AI Approaches to Ultimate Tic-Tac-Toe

Eytan Lifshitz
CS Department
Hebrew University of Jerusalem, Israel

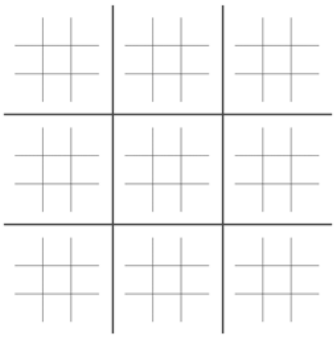
David Tsurel
CS Department
Hebrew University of Jerusalem, Israel

I. Introduction

This report is submitted as a final requirement for the Artificial Intelligence course. We researched AI approaches to the game of Ultimate Tic-Tac-Toe, including aspects of game trees, heuristics, pruning, time, memory, and learning.

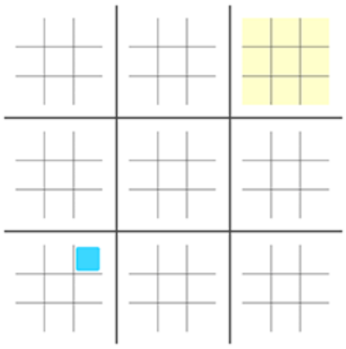
II. Rules

Ultimate Tic-Tac-Toe is a variation of Tic-Tac-Toe which is more challenging than regular Tic-Tac-Toe for a computer. The board consists of 9 small 3-by-3 boards, which together compose a large 3-by-3 board.



Players take alternating turns, and a player wins a small board just like regular Tic-Tac-Toe, by placing three of his symbols in a row. When a player wins a small board, they put their symbol in its position in the large board. The game ends when one of the players gets three symbols in a row in the large board, or in a tie if all squares have been exhausted.

To make gameplay more interesting, a player must place their symbol in the small board that corresponds to the position in the small board of the previous move:



In this example, the blue player played the top-right square in the bottom-left small grid, so the red player must now play his turn in the top-right board (colored in yellow).

If all squares in the small board have been exhausted, or if the board has already been won, the player may choose to make his move anywhere on the board.

III. Evaluation

To evaluate the different agents used in the project, our main statistical tool will be the binomial test. Because it is an exact statistical test of the statistical significance, we can get accurate p-values and not just approximations.[1] In each evaluation of two agents, we will present *n* (number of games played), the percentage of games won by each player, and the p-value of the results. The starting player will be chosen randomly for each game. Our null hypothesis will usually be that the players have equal strength and will therefore win an equal number of games. We use the two-tailed variation of the binomial test to test if either player is better. Tied games are counted as a half-win for each player.

In other cases, we will want to test the hypothesis that two players have the same strength (e.g. minimax and alpha-beta), so our null hypothesis will be that they are unequally proportioned. We will use a proportion test with a confidence level of 0.95, and accept if the p-value of Pearson's chi-squared test statistic for 1 degree of freedom is larger than the commonly accepted threshold of 0.05.[2]

IV. Tree-Traversal Algorithms

We used three basic search algorithms to traverse the game tree: the Minimax algorithm to traverse the whole tree, the Minimax algorithm with alpha-beta pruning to decrease the number of paths observed by the algorithm, and the Expectimax algorithm, that takes the expectation of the other player's values instead of their minimum. (We used the Berkeley AI course assignments as a basis for several stages of the project.)

The game tree is too deep for these algorithms to traverse in reasonable time, so heuristics are used to approximate the value of nodes of a certain depth (described in the next section). Depth is denoted using the Berkeley notation, where a single search ply is considered to be a move for each player.

A random agent was easily defeated even by the simplest agents. A very basic Alpha-Beta agent using the simplest heuristic (heur1) and a depth of 2 won 96.35% of games (n=10,000; p-value<10^{-320}) against a random agent. An Alpha-Beta agent using the heur3 and a depth of 2 won 99.4% of games (n=1,000; p-value<10^{-280}) against a random agent.

