Exploration of Agents for Ultimate Tic-Tac-Toe

Thesis

Submitted in Partial Fulfillment

of the Requirements for the

Degree of

BACHELOR OF SCIENCE (Computer Engineering)

at

NEW YORK UNIVERSITY

TANDON SCHOOL OF ENGINEERING

by

Kenneth Liang

January 2017

Copy Number ___

Approval by the Guidance Committee:

Major: Computer Engineering

Julian Togelius, Ph.D.

Thesis Advisor

Associate Professor

Date

Shivendra Panwar, Ph.D.

Department Head

Professor

Date

Dedication

I would like to dedicate this thesis to my family.

And to my friends for their continual support and encouragement: Franky Chen, Hui Huang, Jason Huang, Franky Cham, Edwin Huang, Allen Zheng, and Ying Zhu.

In addition to my advisor, Julian Togelius, whose knowledge acted as a beacon of guidance.

And last, but not least, students at NYU Poly and anyone who took the time to crack this book open.

VITA

Kenneth Liang was born on May 17, 1995 in New York. He went to high school at Valley Stream Central High School in Valley Stream, NY. He began his undergraduate career at NYU Polytechnic School of Engineering during Fall 2013. During freshman year he played on the men's tennis team as 4th singles and 2nd doubles. He worked as a General Engineering teaching assistant starting his sophomore year and enjoys teaching lab to students. In the summer between sophomore and junior year, he worked at NYU Poly, now NYU Tandon, with graduate students. He handled the Android portion of a Facebook music application called MvRock. In preparation of graduation, he began this thesis in Spring 2016 under the advisement of Julian Togelius. The thesis will be completed by January 2017.

Abstract

Exploration of Agents for Ultimate Tic-Tac-Toe

by

Kenneth Liang

Advisor: Julian Togelius, Ph.D.

Submitted in Partial Fulfillment

of the Requirements for the

Degree of

Bachelor of Science (Computer Engineering)

January 2017

Various algorithms are explored to create a bot for ultimate tic-tac-toe. The algorithms covered are minimax and Monte Carlo tree search. The best bot will play in an online competition hosted at TheAIGames and will conform to its constraints. The best bot created uses pure Monte Carlo tree search with random simulation. As of December 13th, 2016, the bot is ranked 33 out of 447 in the competition under the name TicTacker.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Terminology

To be less verbose, Tic-Tac-Toe will be referred to as TTT. The term micro will pertain to one of the nine standard TTT boards in ultimate TTT. The term macro will pertain to all nine micro boards as a whole. For example, a macro two in-a-row will mean having won two micro boards that are in a row. On the other hand, a micro two in-a-row will mean having two squares that are in a row in a micro board.

1.2 Scope

The scope of this paper will cover the algorithms used to create a bot to play ultimate TTT. The two main algorithms used are minimax and Monte Carlo tree search. The evaluation functions and simulation techniques necessary for these algorithms will also be covered. Finally, results of the best bot in the online ultimate TTT competition held at TheAIGames will be stated.

1.3 Rules

TTT is a fairly simple game. There are two players. The first player plays X and the second plays O. Playing on a 3 x 3 board, the first player to have three markers in-a-row in any row, column, or diagonal wins. If the entire board is filled and there is no winner, then it is a tie.

Ultimate TTT takes the standard game even further. Nine standard TTT boards are placed together, 3 x 3, to create a 9 x 9 board. The top-left most square will have the coordinates s(0, 0). The s before the coordinates refers to square. Coordinates will be denoted (column, row). The top-right square is therefore s(8, 0). Likewise, the top-left standard TTT board will be b(0, 0). The b refers to standard TTT board.

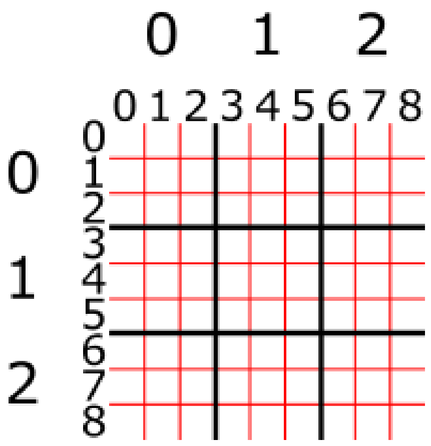

Figure 1.1: Ultimate TTT board with coordinates

In order to win, the player must win three standard TTT games in-a-row in any row, column, or diagonal. A tie occurs when all nine standard boards are finished and there is no winner. A standard board is finished when there is a winner or the board is completely filled up (a tie). When playing, the location the player places his marker determines the next standard board location for the next player to play. For example, if a player plays at s(6, 5) the next player will play in board b(0, 2). If the next board is finished, the next player can play in any unfinished game. For the first move of the game, the entire board is open for player one.

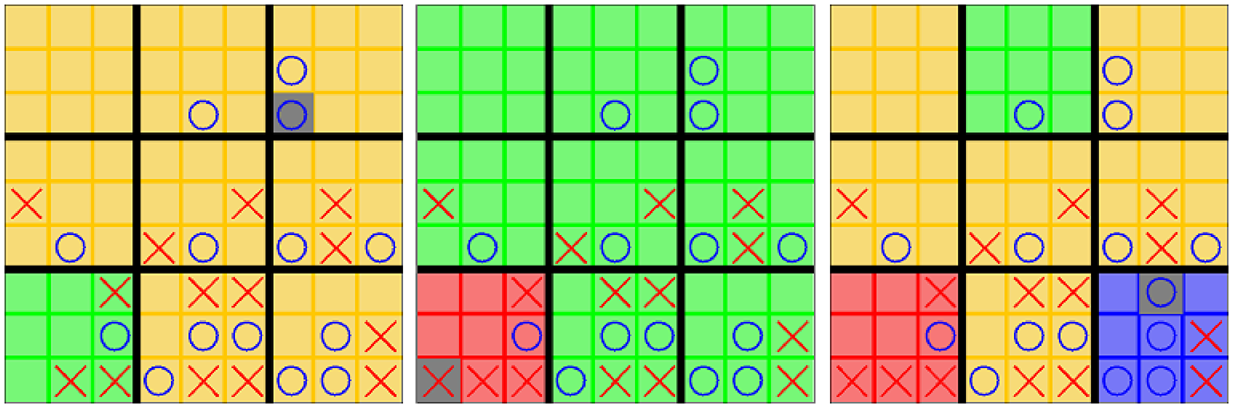
Figure 1.2: Example moves progression

In Figure 1.2, player two has just played s(6, 2). The next board to play is highlighted in green, b(0, 2). Next, player one decides to play s(0, 8), winning b(0, 2). The next board position is b(0, 2), but since it was just finished, the entire board opens up for player two. Player two decides to win b(2, 2) by playing s(7, 6).

