# GAMES AND PROGRAMMING part I

Aaqel Shaik Abdul Mazeed, Abeer Fatima, Ahmed Shah, Jamie Hayes, Joshua Samuel, Moe Hishmeh, Nada Adzic Vukotic, Samuel Effendy, Syed Hussain, Umar Chaudhry, Vincent McNulty, Vira Kasprova
University of Illinois at Chicago

MSCS Undergraduate Research Laboratory

UIC

## About this project

This project was supervised by Drew Shulman and Evangelos Kobotis. The students were divided into groups and they worked separately on different games. During the meetings, all the different games and pieces of code were discussed.

## Games and Computer Programming

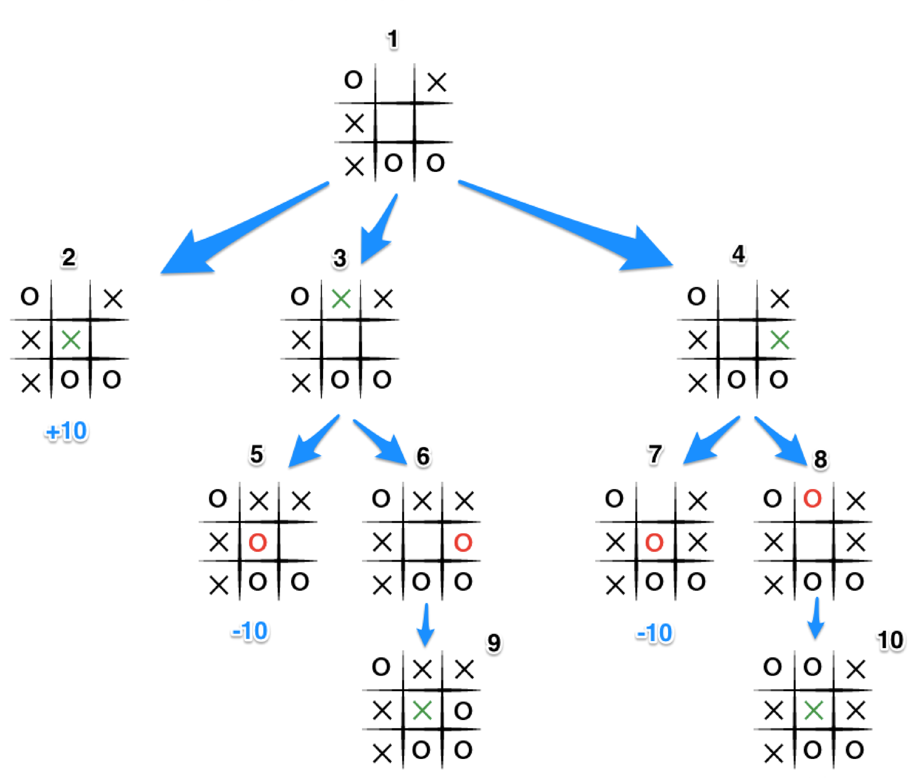
Since the early programming days, there has always been a vivid interest of the scientific community of teaching machines how to play games. The first attempts were rather primitive and it took a while before machines were able to compete with humans as equals. Those days are long gone and through the advances in Computer Science and Artificial Intelligence, we have reached a point that machines dominate humans in an overwhelming way. The purpose of this project was to introduce the students to how different games are programmed so that they can be played by computers.

## Parametrizing a game

One of the preliminary steps needed in the process of developing programs that can play games, is to translate a game into mathematical data that can be processed. For any particular game there is a multitude of ways to achieve this and sometimes it is a trial-and-error process to choose the parametrization that works best. Most games use a coordinate space of a certain dimension and a game can be described as a sequence of collections of vectors in that space.