Alpha-beta agents should return identical results to minimax agents, only faster. Indeed, when comparing an alpha-beta agent to a minimax with the same heuristic and depth, the results were 49.6% wins for the minimax agent (n=1,000; χ2=0.098; p-value=0.7542), so the agents are indeed similar in strength. Their speeds were quite different – when comparing agents of depth 2, the minimax agent took an average of 1,033 milliseconds to make a move, against just 55 milliseconds for an alpha-beta agent (n=10).

Expectimax agents should be better against random agents, but since our minimax agents already defeat random agents, this is not such a big advantage. Indeed, when running an expectimax agent against a random agent, it won 100% of the time (n=100; p-value<10^{-30}), better than minimax agents which also tied and lost some games. When testing an expectimax agent against a minimax agent with depth of 2 and heur3, the minimax agent won convincingly with 78% of games (n=100; p-value<10^{-7}).

How important is search depth? An alpha-beta agent of depth 2 won 79.8% of games against an alpha-beta agent of depth 1 (n=1,000; p-value<10^{-80}).

An alpha-beta agent of depth 3 won 69.5% of games played against a player of depth 2 (n=100; p-value<10^{-4}) and 75.5% of games played against depth 1 (n=100; p-value<10^{-6}).

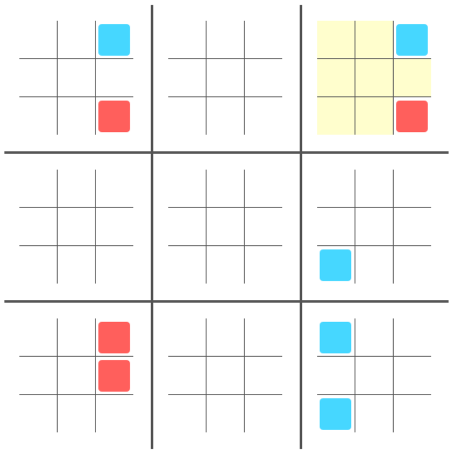
V. Heuristics

We used several heuristics:

1) Our simple heuristic (heur1) evaluates winning and losing with high absolute values: 10,000 for a winning position and -10,000 for a losing position. In all other positions, the score is the number of small boards won minus the number of small boards lost.

2) Our second heuristic (heur2) takes into consideration many more features of the given board: small board wins add 5 points, winning the center board adds 10, winning a corner board adds 3, getting a center square in any small board is worth 3, and getting a square in the center board is worth 3. Two board wins which can be continued for a winning sequence (i.e. they are in a row, column or diagonal without an interfering win for the other player in the third board of the sequence) are worth 4 points, and a similar sequence inside a small board is worth 2 points. A symmetric negative score is given if the other player has these features.

3) Our third heuristic (heur3) uses the fact that there are only 3^9 = 19683 possible configurations for any small board. In the following board, for example, the top-left small board and the top-right small board are identical and will be evaluated with the same score.



We saved time by memoizing values for all possible small board configurations and using these values in our heuristic. This shortened the time spent on the heuristic considerably, from 68.243 microseconds on average on the second heuristic to 33.754 on the third heuristic.

4) Our fourth heuristic (heur4) builds upon the previous heuristic, but takes advantage of an additional feature: if you are sent to a small board that is full or won you can play anywhere, so that add 2 points to the heuristic (and -2 for the other player).

The weights of different features of these heuristics were chosen somewhat arbitrarily. We will return to these weights when dealing with learning agents.

We tested the heuristics one against the other (heur2 was not tested here, since it is similar to the heur3, only slower). The third heuristic proved to outweigh the simplistic first heuristic, winning 84.05% of games when playing with a depth of 2 (n=1,000; p-value<10^{-110}). heur4 beat heur3 in 50.25% of the games, which means it is not significantly better (n=1,000; χ2=0.032; p-value=0.858).