1.4 Constraints

Bots are created for the online ultimate TTT competition held at the TheAIGames[1]. The competition features timeout constraints. Each bot has a timebank initialized to 10 seconds. The timebank is the amount of time the bot has to make a move. The timebank is capped at 10 seconds. Every time the server requests a bot to make a move, 500 ms are added to its timebank. If the timebank reaches zero before the bot moves, the bot has timed out. The server will then request a move from the bot again. If the bot times out for the second request, it is given a third and final attempt. If the bot times out for the final request, the bot forfeits the match. For example, a bot fails its first attempt. Its timebank is therefore zero. The server requests again, adding 500 ms to the bot's timebank. The bot fails its second attempt. The server adds 500 ms to its timebank and requests the final attempt. The bot times out and forfeits the match.

1.5 Fairness

Ultimate TTT has some inherent bias. Running 100,000 games between two randomly playing bots, it can easily be seen that player one has an inherent advantage over player two. This is most likely due to the first move that allows player one to dictate the match from the start.

P1 Wins	41057
P2 Wins	36757
Ties	22186
Timeouts	0
P1 p-value	$\sim$0

Table 1.1: 100,000 games between two random bots

[1]At time of writing, the competition is located at http://theaigames.com/competitions/ultimate-tic-tac-toe

As shown in Table 1.1, player one statistically has an inherent advantage as shown by the p-value of approximately one. This value was obtained running a cumulative binomial test ignoring ties. In Microsoft Excel, this is executed as:

$$p\_value = 1 - BINOM.DIST(41057, 77814, 0.5, TRUE)$$

1.6 Configuration

The data obtained in this report are retrieved from a computer running an Intel Core i7 3630qm with 16 GB DDR3 RAM. Computers with different specifications may obtain different results.

2 Heuristic Evaluation Functions

Heuristic evaluation functions are used to evaluate the state of the board. A value is returned that denotes the desirability of this state. The higher the value, the better the state is for the bot. Likewise, the lower the value, the better the value is for the opponent.

All tests between bots were played using half-and-half mode, meaning for half the games one bot was player one and for the other half, the other bot played as player one. This is used to offset the advantages inherent in playing as player one.

2.1 Simple

The simple evaluation counts the number of standard TTT games won and lost. Starting at zero, the heuristic value is increased by one for every standard game won and decreased by one for every standard game lost.

Feature	Value	Description
WIN	999	Value returned when match is won. -WIN is returned when match is lost.
TIE	0	Value returned when match is a tie
MICRO_WIN	1	Value to increase and decrease the heuristic value when a standard game is won and lost, respectively.

Table 2.1: Heuristic values for simple evaluation

2.2 Connecting

The connecting evaluation is an extension of the simple evaluation. This evaluation adds an extra feature that rewards more points for having won two standard TTT games in-a-row where the third game in that row is still open.

Feature	Value	Description
WIN	999	Value returned when match is won. -WIN is returned when match is lost.
TIE	0	Value returned when match is a tie
MICRO_WIN	1	Value for a micro board win
MACRO_CONNECTING	20	Value for two won boards in-a-row in the macro field where the third board is open
MICRO_CONNECTING	1	Value for two squares in-a-row in a micro field where the third square is open

Table 2.2: Heuristic values for connecting evaluation

2.3 Advanced

The advanced evaluation is an improved connecting evaluation. Instead of a value of one for each micro board win, there is a separate value for winning the middle, corner, and side micro boards. Likewise, there is a value for taking the middle, corner, and side squares in a micro board. A middle square/board has coordinates (1, 1). A corner square/board may have coordinates (0, 0), (2, 0), (0, 2), or (2, 2). A side square/board may have coordinates (1, 0), (0, 1), (2, 1), or (1, 2).

Feature	Default Value	Optimized Value	Description
WIN	999	999	Value returned when match is won. -WIN is returned when match is lost.
TIE	0	0	Value returned when match is a tie
MACRO_TWO_IN _A_ROW	50	1.481980523	Value for two won boards in-a-row in the macro field where the third board is open
MACRO_MIDDLE	30	1.140677320	Value for winning the middle micro board
MACRO_CORNER	20	1.406688950	Value for winning a corner micro board
MACRO_SIDE	10	0.998994086	Value for a winning a side micro board
MICRO_TWO_IN _A_ROW	5	0.199311908	Value for two squares in-a-row in a micro field where the third square is open
MICRO_MIDDLE	3	-0.81672889	Value for having a marker on a middle square in a micro board
MICRO_CORNER	2	0.059772673	Value for having a marker on a corner square in a micro board
MICRO_SIDE	1	0.404213732	Value for having a marker on a side square in a micro board

Table 2.3: Heuristic values for advanced evaluation

2.3.1 Evolution Optimization

At first, default values were chosen for each of the features. The advanced evaluation performed significantly better than the connecting evaluation. To further improve the evaluation, the values for the advanced evaluation were evolved using the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [2]. The fitness evaluation function consisted of playing 300 games between the two bots where for 150 games, one bot was player one. In the other 150 games, the other bot played as player one. This is used to balance the inherent advantage in playing as player one. The evolution was done with two minimax bots, both using the advanced evaluation function. One bot used the optimizing values and the other used the default values.

