Minesweeper Implementation

Term Project

CPTS 440

Project Team 11

Allysa Sewell

Connor Hill

Jacob Ibach

Kenan Anderson

4/23/2024

Table of Contents

I. Introduction

Minesweeper is a classic puzzle game released by Microsoft in 1990. Since then, Minesweeper has been a popular addition to multiple releases of the Windows operating system. Most people have a passing familiarity with Minesweeper, making it an excellent candidate for developing and showcasing artificial intelligence algorithms and techniques. This introduction introduces the game's mechanics and discusses the relevant AI concepts involved in developing an intelligent Minesweeper solver, including logic, probability, search algorithms, and constraint satisfaction.

A game of Minesweeper begins with a grid of covered tiles, beneath which lies a predetermined number of mines. The objective is to clear the grid without detonating any of the hidden mines. Upon starting a game, the player is presented with a grid of unmarked squares. The first click is always safe, revealing a portion of the grid and potentially some numbers. These numbers indicate how many mines are adjacent to that particular square. The player uses these numerical clues to deduce the locations of the mines.

To navigate the minefield, the player must select tiles that they believe are safe. When revealing a square that is safe, it either remains blank, shows a number, or clears a section of the grid. The strategy lies in using the numerical hints to logically determine which squares are safe to reveal. For instance, if a square is numbered with a 1, the player knows there is exactly one mine touching that square. With analysis and some luck, the player can work to reveal all non-mine squares. A successful game ends when all non-mine squares are revealed, and all mines are correctly flagged. Conversely, clicking on a square with a mine results in an immediate loss.

In solving Minesweeper puzzles, AI models heavily rely on logical reasoning, which forms the foundation for identifying safe cells or mine locations from the numbers displayed on the board. When a cell displays a '1', and one of its adjacent cells is unrevealed while the others are either revealed or flagged as mines, the AI deduces that the unrevealed cell must contain a mine. Basic logical deductions are important for clearing large portions of the board without needing more complex algorithms. However, when logical reasoning alone isn't enough to advance in the game, probability calculations become necessary. AI systems estimate the likelihood of mines in hidden cells based on available information and might take calculated risks based on which mine configurations the system is considering.

Minesweeper can also be viewed as a search problem, with the objective being to navigate through the grid while avoiding mines. Search algorithms like Depth-first search (DFS) and Breadth-first search (BFS) are used to explore possible

configurations and strategies. For more complex scenarios, advanced techniques such as backtracking help in systematically testing and eliminating potential mine placements, especially useful in ambiguous situations where there are multiple solutions to choose from.

Additionally, Minesweeper aligns well with the principles of constraint satisfaction problems (CSP), where each number in a cell imposes constraints on the count of mines in its neighboring cells. Solving the game then involves fulfilling these constraints throughout the grid. AI implementations often employ CSP-solving techniques such as backtracking, constraint propagation, and heuristic searches to reduce the search space and efficiently resolve puzzles. These approaches not only lead to effective game resolution but also boost the model's capability to handle similar problems in various contexts.

II. Goals and Desired Outcomes

Paragraph 2- Introduce project, Goal- implement program that solves the minesweeper puzzle, Describe techniques used (best-first search, Naive Bayes calculation, etc.), Desired outcome (design functional game, develop a better understanding of the above concepts and their applications)

This project aims to develop an in-depth program capable of effectively solving the Minesweeper puzzle, a game that presents both a logical and probabilistic challenge to players. The primary goal is to implement a solver using a combination of AI techniques, including best-first search, constraint satisfaction, and probabilistic agents, which are used to navigate the minefield safely. Our current implementation follows the structure and methods that are described in the following paragraphs.

The first desired outcome of this project is to design a functional Minesweeper game solver that can consistently clear mines without detonating any, demonstrating the effectiveness of the implemented AI strategies. Secondly, the project is meant to deepen our understanding of various AI concepts such as search algorithms, probability calculations, and constraint satisfaction, and their applications in real-world problem-solving scenarios. By achieving these objectives, the project will not only provide a practical application of these algorithms in game theory and logic puzzles but also contribute to broader understandings on the integration of deterministic and probabilistic methods described in this course.

