AI Plays Ultimate TicTacToe

Tic tac toe is a classic game that many Americans grew up playing. However, the game is solved. 'Solved' meaning if you start first and you know the solution you can't lose the game. Ultimate tic tac toe takes a tic tac toe board and places a tic tac toe board in each square. Depending on the square placed on the inner tic tac toe board determines which outer square the opponent can place their next move. I am inspired to improve upon the existing ultimate tic tac toe AI. There's room for improvement due to ultimate tic tac toe's complexity and the fact that it's difficult to determine the impact of the moves when there are low amounts of squares filled.

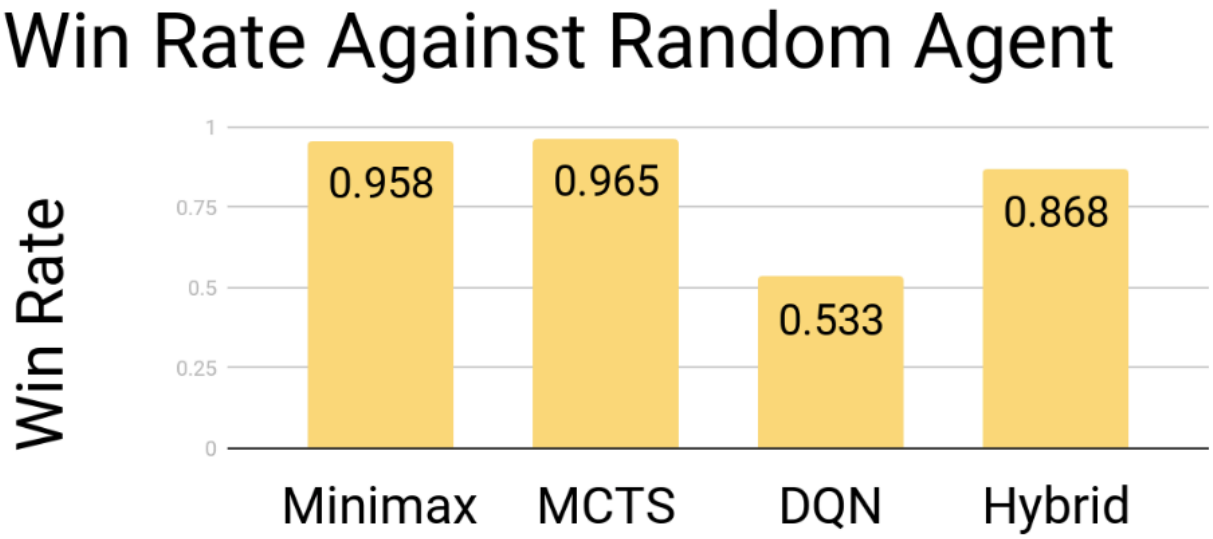
since the creation of the Phil Chen's AI there has been a change to how we think about the variant of ultimate tic tac toe.Guillaume Bertholon discovered a forced win where the starting player can always win. We can have one of two ways of looking about this either
the AI we can create will implement the winning pattern of moves when it's first or our AI can solve a new ultimate tic-tac-toe that doesn't have this forced win.
In the case of the winning pattern there will be an even split against any opponent assuming both players know the winning strategies making it uncompetitive.
there is a variant of ultimate tic tac toe that will solve this issue. This variant proposed by Justin Diamond solves this issue by playing the same game but the first four moves are decided for you. This variant was chosen because it kept its limited number of moves. In our new variant The upper bound for the amount of choices in any game is

O(9 ^7 * 8 ^ 9 * 7 ^ 9 * 6 ^ 9 * 5 ^ 9 * 4 ^ 9 * 3 ^ 9 * 2 ^ 9) or O(1.347045536 * 10^48). The upper bound for the variant of tic tac toe where when placed in a mini board that is already won is the size of O(9 ^6 * 75!) or O(1.31845946 * 10 ^115). You can see the amount of nodes in a decision tree in the variation of ultimate tic tac toe I choose is much smaller. This Should increase the performance of our bots.

The bots used during our research are variations of Minimax and Monte Carlo Tree Search. The Original Implementations of which are provided by a research study done by CHEN, Phil, Jesse Doan, and Edward Xu. Minimax AI uses an evaluation function to determine who is winning on a board then will make the move that results in the best score after seeing a certain number of moves.