Parameter	Value	Description
Number of Dimensions	8	The number of features to optimize
Population Size	20	The number of candidates to evaluate in each generation
Initial X	0.5	The initial value for each feature
Standard Deviation	0.3	The step size
Stop Fitness	0	The fitness value to stop evolving
Stop Max Function Evaluations	10000	The number of fitness evaluations to stop evolving. This is equal to $population\_size * number\_generations$

Table 2.4: Parameters for CMA-ES

The value returned by the fitness evaluation function is calculated as $h = 300 - (optimizing\_bot\_wins) + 0.5 * (number\_of\_ties)$. CMA-ES seeks to optimize the values that has the lowest fitness value. The best value would therefore be zero when the optimizing bot wins all the games. During the fitness evaluation, both bots are seeded with the game number (0-299). In game one, both bots would be seeded with 0, game two with 1, game three with 2, and so on and so forth. This reduces the amount of noise and allows for perfect repeatability. The same heuristics will produce the same fitness value. Seeding also creates 300 test cases with each test case corresponding to a seed number.

The overall evolution is done using an incremental evolution strategy with CMA-ES. The hope is that this will provide a faster optimization. Lower minimax depths run much faster than higher depths. The results of these lower depths can be used to initialize the optimization of higher depths in an incremental fashion [5]. First the values are evolved by playing two minimax depth 3 bots against each other. The results of that evolution are used to initialize the evolution for two minimax depth 5 bots. Finally, the results from that are used to initialize the evolution for two minimax depth 7 bots. The results from this final evolution are the optimized values. This piece-wise method of evolution is used in hopes of speeding up the optimization process. Minimax depths higher than 7 were not used as speed decreased significantly and the time for CMA-ES to converge would take too long. Note that the WIN and TIE values are not optimized. The purpose of the WIN value is to be an infinitely high value compared to the values of the features.

The minimax 3 evolution stopped after 11 generations when the fitness score reached zero, the best score possible. The minimax 5 evolution was stopped manually after 75 generations. Likewise, the minimax 7 evolution was stopped manually after 55 evolutions. The optimized values shown in Table 2.3 came from the best candidate from the minimax 7 evolution. The best candidate values were found to perform better than the mean values calculated by CMA-ES.

2.4 Comprehensive

The comprehensive evaluation seeks to further improve upon the advanced evaluation by breaking down the features into smaller parts. The micro features in the advanced function are broken down further into micro middle, micro corner, and micro side with each micro having its own middle, corner, and side features.

Feature	Optimized Value	Description
WIN	999	Value returned when match is won. -WIN is returned when match is lost.
TIE	0	Value returned when match is a tie
MACRO_TWO_IN_A_ROW	7.47311431679	Value for two won boards in-a-row in the macro field where the third board is open
MACRO_MIDDLE	5.09311003913	Value for winning the middle micro board
MACRO_CORNER	-2.8228110614	Value for winning a corner micro board
MACRO_SIDE	-2.2634357271	Value for a winning a side micro board
MICRO_TWO_IN_A_ROW	0.19692256855	Value for two squares in-a-row in a micro field where the third square is open
MICRO_MIDDLE_MIDDLE	3.32361428014	Value for having a marker on the middle square of the middle micro board

Table 2.5: Heuristic values for comprehensive evaluation

MICRO_MIDDLE_CORNER	3.13352678820	Value for having a marker on a corner square of the middle micro board
MICRO_MIDDLE_SIDE	-1.5451956807	Value for having a marker on a side square of the middle micro board
MICRO_CORNER_MIDDLE	-0.8282925953	Value for having a marker on the middle square of a corner micro board
MICRO_CORNER_CORNER	-0.3731218288	Value for having a marker on a corner square of a corner micro board
MICRO_CORNER_SIDE	-1.5242912219	Value for having a marker on a side square of a corner micro board
MICRO_SIDE_MIDDLE	1.58642322560	Value for having a marker on the middle square of a side micro board
MICRO_SIDE_CORNER	1.11005596994	Value for having a marker on a corner square of a side micro board
MICRO_SIDE_SIDE	-1.4313661716	Value for having a marker on a side square of a side micro board

Table 2.5: Heuristic values for comprehensive evaluation continued

2.4.1 Evolution Optimization

Default values are not used for this evaluation. Instead, the features are evolved in a similar fashion to the advanced evaluation. This is done by pitting a minimax depth 7 bot using the comprehensive evaluation against a minimax depth 7 bot using the advanced evaluation with optimized values.

Incremental evolution strategy is not used for this optimization, since it was observed in the advanced optimization that minimax depth 7 runs rather quickly. It would be best to optimize at minimax depth 7 directly. Like the advanced optimization, higher minimax depths were not used due to convergence time. WIN and TIE features were also not optimized.

The optimized values shown in Table 2.5 were achieved after stopping the evolution after 33 generations or 660 function evaluations. These values result from the best candidate, the one that yielded the best fitness value. CMA-ES advocates that the mean values are generally better, but after 1000 matches between two iterative deepening minimax bots, the one using the best values won more matches than the one using the mean values.

Parameter	Value	Description
Number of Dimensions	14	The number of features to optimize
Population Size	20	The number of candidates to evaluate in each generation
Initial X	0	The initial value for each feature
Standard Deviation	0.5	The step size
Stop Fitness	0	The fitness value to stop evolving
Stop Max Function Evaluations	10000	The number of fitness evaluations to stop evolving. This is equal to $population\_size * number\_generations$

Table 2.6: Parameters for CMA-ES

2.5 Enhancements

2.5.1 Transposition Table

A transposition table is a hash map that maps a board state to some characteristics of that state. A transposition table can greatly increase the speed of an algorithm if it evaluates the same board state multiple times. In this case, a transposition table is used to map a board state to its heuristic value.

The hash function used in the transposition table is known as Zobrist hashing. This function works by first generating a random integer for each marker and its possible locations on the board. In the case of ultimate TTT, 162 random integers need to be created and stored (81 squares * 2 possible pieces at each square). Thus, each square will hold two randomly generated values, one for X and the other for O. The Zobrist key is generated by simply XORing the corresponding random numbers at all locations with markers. Zobrist keys do not guarantee a unique hash for all board states. However, the chance of collision is so low that the benefits outweigh the negatives.

