Minsweeper AI

CPSC 481-11

Jacob Nguyen
Engineering & Computer Science
California State University, Fullerton
Fullerton, CA, USA
jacobnguyen714@csu.fullerton.edu

Wayne Muse
Engineering & Computer Science
California State University, Fullerton
Fullerton, CA, USA
waynemuse@csu.fullerton.edu

***Abstract*— Jerry the minesweeper AI is an autonomous agent that uses first order predicate logic implemented through Z3 theorem solver to identify safe moves, detect mines, and navigate the minesweeper environment when the result is uncertain and solve a game of minesweeper accurately and quickly.**

***Keywords*— AI, Python, Z3, Pygame, Predicate Logic, Minesweeper**

I. Introduction

Minesweeper is a long-standing single player logic puzzle game introduced in early windows subsystems [1]. The player must deduce the locations of all the mines from incomplete numerical clues without triggering a single mine in order to revel the entire board to win. Despite its simple premise, this game poses a challenging inference problem, as certain board states require strict logical deduction and other board states require a guess. This project develops and AI called Jerry the Minesweeper AI, that applies first order predicate logic through the python Z3 theorem solver to model each as a Boolean variable.

II. Problem Statement

The problem addressed in this project is the design and implementation of an autonomous ai agent capable of playing and accurately solving the game Minesweeper by identifying safe cells, detecting unsafe cells, and making the best decision in situations involving uncertainty.

III. Our Approach

A. Creating Our Software

Our initial approach to creating a Minesweeper AI involved developing a vision-based system in which the agent would capture images of the game screen, use template matching to recognize tiles, and convert the visual information into an internal array-based representation of the board. Once the board was constructed into a format the AI can use, the AI solves the minesweeper puzzle using predicate logic [2]. This approach would allow the user to have a minesweeper AI that could use any minesweeper runner, and could dynamically run at any size game. However, such an approach would be extremely time consuming to create a repository of all the necessary images, as well as creating a program to parse the image capture and turn it into an array of usable data. Additionally, this approach would have the AI take a very long time to actually solve the problem.

So we decided on another approach, since minesweeper is a popular game and has been around for so long, there must be a free open source version of minesweeper that is easy to modify to fit our version. After a few days of researching and back and forth we found a version of minesweeper that we could modify and use for the minesweeper ai. This decision allowed us to focus on the meat of the project rather than coding the entire game of minesweepers ourselves, significantly reducing development complexity and time.

B. Choosing our components

We decided to as python as our chosen language because of its readability and its many libraries for artificial intelligence and constraint solving problems. In particular, Python's compatibility with the Z3 solver made it an excellent choice for this project that uses tons of logic based reasoning [4]. Minesweeper naturally lends itself to constraint satisfaction and first-order predicate logic, as each revealed tile imposes explicit numerical constraints on a fixed set of neighboring cells whose mine states must collectively satisfy those constraints. The Z3 solver is a high-performance satisfiability modulo theories (SMT) solver developed by Microsoft Research that is designed to reason over logical formulas involving Boolean variables [4], arithmetic constraints, and first-order logic. In this project, Z3 is used to model the Minesweeper board by representing each cell as a Boolean variable indicating whether it contains a mine, the location of the cell by row and column, all the while the revealed numbered cells generate constraints that enforce the exact number of mines among their neighboring variables, allowing the solver to maintain a consistent set of all board configurations that satisfy the observed information. Z3 enables logical inference through satisfiability checking, allowing the system to prove cells safe or unsafe via contradiction, eliminate impossible configurations, and guarantee that every decision made by the AI is logically consistent with the current game state rather than relying on heuristic or purely probabilistic reasoning when making a safe move.

C. *How we solve minesweeper*

1) *Gathering Knowledge*

The first step for the Minesweeper AI to function, is for it gather and maintain an accurate logical representation of the current board state. The AI takes direct feedback from the game environment whether or not the cell is revealed or not. Each cell on the board is represented as a Boolean variable indicating if a mine is a mine or is safe. Any revealed cell is always marked as a safe cell, because if it was an unsafe cell and reveled the game would end. When a cell is uncovered the AI records the cell as a tuple: the revealed cell is marked as safe, and the numeric value displayed on the cell that is used describe the number of neighboring mines. That numerical value is stored and transformed into a logical constraint by equating the number shown to the sum of the Boolean variables corresponding to its neighboring cells. These constraints are added to the Z3 solver, which maintains a knowledge base representing all board configurations that remain logically consistent with the observed information. As more cells are revealed, additional constraints are incrementally added, continually narrowing the set of valid board states. This allows the AI to see the board as a system of logical relationships that captures what is currently known by the AI and allows us to predicate logical relationships to make safe decisions.