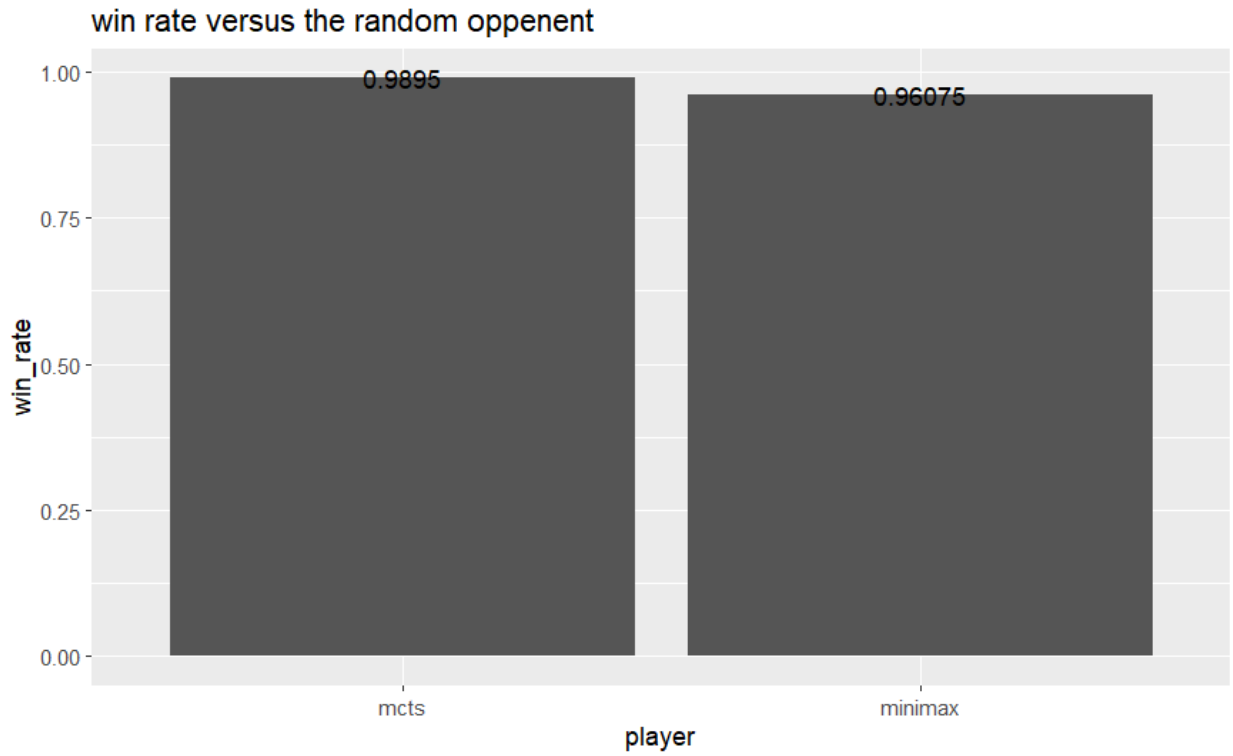
Monte Carlo Tree search or MCTS for short uses randomness to calculate the average win rate of each move. First it picks a move, typically a move that is the result of a move that has a higher win rate. Then they will calculate the result of that game using random moves and add the result to the average to the move and all following moves. After the desired amount iterations is reached the move with the highest average win rate is made

Original results of P Chen Study:



CHEN, PHIL, JESSE DOAN, and EDWARD XU. "AI AGENTS FOR ULTIMATE TIC-TAC-TOE." (2018).

Result after we changed the game by having the first four moves being random:



There is an improvement of both mcts and minimax agents; this is unexpected because random remains unchanged because random already made random moves in the beginning. Random moves were supposed to disadvantage mcts and minimax with these two facts combined would expect a decrease in win rate. Through This observation we can determine that in the original variation the ai ws making moves that were worse than randomly placing.

“We also created a hybrid agent between Minimax and MCTS. Because the board is sparse at the beginning of the game, we predicted that Minimax would be unable to distinguish between the utility different moves and that MCTS would be faster and more effective. Towards the later stages of the game, we predicted that Minimax would be more effective at finding optimal moves.”