During testing, using a transposition table reduced the speed of both the minimax and Monte Carlo tree search algorithms. This is most likely due to the fact that the evaluation functions are light and extremely fast to compute. Computing the Zobrist key and searching the table proved to use more overhead than simply evaluating. As such, transposition tables will not be used. It is important to note that during testing, no key collisions were detected.

O: 45 X: 98	O: 22 X: 67	O: 4 X: 76
O: 27 X: 84	O: 65 X: 1	O: 23 X: 94
O: 85 X: 37	O: 89 X: 76	O: 52 X: 82

Table 2.7: Zobrist hashing example

Suppose there are Os at s(0, 0) and s(2, 1) and Xs at s(1, 0) and s(1, 1). The Zobrist key for this board state would be $45 \oplus 67 \oplus 1 \oplus 23$.

2.6 Comparison

A1	A2	A1 Wins	A2 Wins	Ties	Timeouts	A2 p-value
Minimax 7 (Simple)	Minimax 7 (Connecting)	24	440	36	0	$\sim$0
Minimax 7 (Connecting)	Minimax 7 (Advanced)	147	299	54	0	1.26232E-13
Minimax 7 (Advanced)	Minimax 7 (Comp)	72	425	3	0	$\sim$0
Iterative (Simple)	Iterative (Connecting)	45	376	79	0	$\sim$0
Iterative (Connecting)	Iterative (Advanced)	136	282	82	0	1.84963E-13
Iterative (Advanced)	Iterative (Comp)	107	381	12	0	$\sim$0

Table 2.8: Heuristic evaluation comparison

Table 2.8 shows that comprehensive $>$ advanced $>$ connecting $>$ simple. This is true for both minimax and iterative deepening minimax. This is shown statistically using a cumulative binomial test where n is A1 Wins + A2 Wins. Ties are ignored. The p-value shows that A2 is statistically better than A1 in all the cases.

3 Simulation Types

Simulations or playouts for Monte Carlo tree search (MCTS) will be discussed here. Simulations generally play out from a node until a terminal state is reached. A terminal state is a finished state, meaning there is either a winner or a tie. The simulation will then return a value (WIN, TIE, or LOSS) from the perspective of the bot.

All tests between bots were played using half-and-half mode, meaning for half the games one bot was player one and for the other half, the other bot played as player one. This is used to offset the advantages inherent in playing as player one.

Constant	Value
WIN	1.0
TIE	0.5
LOSS	0.0

Table 3.1: Simulation values

3.1 Random

The random simulation is simple. Starting from a leaf node, random moves are played until a terminal state is reached. The result (WIN, TIE, or LOSS) is then returned.

3.1.1 Random RAVE

This simulation is the RAVE version of the random simulation. Moves made by the bot and the opponent during the simulation will be stored in a container for bot moves and opponent moves, respectively. This is used to implement RAVE functionality.

3.2 Win-First Random

The win-first random simulation is an extension of the random simulation. Starting from a leaf node, for each move, if there is a winning move, play it, otherwise play a random move. The result is then returned.

3.3 Early Playout Termination

Early playout termination or EPT utilizes a heuristic evaluation function to return a simulation result depending on the heuristic value [3]. The evaluation function is called after a certain number of moves. If a terminal state if reached before that number, that result will simply be returned. Early playout termination will be used with the best evaluation function, the comprehensive evaluation.

EPT is extremely powerful with a fast and accurate evaluation function [3]. This allows for more simulations to be made, allowing MCTS to faster converge to the optimal move. With heavy evaluation functions, transposition tables can be used to improve its speed.

EPT was tested with random simulations and win-first random simulations. Results showed random simulations to be much better. This is mostly likely due to the fact that the overhead required to search for winning moves first in win-first random significantly reduced the number of possible simulations. With less simulations, MCTS chose suboptimal moves.

3.3.1 Number of Moves Threshold

The number of moves threshold is the number of moves that will be played out before calling the evaluation function. The optimal value was chosen empirically.

Number of Moves Threshold	20	30	40	50	60	70
MCTS UCT (EPT Comp)	34	68	115	189	175	175
MCTS RAVE Heuristic (Advanced)	340	275	218	150	151	144
Ties	126	157	167	161	174	181
Timeouts	0	0	0	0	0	0
Score	97	146.5	198.5	269.5	262	265.5

Table 3.2: Testing EPT number of moves threshold

Testing the values shown in Table 3.2, the best number of moves threshold is 50. These values were tested with the heuristic boundary at 0.5.

3.3.2 Heuristic Boundary

The heuristic boundary is the threshold that separates wins, losses, and ties. If the evaluation function returns a value in the interval $[-boundary, boundary]$, then a TIE will be returned by the simulation. A value greater than $boundary$ will return a WIN and a value less than $-boundary$ will return a LOSS.

Heuristic Boundary	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
MCTS UCT (EPT Comp)	187	154	156	172	168	191	185	185	169
MCTS RAVE Heuristic (Advanced)	149	154	154	158	155	156	149	154	158
Ties	164	192	190	170	177	153	166	161	173
Timeouts	0	0	0	0	0	0	0	0	0
Score	269	250	251	257	256.5	267.5	268	265.5	255.5

Table 3.3: Testing EPT heuristic boundary

The heuristic boundary is found empirically. The best value is approximately 0.7. The values were tested with a number of moves threshold of 50.

3.3.3 Early Playout Termination RAVE

This is the RAVE version of EPT. It will use the optimized values found using MCTS UCT: 50 for number of moves threshold and 0.7 for heuristic boundary.