III. Insights Into Implementation

This Minesweeper implementation employs notable techniques to analyze and solve the puzzle. The main components include best-first search using a priority queue and probability calculations, and constraint satisfaction. The game itself is represented as a State class that contains dictionaries storing the neighbors of each cell and the game's solution with each square's correct value. In addition, the mines dictionary shows the configuration of the board as a player would observe it. The solution list holds a record of which squares have been explored.

Initializing the game

Games are designed using the MakeBoard() function, which creates a game solution based on a board size and mine number specified by the player. Mines are placed at locations determined by two integers randomly selected with the randint() function. The number of adjacent mines for the remaining squares are then counted and added to the solution dictionary.

Selecting squares to uncover using search()

In the main search() function, a best-first search strategy was implemented using a priority queue, called frontier, to systematically explore cells with the lowest probability of containing a mine, thus minimizing the risk of triggering a mine. Elements of the frontier are also put into the node_list to allow for easy access. Once a node, or square, is selected, it is removed from the frontier and added to the list of known positions.

The update() function marks squares as mines based on the number of available squares next to the most recently explored cell. If the number of free squares is equal to the number of remaining mines needed, all the cells are assigned a probability of 1 and assumed to be mines. If a square already has the required number of adjacent mines, all unassigned neighboring cells are given a probability of 0 and assumed to be safe.

When a square is explored by calling the explore() function, all of its neighbors that have not yet been assigned a value will be either added to the frontier or given a new probability. Additionally, cells adjacent to the current node are revealed if its value is 0. This is similar to how in the game if a blank square is clicked a whole cluster of tiles is uncovered. This method ensures that the safest paths are prioritized during the search process.

Calculating probabilities

The process for probability calculation begins with the GenerateModels() function, which creates all possible scenarios, or models, for a given set of nodes. These models simulate different configurations of mines across the nodes, utilizing the Cartesian product to generate every combination of nodes being mines or empty squares. Each model is then tested for consistency using the CheckConsistency() function, ensuring that the number of mines around any cell matches the known numbers on the board, thus maintaining a consistent state.

Following model generation, the CalculateProbability() function estimates the likelihood of a tile containing a mine. It filters the models to include only those consistent with the board's current state and calculates the probability by dividing the number of models where the tile is a mine by the total number of consistent models. This approach updates the likelihood of a mine being present as new information (board states) is integrated, allowing for dynamic adjustments to our implementation strategy.

These probabilities are then used within the Explore() function, which guides the program's next moves. By prioritizing tiles with lower probabilities of containing mines, it minimizes the risk of triggering a mine. This priority-based exploration is managed through a frontier, a priority queue that ensures tiles are explored in an order that maximizes safety. This probabilistic method strives to emulate strategic thinking in Minesweeper, allowing our implementation to navigate through the minefield with a calculated approach.

Constraint satisfaction

The implementation includes a constraint satisfaction mechanism that checks the consistency of potential mine placements with the clues currently visible on the board. This is accomplished using the CheckConsistency() function, which is called whenever new models are generated. Its purpose is to ensure that these models do not contradict the numbers shown on the board. For instance, the model should not cause the number of mines adjacent to a neighboring cell to exceed its allowed value. Similarly, for each neighbor that has a specified value, the number of unknown positions minus the number of mines in the set of surrounding cells must be greater than or equal to its value to make sure there are enough available spaces for the required number of mines.

IV. Results

Our findings are consistent with our anticipated outcomes, given the scope of the project and techniques utilized. While our implementation effectively computes probabilities using our generative model, it is in the nature of Minesweeper to introduce some element of chance into the gameplay outcomes. Given situations where there are no guaranteed safe tiles, there lies an inherent risk of encountering a mine even when selecting tiles with the lowest probabilities and therefore ending a game in a failure. Our analysis is based on 50 game iterations on our standard 10 by 10 board with a 10% mine density. Tests were completed in a locally run VSCode environment to diligently avoid as much network interference in the time constraint as possible.