## 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**.

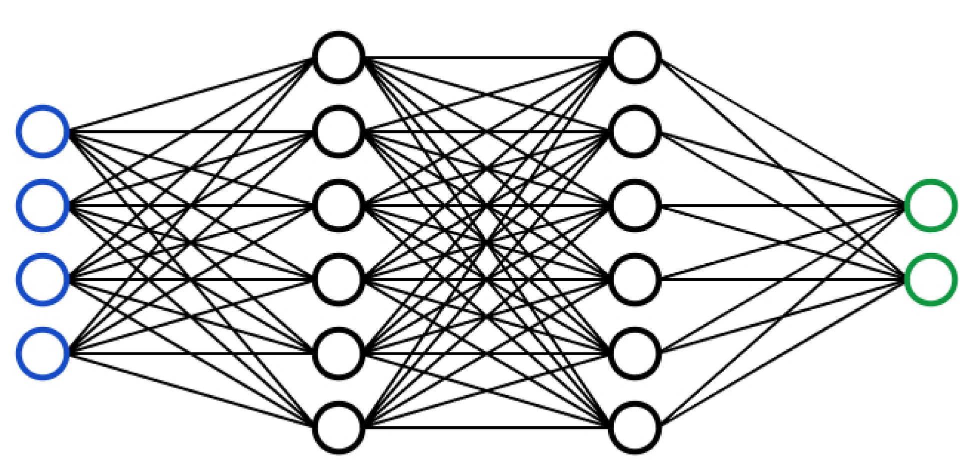



## Minimax Algorithm

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 Minimiax 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

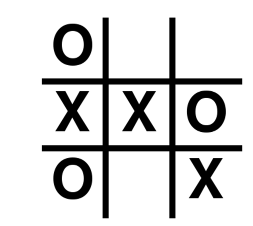
Before we discuss the role of artificial intelligence in game programming, it would be a good idea to discuss the concept of a neural network. A neural network can be thought of as consisting of a series of layers of neurons - each neuron can be thought of as a container of a numerical quantity. There 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.



## $Q$-learning

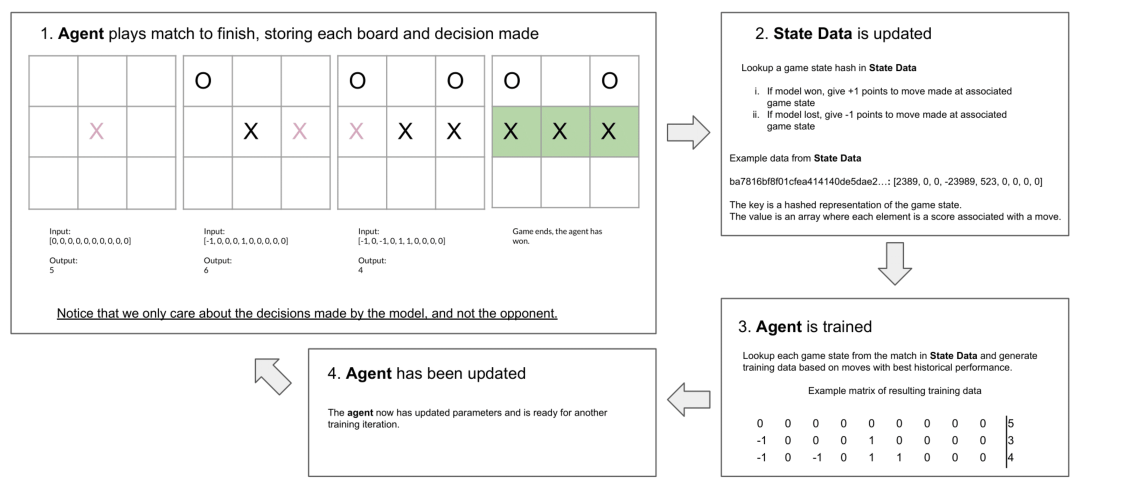
Another training algorithm we implemented was $Q$-learning. The $Q$ stands for "quality", and the idea behind $Q$-learning is to simulate games and reward good moves and punish poor moves, and then learn from those rewards. The first step is to simulate a game and decide if the outcome of the game was 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 were 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.