3.4 Comparison

A1	A2	A1 Wins	A2 Wins	Ties	Timeouts	A2 p-value
MCTS UCT (Random)	MCTS UCT (Win-First Random)	197	115	188	0	0.999998
MCTS UCT (Random)	MCTS RAVE (Random)	162	106	232	0	0.999625
MCTS UCT (EPT Comp)	MCTS RAVE (EPT Comp)	163	122	215	0	0.991169
MCTS UCT (Random)	MCTS RAVE (EPT Comp)	156	116	228	0	0.991059
MCTS UCT (Random)	MCTS UCT (EPT Comp)	141	123	236	0	0.852290

Table 3.4: Simulation comparison

Table 3.4 shows the dominance of a simple random simulation. It won more games than all other simulations. Only against EPT did it not achieve statistical significance. It is quite clear that the best simulation is either random or early playout termination. These two simulations both have a low overhead cost that allows MCTS to execute many iterations.

4 Algorithms

The algorithms tested for ultimate TTT will be discussed here. The two main types of algorithms tested are minimax and Monte Carlo tree search. Monte Carlo tree search will be referred to as MCTS.

All tests between bots were played using half-and-half mode, meaning for half the games one bot was player one and for the other half, the other bot played as player one. This is used to offset the advantages inherent in playing as player one.

4.1 Random

The bot will choose a random move out of all the available moves. This algorithm is used to evaluate the fairness in ultimate TTT as discussed in section 1.5.

4.2 Minimax Algorithms

Minimax is an algorithm that assumes both players will play the moves that maximizes the reward. The algorithm requires a heuristic evaluation function to evaluate game states. The name is derived from the fact that one player will be trying to maximize the heuristic value, while the other is trying to minimize it.

Starting from the current game state, a move tree will be created for all the different combinations of moves up to a certain depth. When that depth is reached, the evaluation function is called to evaluate the state. This value is then sent back up the tree. The bot will choose the highest values (maximum values) to send back up, while the opponent will send the lowest values (minimum values). The best move to make will be the one corresponding with the highest value.

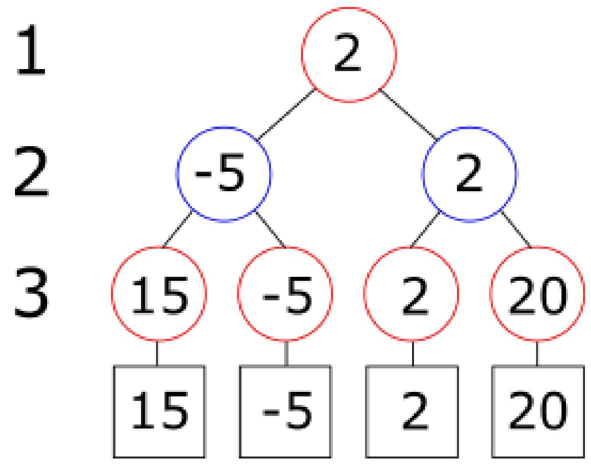

Figure 4.1: Minimax depth 3 example

Figure 4.1 shows an example minimax tree of depth 3 with a branching factor of two moves. After the moves are played out, the evaluation function returns a heuristic value. At depth 3, the bot returns the maximum value, but there is only one value so that value is returned. At depth 2, the opponent chooses the minimum value to return. The left branch returns -5 between 15 and -5. The right branch returns 2 between 2 and 20. At depth 1, the current state, the bot chooses the maximum value, 2, and plays the move associated with that value.

4.2.1 Minimax with Alpha-beta Pruning

Alpha-beta pruning is an optimization that decreases the execution time of minimax. Two values, α and β, are used and initialized to $-\infty$ and ∞, respectively. These two values hold the best past heuristic values. Specifically, α value holds the highest value found so far for the bot and β value holds the lowest value found so far for the opponent. Whenever, $\beta \leq \alpha$, prune this branch and return a value; α is returned if this is the bot's node else β is returned if this is the opponent's node. The reason this branch is pruned is due to the fact that once the two values overlap, there is no reason to explore further, since the the other player will simply ignore the value and choose his respective α or β value.

For example, suppose α is currently 12 and β is currently 20. This node is currently the bot's node. The bot makes a move at this node and the value received is 25. α is updated to 25 and now $\beta \leq \alpha$. Other moves do not need to be tested, since the opponent will seek to minimize the value and simply return the value 20 in the upper levels, since that value is the best value the opponent has found thus far. Further exploration will only result in the bot returning either 25 or some value higher than this, which will simply be ignored by the opponent. For this reason the branch can be pruned.

4.2.2 Iterative Deepening Minimax with Alpha-beta Pruning

Iterative deepening is an algorithm used to iteratively create the move tree depth-by-depth. This means that in conjunction with minimax, the search tree is first created with depth one, traversed, created with depth two, traversed, and so on and so forth. Reaching the same depth with iterative deepening is slower than plain minimax due to the fact that previous depths must be explored.

The benefit of iterative deepening is better time management. The algorithm can continuously search trees with increasing depths until some time constraint is exceeded. By making sure that the time constraint is never greater than the timebank, the bot will never timeout. Careful assignment of time constraints in response to different scenarios will allow the bot to search greater depths than plain minimax. For example, plain minimax at depth 7 never times out. This means that each move takes approximately the same amount of time that the game server gives back every action or less. The moves therefore usually take 500 ms or less. This is not efficient use of the timebank, since the bot will have close to a full timebank when the match is over. Since iterative deepening guarantees no timeouts, greater time constraints, such as one second, can be given to decide a move. Despite iterative deepening being slower, the extra 500 ms is more than enough time for the bot to explore past depth 7. This more efficient time management ultimately allows iterative deepening minimax to perform better than plain minimax.

When time constraints are exceeded the algorithm ceases exploration and returns the best move for the previous depth immediately. Results from the incomplete depth are discarded. For example, if the algorithm is currently exploring a move tree of depth 10 and exceeds the time constraint, it will discard those incomplete results and return the best move from depth 9. The reasoning for this is that since the depth 10 tree was only partially explored, the results are invalid. Results are only correct if the entire tree is explored.