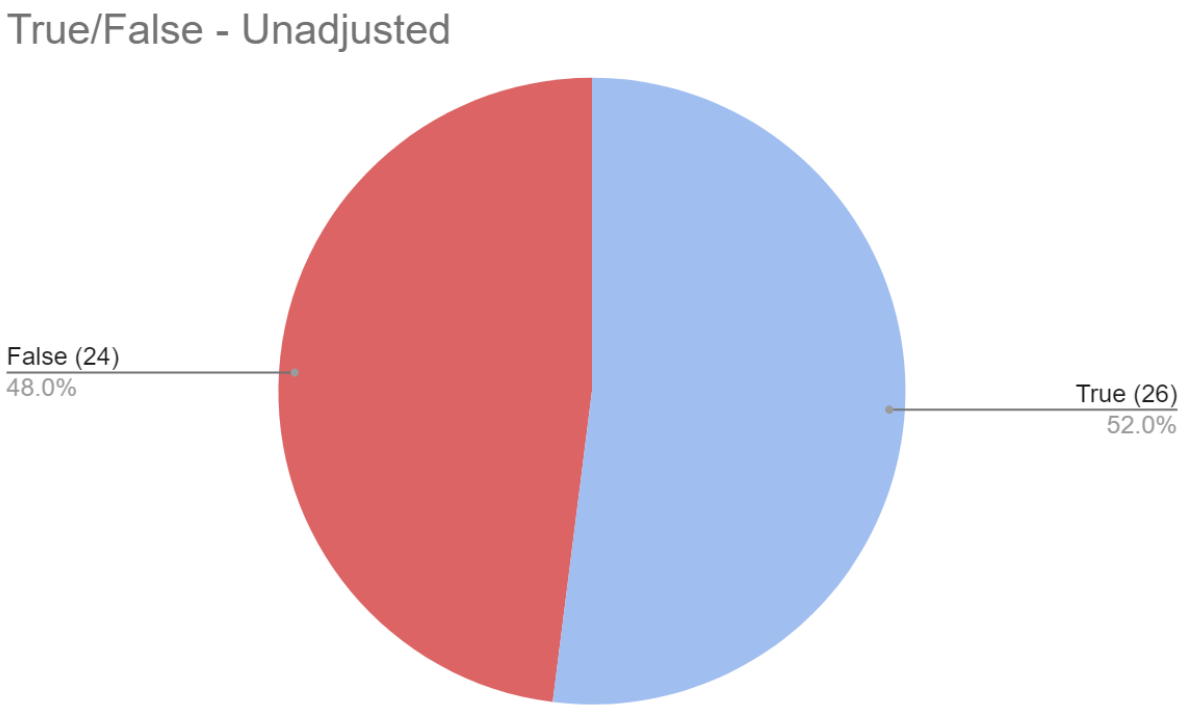

Figure 1.1

Figure 1.1 illustrates our initial success rate for our 50 iterations at just over 50% for determining all tiles on the board without striking a mine. The ratio at a glance is unsatisfactory, but this reflects the first constraint in our implementation. An upper bound of 60 seconds, or 1 minute, was set to automatically return a False result as to speed up the testing process. The following figure reflects an updated ratio.