## Tic Tac Toe



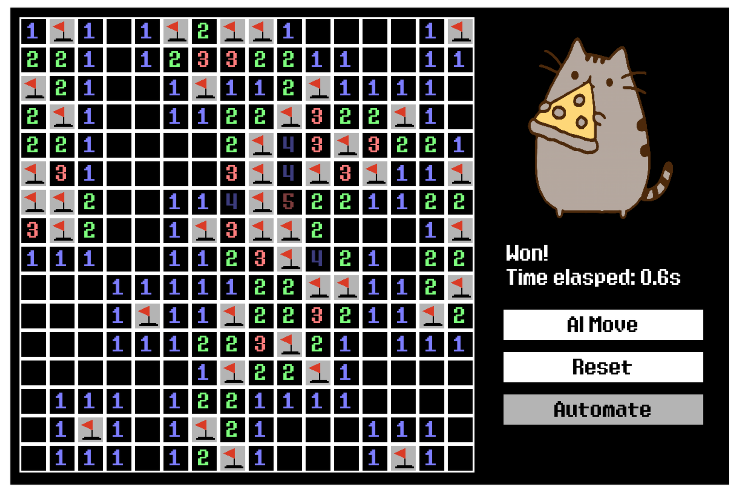
In general, there are three types of machine learning methods; First, the Supervised Learning Method which is more based off of a more task driven approach, examples include classification and regression analysis. Second, the Unsupervised Learning Method, which takes advantage of data driven methods to make inferences, examples include clustering methods. Lastly, the Reinforcement Learning Method, which learns from mistakes and improves the outcome. After careful study and trial/error, Reinforcement Learning was the method we used for our game. The objective was to create a game that can be played to win in as few moves as possible, both from making the first move (player 1) and making the second move (player 2). We used basic object oriented programming by creating a 9 square matrix, where zero represents an open space, -1 represents 'O', and 1 represents 'X'. We also utilized a derivation of Reinforcement Learning called the Incremental Learning method, (similar to reinforcement learning except when new data is presented, our model adjusts), because it gives us the better outcome in winning while being player 1 and player 2. The difficulty was generating and formatting the data to train our models, as well as labeling our data from unfinished games. It was challenging because it was difficult to qualify a model's decision when the outcome is unknown. To fix this problem, Incremental Learning Methods with parameter tuning(s) were used to get our game to work correctly.

1. **Agent** competes against random moves until completion of match
2. Results of the math are used to update the **State Data**
   (a) If **agent** wins, each move made is given $+1$ points
   (b) If **agent** loses or draws, each move made is given $-1$ points
3. Data is generated to incrementally train the **agent**
   (a) A matrix of all the game states seen during the iteration is generated to be used as **features**
   (b) A lookup into **State Data** is conducted to find the moves associated with each game state that have historically have the best performance. These moves are converted into a vector to be used as labels.
   (c) The **features** and **labels** are passed to the **agent** for partial fitting
4. The **agent**'s parameters have been updated to perform better.




The results were as follows: Random player vs Random Player was on average 92%. Trained Player vs Random Player was on average 92%. Finally, Trained player vs Trained player was 99%. After applying these methods, I believe that Incremental Learning is the key to building these models, and additionally, including decision trees with incremental learning may improve the outcome of this game to win at 99%.

## Minesweeper




Minesweeper 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 base provided by Harvard CS50'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 it from every sentence in the knowledge base.

$\{(1, 2), (1, 3), (1, 4), (2, 2), (2, 4), (3, 2), (3, 3), (3, 4)\} = 3$

If a tile at (2, 3) is uncovered and it displayed a '3', the sentence above would be added to the knowledge base and existing sentences would be updated.

**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 consequence of the nature of the game itself. Even with the inherent randomness, the agent performs very well considering the information at its disposal.