Minimax Algorithm

If we assume we are playing against an optimal player, then an algorithm used to make the best possible decision against said player is the Minimax algorithm. This algorithm searches the tree and tries to maximize the current player’s score, assuming the optimal player chooses their move to minimize the current player’s score. We studied the Minimax algorithm for a “small” game like tic-tac-toe, but this algorithm is infeasible for larger games where the decision tree has a “large branching factor,” which means the tree is too large to keep track of even on a computer.

Neural Networks

Before we discuss the role of artificial intelligence in game programming, it would be a good idea to discuss the concept of a neural network. A neural network can be thought of as consisting of a series of layers of neurons - each neuron can be thought of as a container of a numerical quantity. There are connections between the neurons that are determined by different parameters. The general idea is to determine optimal values for these parameters so that by identifying the numerical values of the first layer, to obtain the numerical values for the last layer in a way that a particular problem is solved. Initially one does not know what those parameters are. By an initial guess and continuous adjustments that are based on instances of the problem where the answer is known, optimal or near-optimal values for these parameters are reached. Once this is accomplished one can use a neural network in order to obtain answers to a given question that is parametrized by the given neural network.

Decision Trees

Given the current state of a game, the next player has a certain number of moves they can explore. This exploration leads to different states of the game, visualized as a tree. Each branch of the tree leads to a different state of the game, and the deeper this tree can be traversed, the further ahead in the game we can see. This tree is called a decision tree.

Q-learning

Another training algorithm we implemented was Q-learning. The Q stands for “quality”, and the idea behind Q-learning is to simulate games and reward good moves and punish poor moves, and then learn from those rewards. The first step is to simulate a game and decide if the outcome of the game was a win or a loss. If it was a win, then we would reward the player for the moves that generated the win, and if it was a loss, then it would punish the moves it made. Each move resulted in a state of the game, and that state was remembered (hashed) so when simulating further games, we could return to the hash and decide if the move is new or old and try to learn from it.

Minisweeper

Minisweeper is a logic puzzle video game generally played on personal computers. It first arrived on the scene in 1992 when it was bundled with Windows 3.1 as Microsoft Minesweeper. Since then, it has spawned many different variations, but the classic game has remained iconic amongst Windows games, only being removed as a pre-installed application with the release of Windows 8, and later being published as a free game on the Microsoft Store. Minesweeper features a grid of tiles, usually of 16x16 size, with 40 mines to uncover. Each tile, when clicked, reveals the number of mines around it. If the player clicks on a mine, they lose the game. The aim is to flag all mines and uncover all tiles that don’t contain mines. Minesweeper is a game with inherent unpredictability. The random position of the mines in each newly-generated game makes it hard to learn effective strategies to win. For this reason, although possible, a traditional approach in reinforcement learning may produce less than desirable results for a beginner. A more fruitful approach would be to hard-code the actions to be taken by the agent, based on what it can learn from the board through previous actions.

Game parameterization

Building on the proof provided by Harvard CS80’s Introduction to Artificial Intelligence with Python, we can parameterize the game and represent our AI’s knowledge through ‘sentences’. Every time a move is made, the board adds a new sentence to the knowledge base and existing sentences in the knowledge base are updated accordingly. If the agent is able to discern a tile as being completely safe, it adds the tile to a list of safe moves to make and removes it from every sentence in the knowledge base. If the agent concludes that a tile has a mine underneath, it flags it and removes that sentence from the knowledge base.

Win/Lose conditions

The agent wins the game when there are no possible moves to make and all mines have been flagged. As expected, the agent loses the game when it clicks on a mine. As of now, the agent wins the game roughly 45 - 55% of the time. The agent was told to play 100 games with three different grid sizes: 8x8, 12x12, and 16x16. This experiment was repeated for each grid size thrice, to get an average win rate of 53% for 8x8, 48% for 12x12, and 46% for 16x16. We can attribute most, if not all these losses to one factor: the random move. The random move is taken when the agent runs out of safe moves and the calculated moves all have the same probability. Unfortunately, this is a characteristic of the nature of the game itself. Even with the inherent randomness, the agent performs very well considering the information at its disposal.