjacent open cells in all eight directions, including diagonals.

Identifying Inner Edges

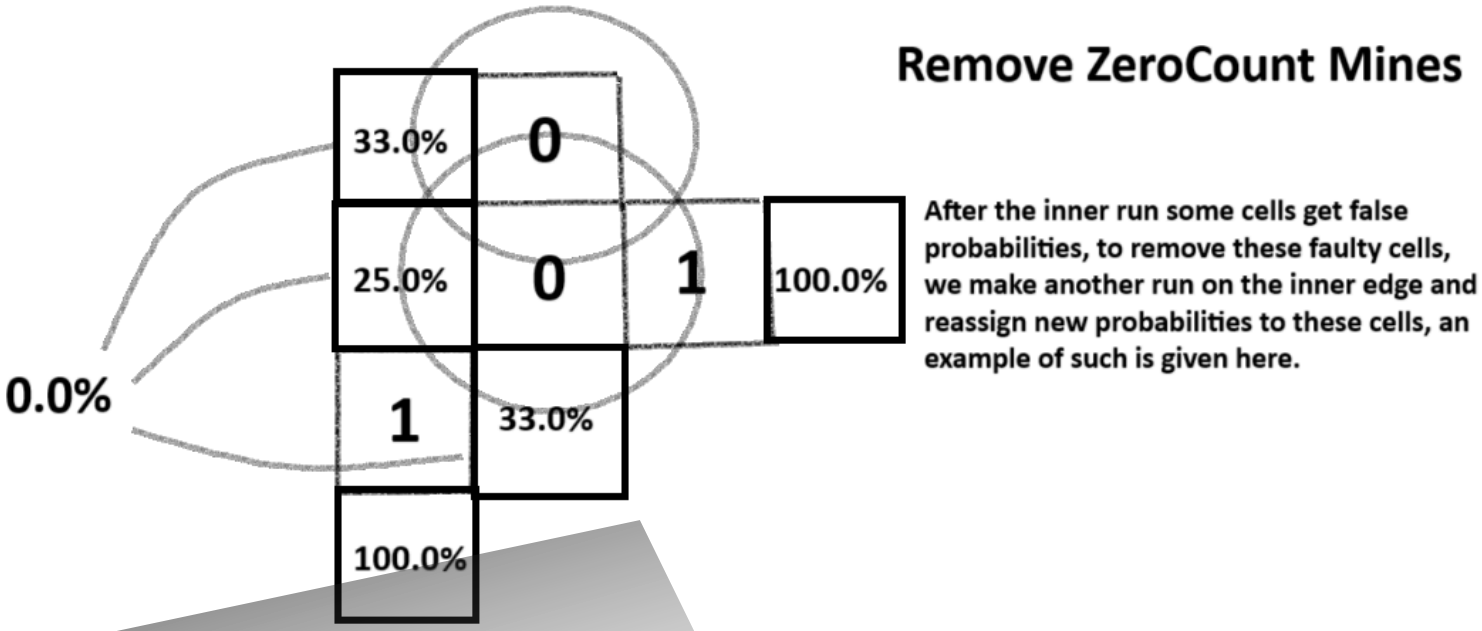
From the list of edges, the function extracts inner edges representing cells entirely surrounded by open cells. These inner edges signify regions likely to contain zero mines.

Removing Zero Mines

For each inner edge where the count of neighboring mines is zero, the function reveals the cells and updates their visual representation on the game board. It modifies the button text to display a zero mine probability, providing players with immediate visual feedback.

