Cedric Williams, Janelle Zeng, Daniel Kim

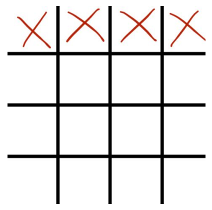
In this reinforcement learning problem, the group created a 3D Tic Tac Toe program in which the program would play against itself to learn to better master the game. After each game, the program would document the squares of the winners in a probability table which would be referenced later in the games to make better moves which would allow it to ultimate beat the opponent.

Two utility functions were utilized in this game. The first utility function taken into consideration was the probability table. The table calculated the utility function of each square and showed which squares were more likely to lead to a win. The second utility function that the group took into consideration was the minimax used with detecting the cases of three in a row and two in a row to either prevent the other player from making the winning move or allow the player to make the final move and get four in a row to win the game.

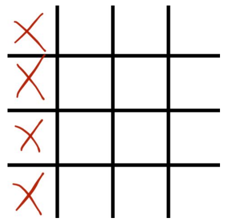
At the start of the game (first 250 games), the program played itself in a random order to simply learn the game - through exploration. In this set of iterations, the program did not use the probability table to make decisions; there were no utility functions at the first few iterations of the game. After playing 250 random iterations of the game until the games either resulted in a win or a draw, the program would use the two utility functions to make better decisions about the moves - exploitation. The game started by playing the highest probability values in order, then used minimax once we've passed 75% of the total_runs. Then utility becomes the results of minimax, 1 or -1 for - win or loss respectively or the default probability table value. If minimax could not determine a winner/loser then the next best probability table value was used.

We created a gameboard class that was used to create the game board itself, set out the rules of the game (i.e. not putting a marker in a square where there is already one), and declare winners. In order to determine the winner, we decided to study all the different ways that winning was possible and listed out seven different possibilities on the 3D Tic Tac Toe. Below we showcase a variety 4x4x4 Tic Tac Toe board wins since it's easier to see winners

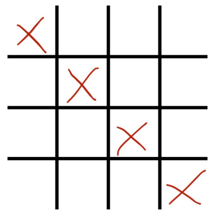
1. Check all the 2D row spaces across one dimension



2. Check all the 2D column spaces in one dimension

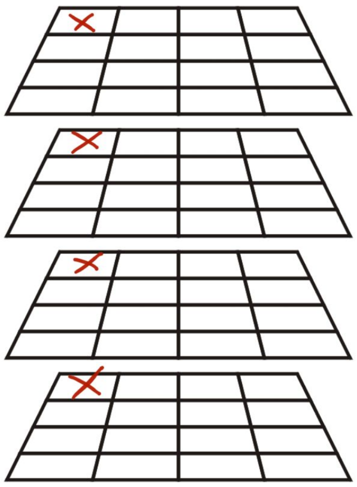


3. Check all the 2D (up and down) diagonal spaces across multiple dimensions

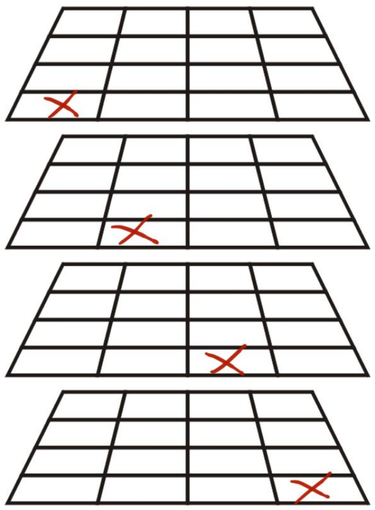


Cedric Williams, Janelle Zeng, Daniel Kim

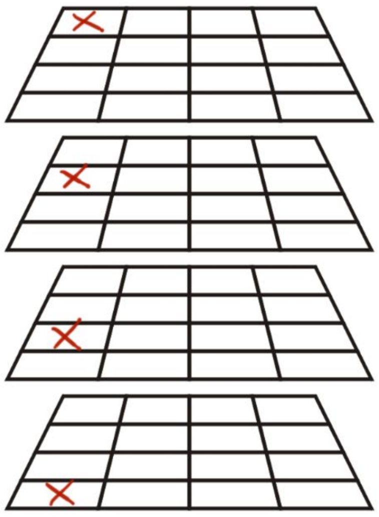
4. Check all the 2D dimension column spaces in one dimension



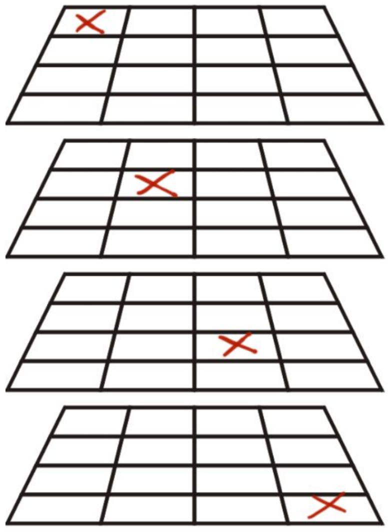
5. Checks all the 3D row diagonal spaces



6. Checks all the 3D column diagonal spaces.



7. Check all the 3D column, row and dimension diagonal spaces



Using these possibilities, we were able to successfully document wins in each iteration.

Another class that was created was BestMove which helped us to determine the best move available on the board using MiniMax. The MiniMax went to depth of 2 as we discovered that going any deeper would lead to too many draws, which meant that the program could not defeat itself. The findBestMove function inspected the whole board to put the marker in the spot which would lead to a four in a row for the current player. The specialCase function was specifically created to find and handle cases such as three in a row or more than one two in a row. This method would return any move that causes or blocks a four in a row or causes or blocks a scenario where there are more than two or three in a row.

The ProbabilityTable's main function was to list out all the squares in the 4x4x4 board. Each square held the utility values corresponding to the spaces on the board. The decimal numbers within the board indicated the percent that the square was part of the winning board. The ProbabilityTable created to print after the iterations of three numbers that were typed in.

In terms of how we chose which moves to make during the learning process, we started with random winners being added to the probability table. Then, we played the highest probability values in order, followed by the use of minimax once we passed 75% of the total runs. If minimax cannot determine a winner/loser then the next best probability table value is used. There was an overall random move decision based on a percentage that approaches zero with each play iteration. If the random move was not chosen then player moves are determined as follows: for the first 25% of the runs: randomly generated moves, for the next moves between total_run_length * 25% to total_run_length * 75%: probability table, total_run_length * .75 and higher: use minimax. We limited minimax till later part of total_runs to reduce the number of games drawn.

Individual contributions of each group member were the following:
Cedric: AI/game architect, debugging, and game mechanics/rules
Janelle: debugging, minimax, special case detection, probCell priority queue, and the probability table developer
Daniel: debugging, minimax & probability table pseudocode writer, and technical writer