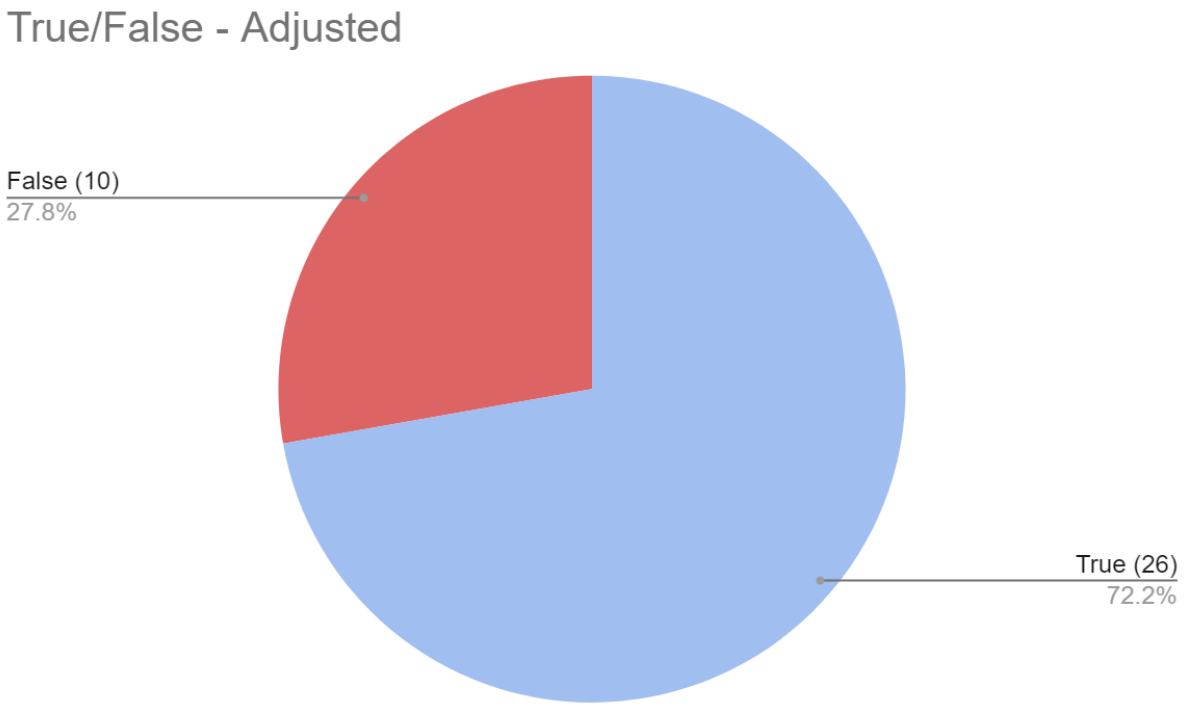

Figure 1.2

Figure 1.2 removes all False results with a time longer than the 60 second threshold. This is an acceptable adjustment, as trials that took longer than this threshold were generating proportionately more models and therefore made it impossible to assume a True or False result. It could reasonably be assumed that given our generative-model based implementation, trials that take longer periods of time would be more likely to return successes due to more models requiring generation. Simply put, we can remove any results that are out of the scope of work we set for our testing environment. There were 14 results that returned False due entirely to the upper-bound time constraint. Out of the completed algorithmic iterations, we can see 26 True and 10 False results. This adjustment moves the successful ratio from 52.0% to an acceptable 72.2%. There is a direct correlation between models generated and for how long it takes for our implementation to return a result. The number of models depends entirely on the randomly generated board environment.