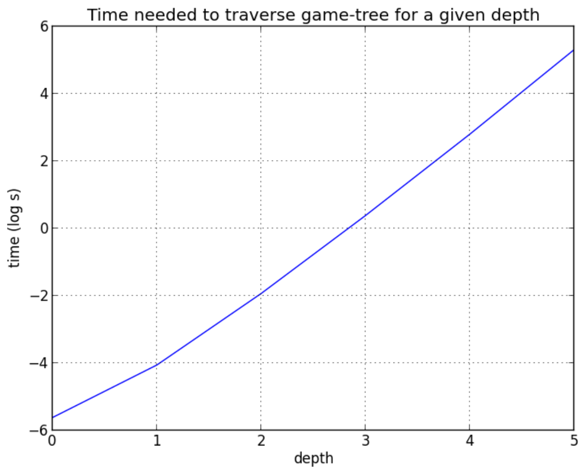
VI. Time

So far we used a constant depth for our tree-traversing agents. But the importance of depth changes between different stages of the game. In the early stages of the game, the search space is still quite large. In the endgame, the search space is considerably smaller, and the importance of moves for a winning result is much more critical. For comparison, a minimax agent using depth=3 will need to examine $81*9^6$ = 43,046,721 positions, while a minimax agent using depth=5 will need to examine just 11! = 39,916,800 positions in an endgame with 11 empty squares (and probably less, since many of the positions are terminal states or forced moves).

We created a family of agents that run a bounded amount of time for each move. A time-bounded agent evaluates a move with an increasing depth (starting with 1), returning the action which was deemed the best in the deepest level he managed to reach before his time ended. The agent also stops if it reached the bottom of the game tree.

We first checked a mininax agent against an alpha-beta agent. As we have previous noted, an alpha-beta agent is identical to a minimax agent when they run for a fixed depth, but takes considerably less time to reach the same result. We now use this extra time to advance to deeper levels of the game tree. Both agents were given 0.1 seconds to make a move, using heur3. The alpha-beta used his speed to win 71.15% of games (*n*=1,000; p-value<10^{-40}).

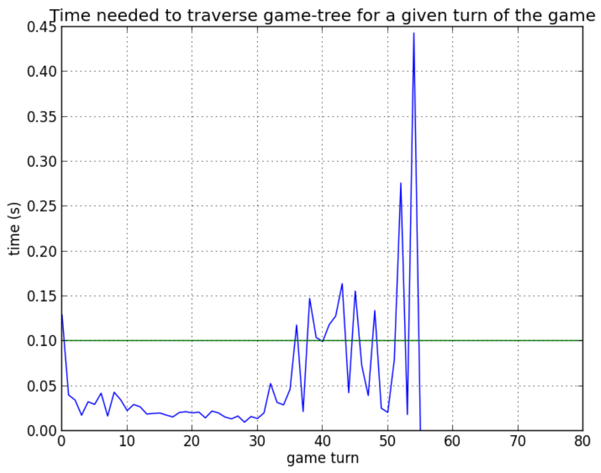
Speed is important, but since the number of nodes grows exponentially with depth, the time needed to traverse the tree also grows exponentially with depth. Below is a log-scale graph for the time needed for an alpha-beta agent to traverse a given depth.



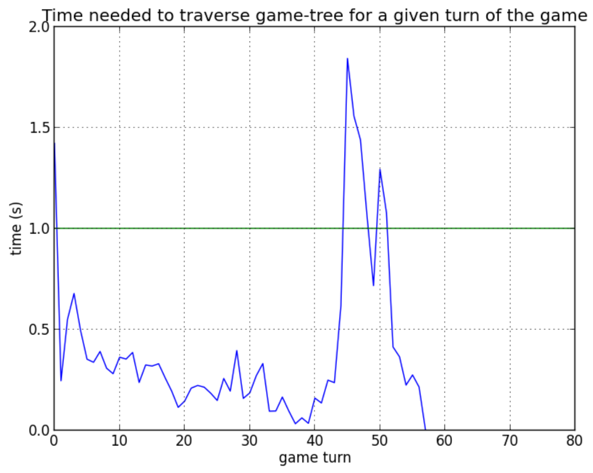
This implies that time is important only if the agent passes the threshold needed to fully compute the next level.