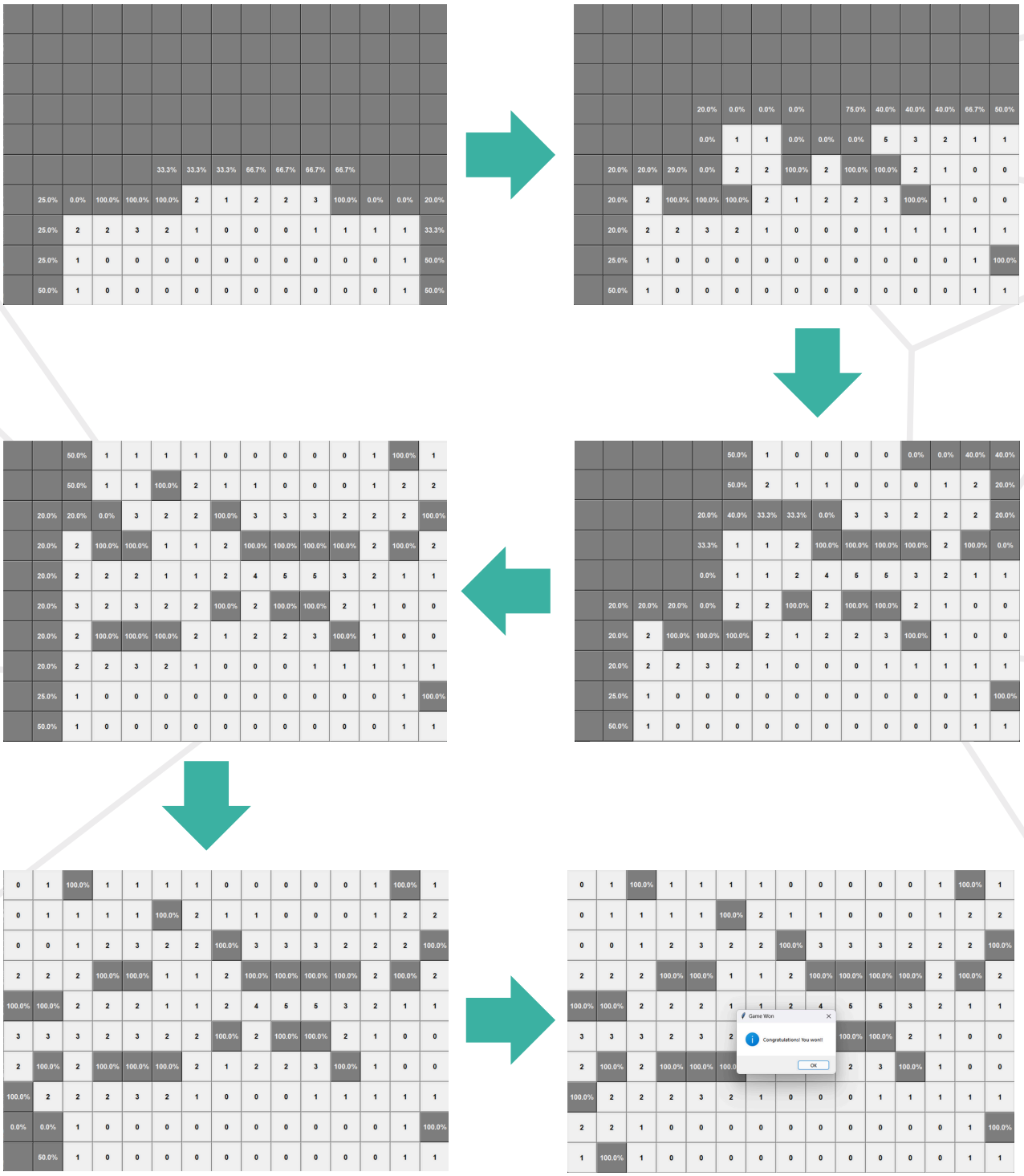
Organizing Edge Data

It organizes the collected data into a structured list of edges, where each edge corresponds to a covered cell and its neighboring open cells. This structured approach aids in efficiently identifying potential areas to be cleared.