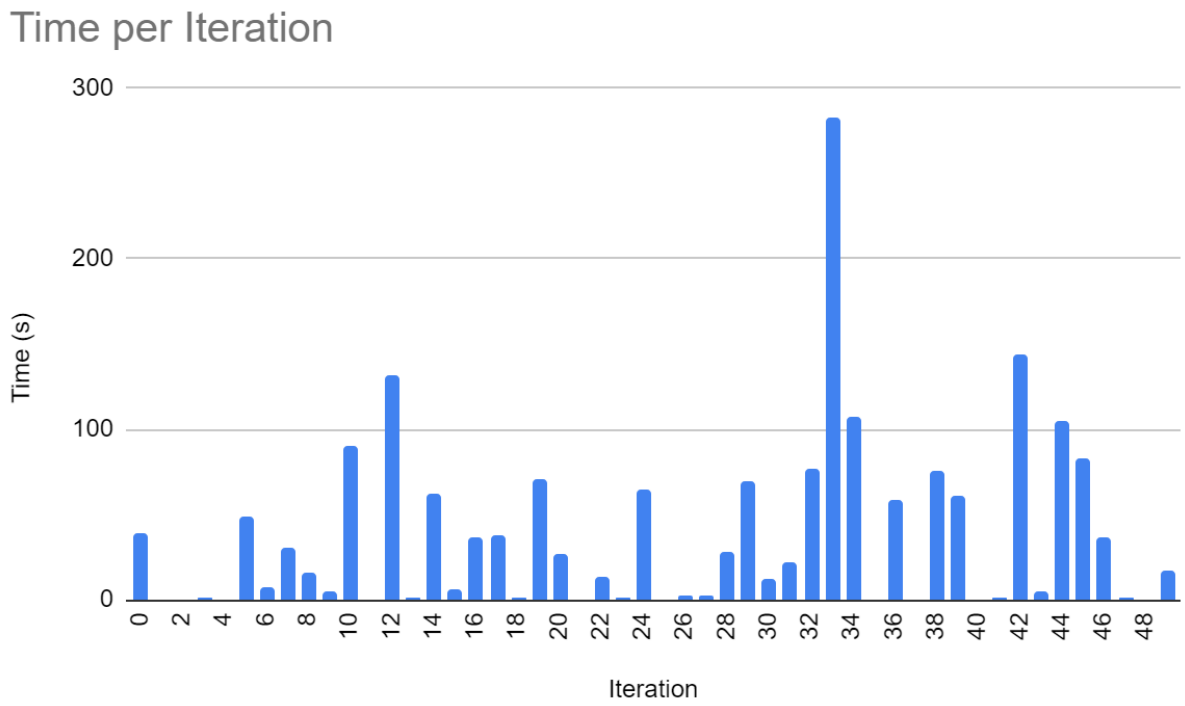

Figure 2.1

Figure 2.1 reflects the temporal results for each iteration, revealing a relatively incohesive average. The fluctuations are reasonable, likely indicative of the randomly generated board environment, the number of generated models for each iteration, and premature game terminations due to mine encounters.

Again, at first glance, the average time for all iterations was 37.74 seconds, with True results requiring a shorter computation time of 15.58 seconds compared to the False outcomes at 61.76 seconds. To stay consistent with the data presented graphically, we will remove the results that took longer than our time threshold. The new average drops to 12.89 seconds, the True results average is 15.58 seconds, and the False results average is 5.89 seconds. These adjusted times appear to be consistent with our algorithmic implementation, as a False result would imply a mine tile was flipped and ended the game prematurely, while a True result would require a larger number of models to be generated.

V. Conclusion

In conclusion, our project has successfully demonstrated the application of artificial intelligence techniques to solve the Minesweeper puzzle. Through the implementation of best-first search, constraint satisfaction, and probability calculations, we have developed a program that not only solves the game with high efficiency but also serves as a practical example of these concepts in action.

Our results, based on 50 game iterations, show a success rate exceeding 72% highlighting the effectiveness of the AI strategies employed. The average time of 12.89 is relatively quick and definitely acceptable given the number of calculations done.

Optimizations in regards to either the number of models generated or the size of generated models would be a next step in developing our implementation. The model generations directly reflect the temporal calculations, and this would help the timeliness of the overall function. Further work could focus on improving the heuristics for assigning probabilities to allow for quicker and more accurate calculations. This would increase the frequency with which the puzzle is solved correctly, increasing our success ratio. The code could also be adjusted to allow users to specify different board sizes and mine numbers.

Overall, the implementation has led to a better understanding of core concepts in AI, such as search, constraint satisfaction, and probabilistic agents. The project illustrates how these techniques can be combined to effectively solve a problem.