Case	Time (ms)
Move number <15	500
Available moves < 4	500
Available moves < 7	800
Available moves < 10	1100
Available moves ≥ 10	1700

Table 4.1: Iterative deepening minimax time values

4.3 Monte Carlo Tree Search Algorithms

Monte Carlo tree search is an algorithm that is based on guided sampling of the search space in order to determine the best course of action. In ultimate TTT, this means determining the best move to make. Monte Carlo tree search consists of many iterations. Each iteration is composed of four parts: selection, expansion, simulation, and backpropagation.

Selection is the process of choosing a leaf node in the current move tree to explore. For the first iteration, the root node (current state node) will always be selected. Expansion is creating a child node for the selected node. If the selected node is a finished state, expansion and simulation are skipped and only backpropagation will be executed. Backpropagation is necessary in this case because the next iteration will most likely select the same node if the values are not updated, resulting in an infinite loop scenario where there are many iterations, but with no useful work done. Simulation is playing the game out from the newly created node until a finished state (WIN, TIE, LOSS) is reached. Backpropagation is the process of updating the values in all the nodes traversed during the iteration, including the node created in expansion. Each node holds q, the total reward, and n, the total number of visits. During backpropagation, n is always incremented. Q is updated with the result of the simulation. In this case, WIN corresponds to 1, TIE to 0.5, and LOSS to 0.

Over a period of time many iterations will have been executed. Iterations cease after the time limit is reached. According to a relatively recent MCTS survey by Browne et al. (2012), there are different criteria in choosing the best root child (best move). The max child is the child with the greatest reward. The robust child is the child with the most visits. For this MCTS implementation, the max-robust child will be chosen. This is the child with the highest reward and the most visits [8]. In the case that the max-robust child does not exist, the robust child will be selected. This is due to the nature of the policy used in selection that generally equates more visits as being a better choice than a greater reward.

Case	Time (ms)
Move number < 30 and available moves < 4	500
Move number < 30 and available moves < 7	800
Move number < 30 and available moves < 10	1100
Move number < 30 and available moves ≥ 10	1700
Move number ≥ 30 and available moves < 4	400
Move number ≥ 30 and available moves < 7	700
Move number ≥ 30 and available moves < 10	1000
Move number ≥ 30 and available moves ≥ 10	1600

Table 4.2: MCTS time values

4.3.1 Monte Carlo Tree Search UCT

UCT stands for upper confidence bound applied to trees. It is a policy used during selection that balances reward and exploration. The formula is [8]:

$$UCT = \frac{q}{n} + C\sqrt{\frac{\log t}{n}}$$

where q is the reward of the child considered, n is number of visits of the child considered, t is the number of visits of the current node (the parent), and C is the exploration constant. The left-hand side of the formula selects nodes with a greater reward, while the right-hand side of the formula selects nodes with a few number of explorations [8]. The exploration constant is set, usually empirically, to determine how much exploration is optimal for the task at hand. During the selection process, the child with the greatest UCT value is selected starting from the root node. This continues iteratively until a leaf node is reached.

It is important to note that q and n are updated according to who created the node. If the bot created the node (made the move towards this node), the values are updated normally with WIN(1), TIE(0.5), and LOSS(0). If the opponent created the node, the values are updated with WIN(0), TIE(0.5), and LOSS(1); this is due to the fact that WIN, TIE, and LOSS are from the persective of the bot. Updating the values in this regard allows MCTS to converge to minimax after many iterations. The selection process is also greatly improved.

Exploration Constant

As mentioned previously, the exploration constant is usually chosen empirically. Values are tested by playing 1000 games between MCTS UCT and minimax 7 using the advanced evaluation function. As shown in Table 4.3, the best exploration constant is approximately 0.5. This value will be used in the MCTS UCT implementation.

Exploration Constant	0.3	0.4	0.5	0.6	0.7	0.8	0.9
UCT	652	733	762	752	738	745	717
Minimax 7 (Advanced)	147	111	104	124	124	119	133
Ties	201	156	134	124	138	136	150
Timeouts	0	0	0	0	0	0	0
Score (UCT + 0.5 * Ties)	752.5	811	829	814	807	813	792

Table 4.3: UCT exploration constant testing

4.3.2 Monte Carlo Tree Search RAVE

Rapid action value estimation (RAVE) is a method for quickly initializing the move tree for MCTS to allow for a faster convergence to the best move [7]. This is done by holding two sets of values for the total reward and total number of visits. One set is the UCT values while the other is called the AMAF values. Both sets of values factor into an adapted UCT equation during the selection process [7]:

$$UCT = (1-\beta)\frac{q_{UCT}}{n_{UCT}} + \beta\frac{q_{AMAF}}{n_{AMAF}} + C\sqrt{\frac{\log t_{UCT}}{n_{UCT}}}$$

where β is the child's beta constant, q is the child's total reward, n is the child's number of visits, C is exploration constant, and t is the parent node's number of visits. The best exploration constant found for MCTS UCT, 0.5, will be used for MCTS RAVE as well.

In the beginning β is one, meaning the AMAF values play a large role in the selection process. Overtime β decays to zero according to the minimum mean square error (MSE) schedule [7]:

$$\beta_{new} = \frac{n_{AMAF}}{n_{UCT} + n_{AMAF} + R * n_{UCT} * n_{AMAF}}$$

When β reaches zero, the AMAF values play no role in the selection process and the adapted UCT equation effectively converges to the original UCT equation [4].

Nodes chosen during the selection process have both their UCT and AMAF values updated normally. In addition, siblings of these nodes are also updated if the moves leading to these nodes match a move made during the simulation. For example, nodes A, B, and C are siblings. Node A is part of the selection process. During backpropagation, node A's UCT and AMAF values are updated. The move that created node B does not match any move made during the simulation. It is not updated. However, node C does meet the criterion; its AMAF values are updated. Like MCTS UCT, it is important to update the nodes with respect to who made those nodes. Nodes that are created by the bot are updated with WIN(1), TIE(0.5), and LOSS(0), while nodes that are created by the opponent are updated with WIN(0), TIE(0.5), LOSS(1). This ensures that during the selection process, both the bot and the opponent chooses the best move for them according to the adapted UCT equation. This results in a tree that converges to minimax after many simulations.