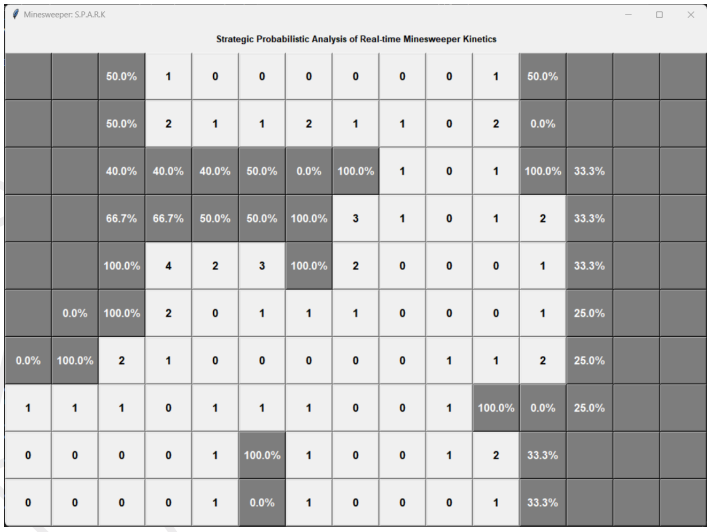
Winning The Game

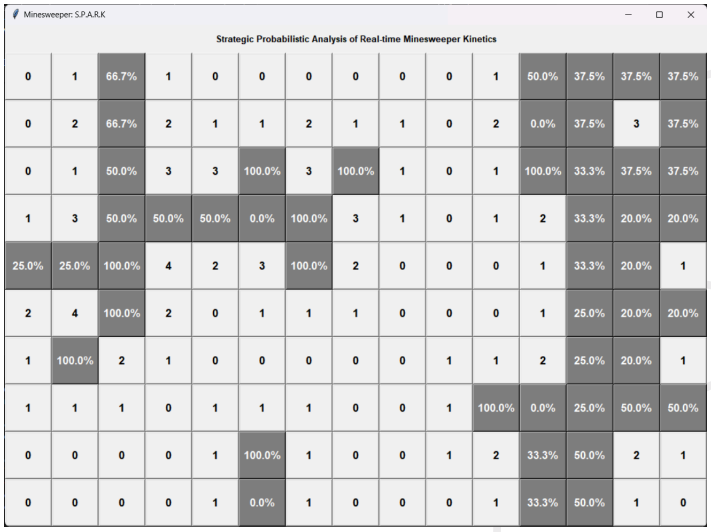
Let's look at snapshots from a Game that was won, selecting the least probabilities in each term, with dynamically updated probabilities everywhere. We can change the difficulty level in the code by increasing no. of mines, keeping no. of rows and columns constant, or any other way. Here, no. of rows = 10, no. of cols = 15 and no. of mines =25. Hence, this is a easier game.

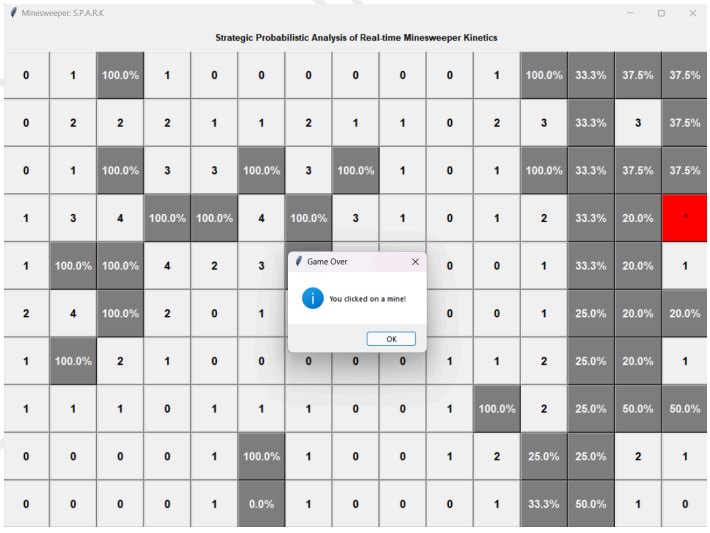


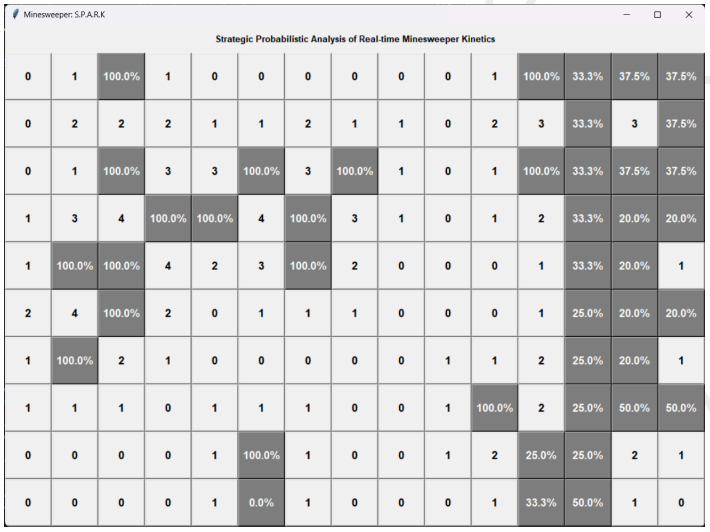
Losing The Game

Without changing any parameters, we can also lose ,if by luck, it comes down to a choice between various equal smaller probabilities, and we chose the wrong one. This happens in the real game too.