We tested the strength of a time agent against a constant depth agent. Both agents were alpha-beta agents using heur3. Below is a graph for the amount of time given to the timed agent plotted against its win rate.

This graph shows the amount of time used by a constant depth of 2 agent (blue line) plotted against the ordinal number of the turn currently played. The horizontal green line is used as a reference for a constant time agent – if the line is above the the blue line then the constant time player would have also succeeded in evaluating the move for depth 2, while if the green line is below the the blue line, it would have to return the evaluation of a shallower lever.

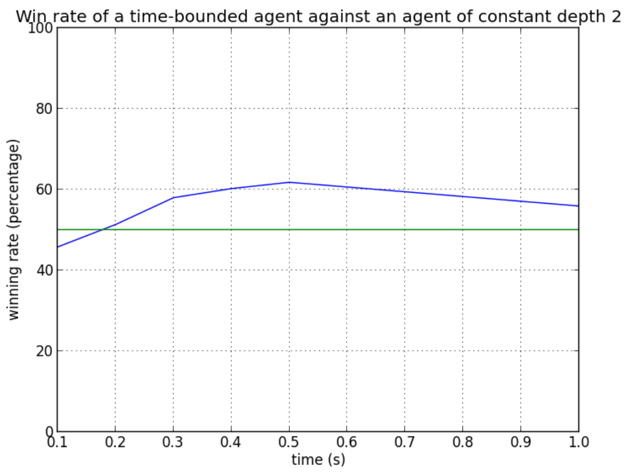


This is the graph for a player of depth 3:

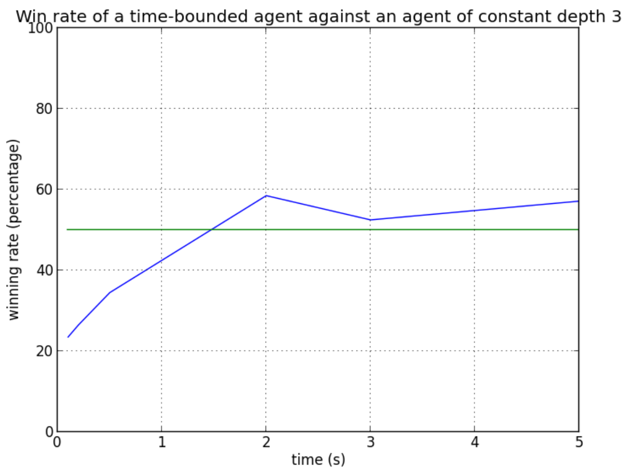


We can see that the first move takes a long time to compute, since there are 81 possible moves. We see a general trend of decline in the time needed until about 30-40 moves in, then a leap, probably because boards are starting to be filled up and players are sent to them, meaning they can play anywhere.

We tested the time agent against a constant depth agent. This is the graph for the winning odds of a time-bounded agent, plotted against the amount of time it was given. Here is the graph when playing against a player of constant depth 2:

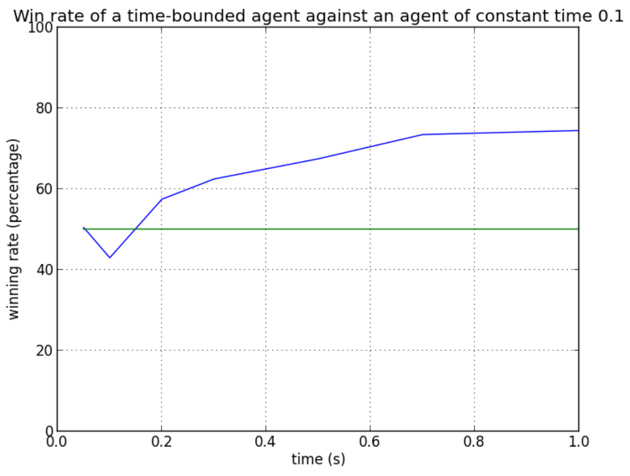


Here is the graph of a time-bounded player against a player of constant depth 3:



We can see the agent getting better with more time, surpassing the constant depth agent somewhere between t=1 and t=2. Adding more time after that does not change the outcome significantly, due to the exponential time gap between depth=3 and depth=4.