RAVE quickly builds a tree with rough estimates since it updates selected nodes and their siblings. While this is usually beneficial, it may also be dangerous. Sibling nodes are updated if their moves match one made during the simulation. However, while the moves are the same, they are usually made in different contexts [7]. A move made in one context may lead to a win, while the same move made in another context may result in a loss. Despite this, RAVE usually increases the win percentage of the bot.

RAVE Constant

The RAVE constant is used to adjust the rate of decrease in the minimum MSE schedule [7]. The RAVE constant can be found empirically.

RAVE Constant	0.00001	0.0001	0.001	0.01	0.1	1	2
RAVE (Win-First Random)	681	702	697	750	761	741	729
Minimax 7 (Advanced)	120	123	113	113	101	111	105
Ties	199	175	190	137	138	148	166
Timeouts	0	0	0	0	0	0	0
Score (RAVE + 0.5 * Ties)	780.5	789.5	792	818.5	830	815	812

Table 4.4: RAVE constant testing

The RAVE constant is found using the values shown in Table 4.4. This is done by playing MCTS RAVE using the win-first random RAVE simulation against a minimax 7 bot using the advanced evaluation algorithm. A thousand (1000) games are played for each value. The best value is approximately 0.1.

4.4 Monte Carlo Tree Search Enhancements

4.4.1 GRAVE

Generalized rapid action value estimation (GRAVE) is an extension of RAVE. It functions by using ancestor values for AMAF initialization [6]. Quick tests were run using GRAVE, but they proved to perform worse than RAVE. For this reason, GRAVE will only receive a quick mention here.

4.4.2 Heuristic Node Initialization

Heuristic node initialization is a technique that uses a heuristic evaluation function to initialize the q and n values in a MCTS node [7]. For nodes with better states, the values q and n will be higher, guiding the selection process to favor these nodes more.

A MCTS bot using heuristic node initialization will be called MCTS UCT heuristic or MCTS RAVE heuristic. Testing done with MCTS UCT heuristic showed a decrease in skill level compared to MCTS UCT. For this reason, MCTS UCT heuristic will not be discussed in detail.

Heuristic Bias

The heuristic bias is used to determine the weight of the heuristic value for the total reward. It is a constant multiplied to the heuristic value. For evaluations that may result in negative values, an offset may be applied to each heuristic value first. This is to prevent negative q and n values, which may never occur. The total reward is calculated as:

$$q = (heuristic\_value + OFFSET) * HEURISTIC\_BIAS$$

For MCTS RAVE, q_{AMAF} is calculated in the same manner:

$$q_{AMAF} = (heuristic\_value + OFFSET) * HEURISTIC\_BIAS$$

Assuming UCT and AMAF heuristic confidences of two, the heuristic bias was found to be approximately 0.1. This value will be used in MCTS RAVE heuristic.

Heuristic Confidence

The heuristic confidence is used to determine the number of visits for the node [7]. The number of visits is calculated as:

$$n = q * UCT\_HEURISTIC\_CONFIDENCE$$

where q has been initialized with the heuristic bias. Since n may never be less than q, the heuristic confidence may never be less than one. For MCTS RAVE, n_{AMAF} is calculated with a separate heuristic confidence:

$$n_{AMAF} = q_{AMAF} * AMAF\_HEURISTIC\_CONFIDENCE$$

Using the heuristic bias of 0.1, and assuming an AMAF heuristic confidence of two, the best value for the UCT heuristic confidence was found to still be approximately two. The same occurred during the AMAF heuristic confidence testing. The value two will be used for both confidences.

4.4.3 Heuristic Simulation Influence

Heuristic simulation influence is a technique that uses evaluation functions to influence a simulation. Two types were tested: partial and total. Both types underperformed compared to a completely random simulation. This result is not new in the research of MCTS; other tests have shown that biasing simulations may lead to a general decrease in performance [4, 7]. This may be due to the extra overhead required in using an evaluation function, which limits the total number of iterations, and the biased nature of using heuristic influence, which constrains the search space of a simulation. Constraining the simulation search space does not allow MCTS to explore all possible scenarios it needs to perform well. Given the results of heuristic simulation influence, it will not be used in any of the bots.

Partial

Partial influence plays the move with the best heuristic value given a certain percentage, otherwise it defaults to playing a random move. This is a subtle way of influencing a simulation.

Total

Total influence affects the entire simulation. For a certain percentage all the moves played with be the ones with the best heuristic values. Otherwise the entire playout with be completely random.

4.5 Comparison

4.5.1 Minimax

A1	A2	A1 Wins	A2 Wins	Ties	Timeouts	A2 p-value
Minimax 1 (Comp)	Minimax 2 (Comp)	20	463	17	0	∼0
Minimax 2 (Comp)	Minimax 3 (Comp)	126	277	97	0	8.88178E-15
Minimax 3 (Comp)	Minimax 4 (Comp)	165	255	80	0	4.15338E-06
Minimax 4 (Comp)	Minimax 5 (Comp)	135	261	104	0	6.07534E-11
Minimax 5 (Comp)	Minimax 6 (Comp)	148	222	130	0	4.58195E-05
Minimax 6 (Comp)	Minimax 7 (Comp)	133	271	96	0	1.42086E-12
Minimax 7 (Comp)	Iterative (Comp)	253	175	72	0	0.999904776

Table 4.5: Minimax comparison

Table 4.5 shows the correctiveness of the minimax algorithm. As depth increase, performance increases. However, it is interesting to note that iterative deepening minimax performed worse than minimax 7. This curious case is caused by the optimization done in CMA-ES. Optimization done at a certain depth improves the performance at that depth and all lower depths. However, subsequent depths drop in performance. This is the case here. To fix this problem, the optimization would have to be done using iterative deepening minimax instead of minimax 7. Optimization also causes a disparity between odd and even depths. Since the optimization was done using minimax 7, odd depths will perform better. This can be seen from depths 3 - 7 where depth 3 wins 277 games, which then drops down to 255 at depth 4, rises back up to 261 at depth 5, drops back down to 222 at depth 6, and finally rises again to 271 at depth 7. The decline in performance for iterative deepening is most likely attributed to a combination of these two factors: functioning at a depth greater than 7 and reaching even depths at certain moves. Based on these results, minimax 7 is the best choice using the comprehensive evaluation.