CHEN, PHIL, JESSE DOAN, and EDWARD XU. "AI AGENTS FOR ULTIMATE TIC-TAC-TOE." (2018).
And with our observation we can say that minimax is not only worse than mcts but also worse than a random move. The

Our first experiment to improve monte carlo tree search is to increase the number of random games that our agent plays until it decides which move is best and takes it. This number was 100 and will now be 1000.

The original MTCS performed like this graph:

Win Rate first and win rate second refer to the wins and ties * 0.5 total going first and second and win rate is adjusted for how many total you can have

	player	other_player	win_rate_first	win_rate_second	win_rate
1	MCTS 100	Random	993.5	987	0.99025
2	MCTS 100	Minimax	583.0	503	0.54300
3	MCTS 100	MCTS 1000	183.5	141	0.16225

	player	other_player	win_rate_first	win_rate_second	win_rate
1	MCTS 1000	Random	999.0	999.5	0.99925
2	MCTS 1000	Minimax	758.5	746.0	0.75225
3	MCTS 1000	MCTS 100	859.0	816.5	0.83775

The win rate increased by .9 % against random this expected mcts with 100 playouts was already very good against random so there's not much more to improve 20% against minimax.

And Importantly the MCTS with 1000 playouts scored 83.7% of the possible points against MCTS with only 100 playouts. The more the game plays the better picture MCTS gets and the more likely they are to make the best moves and win. It's possible that further increasing the playouts will show continued improvements to performance.

	player	other_player	win_rate
1	MCTS 1000	MCTS 100	0.83775
2	MCTS 2000	MCTS 1000	0.57725
3	MCTS 4000	MCTS 2000	0.52975
4	MCTS 8000	MCTS 4000	0.51275

Increasing a playout is a mostly linear operation being O(log n * n) due to the backpropagation done after simulating the game. The majority of the run time mcts spent was on simulating games and that increased linearly as depth increased so as I doubled the amount of playouts the time it took to play also doubled.

Phil Chen's original evaluation function was whoever had the most boards won is currently winning. But for the early section of the game there are no boards won so minimax makes no attempt to evaluate any differences there are in player's positions .

For my first attempt I borrow heavily from Bichot and Lilan's work to inspire the bot I have created. In Bichot and lilan's reward function three key elements are used when evaluating a player's position one:

- Which type of square do you have? Squares have different numbers of win opportunities depending on the position. The center square has the most with four, the corners have 3 and the edges have only two.
- Do you have any two squares in a row with an empty space to win? This is important in both the larger and smaller squares. Players must have two in a row to complete the tic tac toe

- The value of a smaller square depends on the value of the bigger square it's placed in. Players get only one big square they are to place in but this important element is best used when depth is added to the minimax tree. This way you can determine not just the best square in a board but avoid sending opponents to the best square in the big board.

After recreating the reward function it's time to test it against the original.

results:

	player	other_player	win_rate
1	Minimax	Random	0.9602500
2	Minimax	MCTS	0.2465000
3	Minimax	New Minimax	0.4907500
4	New Minimax	Random	0.9547500
5	New Minimax	Minimax	0.5092500
6	New Minimax	MCTS	0.2547500
7	Minimax	total	0.5658333
8	New Minimax	total	0.5729167

As you can see there is hardly a difference between the new minimax and the old minimax. This is possible due to the shared desire to win more big boards that means the rest of the reward function gives no extra advantage. It's also possible that some strategies are wrong and misdirect the AI while others are steering in the right direction. And a combination of these two strategies equates to a neutral change.

Players of this variant of ultimate tic tac toe have at least one move to guarantee success. This move is where you send an opponent to square that is already won so the opponent is forced to make a move that does nothing. As long as they can't do the same we are effectively plus one move over them. Setting up something like this is certainly possible and an effective way to do this is to have the move that would win the square for you, empty. Once you are placed in that

square once again you both take the board and force the opponent into the same square you just won.

player	other_player	win_rate
New Minimax	Random	0.9547500
New Minimax	Minimax	0.5092500
New Minimax	MCTS	0.2547500
New Minimax	total	0.5729167
Move Taking Minimax	Random	0.9542500
Move Taking Minimax	Minimax	0.5127500
Move Taking Minimax	MCTS	0.2415000
Move Taking Minimax	total	0.5695000

The new minimax is better by a small insignificant fraction if we take the average of all matchups to be the statistic that best shows which ai performs the best. Move taking minimax only evaluates differently in specific scenarios. These scenarios I thought would come up enough to produce a result that's different but it’s not relevant enough to improve performance.