2) *Making a Safe Move*

After gathering sufficient knowledge from revealed cells, the AI attempts to identify a move that is guaranteed to be safe using deterministic logical reasoning. This process relies on first-order predicate logic implemented through the Z3 solver and is based on proof by contradiction. The frontier is populated with cells adjacent to at least one numbered tile, these cells represent unrevealed locations that we will solve for. For each unrevealed and unflagged cell that lies within the frontier, the AI will temporarily assume that the cell contains a mine and is not safe. This assumption is added as a constraint to the existing knowledge base, and the solver checks whether or not the resulting system of constraints remain satisfiable under the new constraint. If the solver determines that the constraint set is unsatisfiable under this assumption, it implies that the cell cannot logically contain a mine without violating the previously observed constraints. Thus, by proof by contradiction, the AI concludes that the cell must be safe and selects it as the next move. This approach ensures that every safe move made by the AI is provably correct given the current state of the board. Additionally, the AI also identifies and flags any cells that are deterministically proven to be mines using the same logical process before selecting a move so that it reduces the uncertainty of where a mine could be.

3) *Making an Unsafe Move*

In certain Minesweeper configurations, no deterministic safe moves exist due to inherent ambiguity in the board state. When the AI is unable to prove that any unrevealed cell is safe using logical inference, the AI must make an unsafe move. Rather than selecting a cell randomly, the AI applies constrained logical reasoning to minimize risk. For each unrevealed and unflagged cell in the frontier, the AI temporarily assumes that the cell is safe and checks whether this assumption is logically consistent with the current set of constraints maintained by the Z3 solver. If assuming a cell is safe results in a satisfiable constraint system, then that cell represents a valid candidate move, as it does not contradict any known information about the board. Cells that would cause an inconsistency under the assumption of safety are excluded from consideration, as they must then logically contain mines. This process is similar to making a safe move just in reverse, where this one assumes that the cell is safe and checks it against the constraints. From the remaining valid candidates, the AI selects a cell to reveal, acknowledging that the move is not guaranteed to be safe but ensuring that it is not provably dangerous. This approach guarantees that the AI never selects a move that is logically known to result in failure.

IV. Description of the Software

The application is structured in three components that collectively implement the minesweeper environment, AI reasoning, and the graphical interface the user interacts with. The board.py file defines the underlying game logic, including mine placement, adjacency calculations, and win-state evaluation. The ai_logic.py file implements the intelligence of the system by maintaining a logical model of the board using first order predicate logic and the Z3 solver, enabling the AI to reason about safe moves and mine locations based on revealed information. The main.py module integrates these components within an interactive graphical interface built using Pygame [5], managing user input, rendering the board state, and coordinating communication between the game environment and the AI agent.

A. *ai_logic.py*

The ai_logic.py module implements the core intelligence of Jerry the Minesweeper AI by translating Minesweeper observations into a constraint-satisfaction problem and using the Z3 solver to perform logical inference. The file creates the MinesweeperAI class which initializes a Z3 Solver instance and creates a 2D grid of cell variables. The AI also maintains runtime state tracking which cells have been opened, which have been flagged as mines, and a "frontier" set containing unrevealed cells adjacent to revealed numbered tiles. When the game reveals a cell, the add_observation() function updates the knowledge base by asserting that the revealed cell is safe and, if the cell displays a number, adding a constraint that forces the sum of neighboring mine variables to equal the observed number. This continuously reduces the set of valid board configurations that satisfy all observed information. Additionally, the AI precomputes neighbor sets for each coordinate and uses caching to avoid repeated satisfiability checks on the same cells during a turn.

Decision making is performed through two primary functions: make_safe_move() and make_random_move(). The safe-move routine attempts to prove that a candidate frontier cell cannot be a mine by temporarily assuming it is a mine and checking for unsatisfiability; if the constraints become inconsistent, the cell is logically guaranteed safe and is selected. In parallel, the update_flags_deterministic() function identifies cells that must be mines by temporarily assuming they are safe and flagging any cell that causes an unsatisfiable constraint set, then permanently adding the corresponding mine constraint to the solver. If no provably safe move exists, make_random_move() performs a constrained guess by selecting a cell that can be assumed safe without contradicting the current knowledge base, ensuring the AI never intentionally chooses a cell that is logically provable as a mine.

B. *board.py*

The board.py module defines the Minesweeper game environment and serves as the source for the game's mine placement and numeric clue generation when a cell is revealed. It creates the Minesweeper class, which is responsible for initializing the board dimensions, randomly placing a fixed number of mines, and storing the mine layout internally as a two-dimensional grid of Boolean values. Each cell in this grid indicates whether a mine exists at that coordinate, while a separate set tracks the mine locations. Core gameplay functionality is provided through a small set of methods that the GUI and AI rely on during execution. The is_mine(cell) function returns whether a given coordinate contains a mine, enabling the game loop to determine when the player or AI has triggered a loss. The nearby_mines(cell) method computes the numerical clue for any revealed safe cell by iterating over the surrounding eight neighbors and counting how many contain mines, which directly corresponds to the number displayed in traditional Minesweeper. Finally, the won() method checks whether the game has been solved by comparing the set of mines the player has flagged against the true mine set.