4.5.2 Monte Carlo Tree Search

A1	A2	A1 Wins	A2 Wins	Ties	Timeouts	A2 p-value
MCTS UCT (Random)	MCTS RAVE (Random)	162	106	232	0	0.999625
MCTS UCT (Random)	MCTS UCT Heuristic (Random)	207	74	219	0	$\sim$1
MCTS RAVE (Random)	MCTS RAVE Heuristic (Random)	171	103	226	0	0.999976
MCTS UCT Heuristic (Random)	MCTS RAVE Heuristic (Random)	106	157	237	0	0.00065

Table 4.6: Monte Carlo tree search comparison

According to Table 3.4 in section 3.4, a random simulation is best. Therefore, only the random simulation will be used to test the various MCTS versions.

Based on the above data, it is clear that MCTS UCT (Random) outperforms all the other MCTS variations. The overhead required for the variations only proved to hinder MCTS.

4.5.3 Best

A1	A2	A1 Wins	A2 Wins	Ties	Timeouts	A2 p-value
MCTS UCT (Random)	Minimax 7 (Comp)	456	34	10	0	$\sim$1

Table 4.7: Best comparison

Table 4.7 shows the results between the best minimax algorithm, minimax 7 using the comprehensive evaluation, and the best MCTS algorithm, MCTS UCT (Random). MCTS UCT (Random) overwhelmingly defeats minimax 7. To improve minimax performance, optimization will have to be done with CMA-ES using iterative deepening minimax. This will allow iterative deepening minimax to perform better against minimax 7 and MCTS.

5 Competition Results

5.1 Bot Results

The current bot on TheAIGames is using MCTS UCT (Random) under the name TicTacker. As of December 13th, 2016 it is currently ranked 33 out of 447. It is a high ranking bot. To break the top 20 using MCTS, more optimizations and research will have to be done. One method is utilizing early playout termination with a better evaluation function. MCTS UCT (EPT Comp) is the only algorithm to statistically show to be on par with MCTS UCT (Random). This can be seen in Table 3.4. A better evaluation function will drive the maximum number of moves down, allowing for more iterations. This will cause the performance break necessary. Other improvements are code optimizations, such as using bit manipulation for the board logic.

5.2 Top Bots

In a discussion[1] in the competition's forum, competitors briefly discussed algorithms used. Many of the top bots used some variation of minimax, such as negamax with iterative deepening and other numerous optimizations. These optimizations include killer moves, move ordering, transposition tables, alpha-beta pruning, and quiescent search. Only a few bots used MCTS. One such bot is ranked top 5. The creator of this bot claims it is able to obtain 150,000 - 200,000 iterations. For another MCTS bot, ranking in the top 20, its creator claims it obtains approximately 90,000 iterations. For comparison, MCTS UCT (EPT Comp) obtains approximately 25,000 iterations. This shows a direct correlation between performance and number of iterations. From this online discussion, it is clear that both minimax and MCTS have the capability of performing well. To break the top 20 barrier, heavy optimization is key.

[1] http://theaigames.com/discussions/ultimate-tic-tac-toe/57893ed25d203c1f6ccc86c7/how-deep-do-the-top-ais-search-/1/show

5.3 Conclusion

The bot currently being used in the competition is MCTS UCT (Random). While the bot performs well, it can be further improved. According to Table 3.4 there is no statistical evidence that MCTS UCT (Random) is better than MCTS UCT (EPT Comp). There is promise that early playout termination can surpass a purely random simulation by implementing a better heuristic evaluation function. For now, MCTS UCT (Random) has proven itself to be a high ranking bot with exceptional performance.

Bibliography

[1] Chaslot, G., Winands, M., Uiterwijk, J., Herik, H., and Bouzy, B. (2007). *Progressive Strategies for Monte-Carlo Tree Search*. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.3015&rep=rep1&type=pdf.

[2] Hansen, N. (2016). *The CMA Evolution Strategy: A Tutorial*. Retrieved from https://arxiv.org/pdf/1604.00772v1.pdf.

[3] Lorentz, R. (2015). *Early Playout Termination in MCTS*. Retrieved from http://www.springer.com/cda/content/document/cda_downloaddocument/9783319279916-c2.pdf?SGWID=0-0-45-1545168-p177846880.

[4] Helmbold, D. and Parker-Wood, A. (2009). *All-Moves-As-First Heuristics in Monte-Carlo Go*. Retrieved from https://users.soe.ucsc.edu/~dph/mypubs/AMAFpaperWithRef.pdf.

[5] Gomez, F. and Miikkulainen, R. (1996). *Incremental Evolution of Complex General Behavior*. Retrieved from http://www.cs.utexas.edu/~ai-lab/pubs/gomez.adaptive-behavior.pdf.

[6] Cazenave, T. *Generalized Rapid Action Value Estimation*. Retrieved from http://www.lamsade.dauphine.fr/~cazenave/papers/grave.pdf.

[7] Gelly, S. and Silver, D. (2011). *Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go*. Retrieved from http://www0.cs.ucl.ac.uk/staff/d.silver/web/Publications_files/mcrave.pdf.

[8] Browne, C., Powley, E. , Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P. Tavener, S. Samothrakis, S., and Colton, S. (2012). *A Survey of Monte Carlo Tree Search Methods*. Retrieved from http://www.cameronius.com/cv/mcts-survey-master.pdf.

[9] Lifshitz, E. and Tsurel, D. (2013). *AI Approaches to Ultimate Tic-Tac-Toe*. Retrieved from `http://www.cs.huji.ac.il/~ai/projects/2013/UlitmateTic-Tac-Toe/`.

Appendix

https://github.com/KaYBlitZ/uttt-engine

https://github.com/KaYBlitZ/UTTTBots

https://github.com/KaYBlitZ/UTTT_CMA-ES