All players use a depth of two to evaluate moves increasing the depth could have the minimax improve its performance and possibly two minimax that have similar performance could change and one could perform better than the other. Previously our move taking minimax performed better than minimax by a small margin but after increasing the depth of both the original minimax does better. The depth increase clearly allowed the minimax player to pick a better move based off what the result will be in the future.

player	other_player	win_rate_depth_4	win_rate
Minimax	Random	0.9635000	0.96100
Minimax	MCTS	0.3935000	0.24300
Minimax	Move Taking Minimax	0.4897500	0.48725
Move Taking Minimax	Random	0.9625000	0.95425
Move Taking Minimax	MCTS	0.3700000	0.24150
Move Taking Minimax	Minimax	0.5102500	0.51275
Minimax	total	0.6155833	0.56375
Move Taking Minimax	total	0.6142500	0.56950

Adding a feature was an insignificant change. Let's try to remove a feature. One feature to remove that could give an AI a more clear direction is the fact the individual squares within the big square have value based on the squares position. The confusing part of this feature is when you consider that placing a piece in a more valuable square means the opponent will be sent to a more valuable square. Sometimes this can result in equal actions meaning the AI will take the first available one. In this iteration the AI player will prioritize sending the player to the worst square instead of picking the best available square. This priority of not risking the most important big square is why I named it Less Risky. Less Risky Did the best between the three minimax strategies but it’s not a large difference.

The next logical strategy to try is more risky where the ai player does not value where the big square that it’s sending to it is and will instead make the best moves within the square it can place in. More risky will still avoid giving opponents opportunity to win big squares or prepapare a two in a row. Sparse is a strategy where testing along side more risky. This strategy is unique because it prioritize having more squares with at least one square over winning those squares, meaning that it will attempt to send opponents to squares they are already in. The strategy is to mimic The strategy described by Bertholon. Of course the automatic is no longer feasible but sparse strategy will lower the number of squares its enemy can win and lower the number of options the opponent can place. You would expect a unique result from sparse due to it being the first to not prioritize big squares but sparse had results similar to the other minimaxs. The results of the average were even closer. All minimax strategies both won against a different minimax strategy and lost against a different strategy. This was the closest the minimax strategies have scored against each other.

	player	average_win_rate
1	mcts	0.79870
2	original	0.54430
3	less risk	0.54225
4	sparse	0.54000
5	more risky	0.53955
6	random	0.03520

	player	other_player	win_rate
1	mcts	random	0.99850
2	less risk	random	0.95775
3	original	random	0.95725
4	more risky	random	0.95650
5	sparse	random	0.95400
6	mcts	sparse	0.75150
7	mcts	original	0.75025
8	mcts	more risky	0.75000
9	mcts	less risk	0.74325
10	original	more risky	0.51475
11	more risky	less risk	0.50925
12	less risk	sparse	0.50875
13	sparse	more risky	0.50325
14	sparse	original	0.50300
15	original	less risk	0.50275

Reference:

CHEN, PHIL, JESSE DOAN, and EDWARD XU. "AI AGENTS FOR ULTIMATE TIC-TAC-TOE." (2018).

Bertholon, Guillaume, et al. "At most 43 moves, at least 29: Optimal strategies and bounds for ultimate tic-tac-toe." *arXiv preprint arXiv:2006.02353* (2020).

Addison, Brycen, Michael Peeler, and Chris Alvin. "Ultimate Tic-Tac-Toe Bot Techniques." *The International FLAIRS Conference Proceedings*. Vol. 35. 2022.

Bichot, Lilian, et al. "Reinforcement Learning Project: Ultimate Tic-Tac-Toe."

Diamond, Justin. "A Practical Method for Preventing Forced Wins in Ultimate Tic-Tac-Toe." *arXiv preprint arXiv:2207.06239* (2022).