C. *main.py*

The main.py module implements the graphical user interface and overall control flow of the application, acting as the bridge between the Minesweeper environment (board.py) and the AI reasoning engine (ai_logic.py). Using the Pygame library, it initializes the display window, loads visual assets such as the flag and mine icons, configure fonts, and computes board layout values such as cell size and board origin so the grid scales correctly within the window. It also defines constant parameters for the game configuration such as board height and width, number of mines, and maintains the runtime state used by the GUI, including the sets of revealed cells, flagged cells, whether the game has been lost, and whether "auto mode" is enabled.

The main event loop continuously listens for Pygame events and updates the screen each frame. When instructions are enabled and the game boots up, main.py displays a title screen and waits for the user to start the game. During gameplay, the file renders each cell as a rectangle whose appearance depends on state: unrevealed cells are drawn as gray tiles, revealed cells are drawn as white tiles, flagged cells display a flag image, and mines are displayed only after a loss condition is triggered. For revealed cells, the game computes the numeric clue using nearby_mines() from the environment and draws the corresponding number in a color-coded style, matching standard Minesweeper conventions. The right side of the interface includes three interactive buttons AI Move, Auto, and Reset. These buttons are handled through mouse collision checks against button rectangles.

Game interaction is coordinated through click logic and a helper routine that applies moves. A left-click on a grid square reveals a cell (unless it is flagged), while a right-click toggles a flag on an unrevealed cell. When the AI is invoked through AI Move button, the module requests a move from the AI model by calling make_safe_move(), and if no provably safe move exists, falling back to make_random_move(). After selecting a move, main.py applies it through the move-handling function: if the cell contains a mine, the game enters a lost state; otherwise, the cell is revealed and the numeric clue is computed and passed into ai.add_observation(), which updates the Z3 knowledge base. If a revealed cell has zero adjacent mines, the function recursively reveals surrounding neighbors, mimicking standard Minesweeper behavior while simultaneously feeding each newly revealed observation into the AI so its constraints remain consistent with the expanded information. The Auto-solve mode extends this same logic by repeatedly invoking the AI every frame (with a short delay for readability), automatically applying safe moves when available and switching to constrained guessing when required. Auto mode stops when the AI reports that no legal moves remain or when the game is lost. The Reset button reinitializes the environment, AI agent, and GUI-tracked state, allowing repeated trials under new randomized mine layouts. The main.py file is responsible for real-time visualization, input handling, and the flow of data between the board and solver, ensuring that human actions and AI decisions are reflected immediately in both the GUI and the AI's logical model.

V. Evaluation Metrics

Our AI was evaluated on two criteria, the first criteria focused on ensuring that the AI consistently makes the most logically optimal decisions possible during gameplay in order to maximize the likelihood of achieving victory. This goal was largely met early in development, as the use of first-order predicate logic and constraint satisfaction allows the AI to deterministically identify safe moves and provable mines whenever sufficient information is available. As a result, the AI reliably solves deterministic regions of the board and avoids moves that are logically guaranteed to lead to failure. The second evaluation criteria required that the AI operate with reasonable responsiveness with each decision being made within one second. Achieving this evaluation metric proved more challenging as the game progressed. As additional cells are revealed, the number of logical constraints maintained by the Z3 solver grows, increasing the complexity of satisfiability checks and, consequently, the time required to compute each move. To address this issue, optimizations were introduced to reduce solver overhead, including removing or deprioritizing constraints that no longer contribute meaningful information to the current decision space.

VI. Conclusion

This project presented *Jerry the Minesweeper AI*, creates an autonomous AI agent that solves Minesweeper using first-order predicate logic implemented through the Z3 theorem solver by modeling each cell as a Boolean variable and translating revealed numerical clues into logical constraints. Through this approach, the AI is able to deterministically identify safe moves, flag provable mines, and make constrained probabilistic decisions when ambiguity is unavoidable, while remaining fully integrated with an interactive graphical interface. Overall, we've learned a lot about predicate logic and how effectively it can be applied to real-world problem solving, particularly in environments like Minesweeper where game rules translate naturally into logical constraints.

References

[1] Free Minesweeper Project, "History of Minesweeper," *FreeMinesweeper.org*. [Online]. Available: https://freeminesweeper.org/minesweeper-history.php. Accessed: Dec. 2025.

[2] C. M. Wilson, "Predicate logic," Univ. of Washington, Winter 2017. [Online]. Available: https://faculty.washington.edu/conormw/Teaching/Files/PhilMath/Winter_2017/Readings/Predicate_Logic.pdf. Accessed: Dec. 2025.

[3] H. Hiroki, "AI-Minesweeper," GitHub repository. [Online]. Available: https://github.com/Hiroki39/AI-Minesweeper. Accessed: Dec. 2025.

[4] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," GitHub repository. [Online]. Available: https://github.com/Z3Prover/z3. Accessed: Dec. 2025

[5] Pygame Development Team, "Pygame documentation." *Pygame.org*. [Online]. Available: https://www.pygame.org/docs/. Accessed: Dec. 2025.