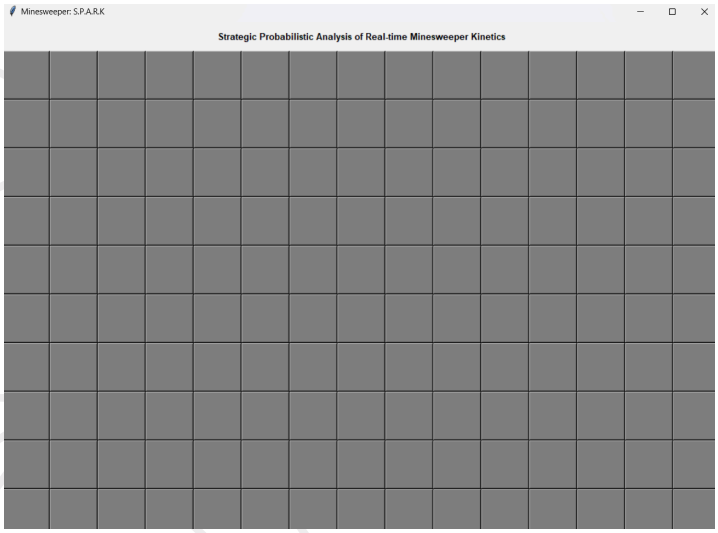


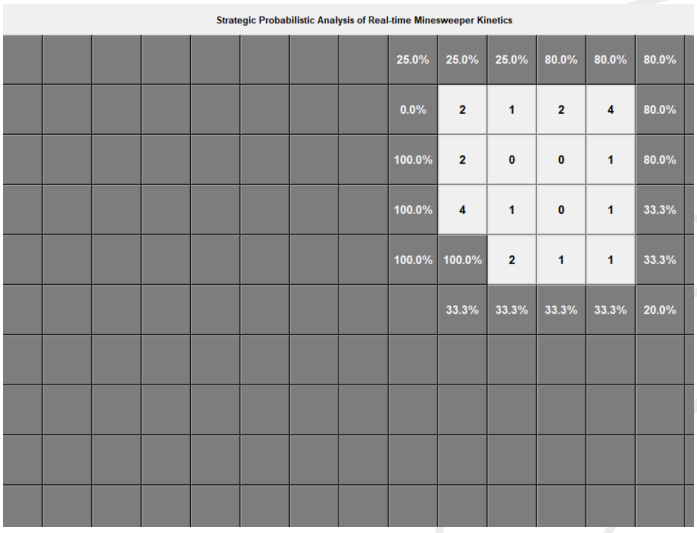


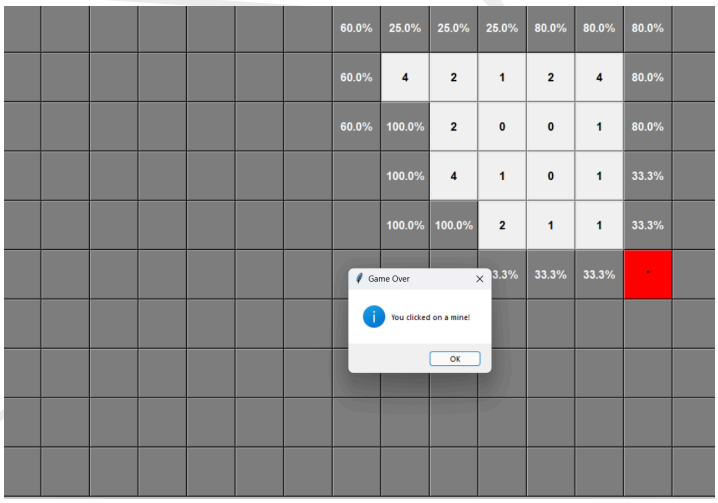


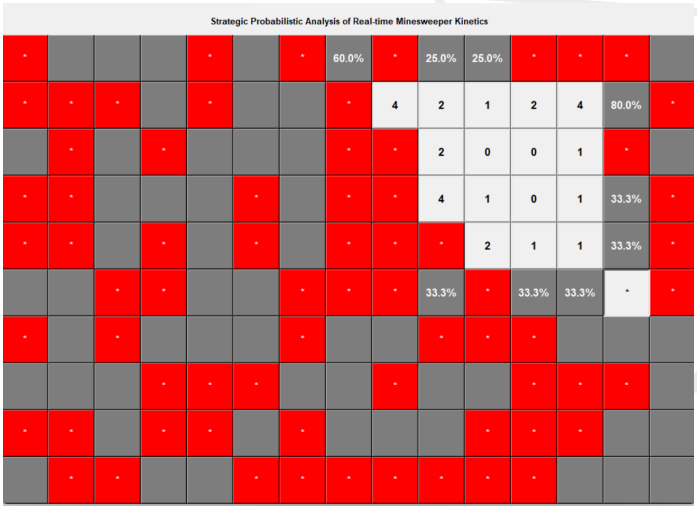
Losing The Game

Now, we have increased the no. of mines to 70 to make the game very hard. Now, choosing the correct option gets difficult. Also, the flow may get blocked by such large no of mines









DSA Report

Team Mine-sweeped

(in order of pictures)

Aditya Kumar (B22ME004)

Sonal Raj (B22ME062)

Yash Golani (B22ME072)

Sangameshwar Wanole (B22CH029)