How much does adding time help against another time bounded player? We checked several timed agents against an agent with 0.1 seconds.



Again, we see that more time helps an agent win more games, but due to the exponential time gap between levels, the slope levels off.

In conclusion, time-bounded agents have the advantage of being able to quickly compute many levels of the tree, which is especially important in endgame positions. They do have their drawbacks, however, since they aren't as efficient in positions which do require a large computational effort, which are the positions where the player is not limited to a single small board. They are also wasteful in the sense that the computational power is wasted if the timeout happened before the level was completed. Due to the exponential gap between levels, this can be quite a considerable proportion of the time.

VII. Memory

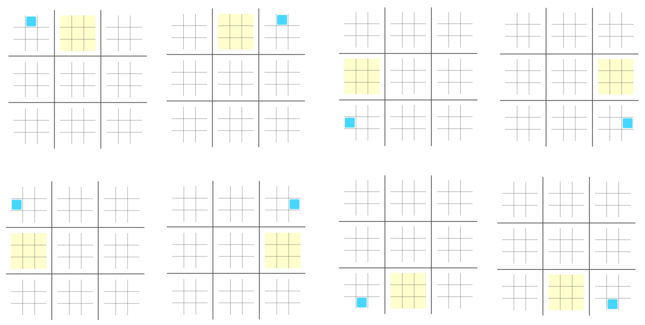
We have previously used memoization in the construction of heur3, keeping the values of small boards. We will now build agents using transposition tables, saving time instead of recomputing positions that were already evaluated.

Our first agent is a basic one, which just saves combinations of boards and depths in a transposition table, and checks every position it gets to see if it has already been evaluated. An infinite memory is unpractical (and after about 200,000,000 boards the computer becomes unresponsive), so we limited the agents memory size.

We tested this agent with memory of 10^6 boards against the memory-less time-bounded agent. The results were quite similar for both agents: 50.5% (n=100, p-value≈1), slightly in favor of the memory agent.

This agent is not very efficient in his memory usage, since it remembers all positions it has encountered, but many of them are very rare and have a low probability of appearing again. We therefore created a new agent that works in a FIFO method, discarding the last used position once it encounters a new position and it has reached the full capacity of memory allocated. Again, we tested this agent with memory of 10^6 boards against the memory-less time-bounded agent. The results were again very similar: 48% (n=100) for the memory agent.

We also tried to use the symmetry of rotation and reflection. For example, the following eight boards are equivalent, and should therefore have equal heuristic value.



This should help both in terms of time (we don't have to reevaluate the same board eight times) and in terms of memory (we only have to remember one representative of the group). When evaluating a position, the agent will check all 8 possible rotations and reflection to see if their values were already evaluated.

In practice, however, this did not work well. The overhead caused by hashing 8 different boards for every node of the tree took toll on the limited time, and when running a symmetry agent against a memory-less agent, the symmetry agent won only 43.5% of games (*n*=100, p-value>0.19).

Our memory agents until now computed the hash separately for each board. The changes between boards are local and limited to adding a single symbol (or removing it, when returning upwards in a tree search). To speed up the hashing process, we tried a different hashing technique. Zobrist hashing generates random integers in the range [0, 2^{256}] for each possible element of the board: 81 squares of the board × 2 possible symbols, one for each agent. The hash function of a board is the bitwise xor of all elements. The random integers are large enough for hash collisions to be probabilistically improbable. Updating the board between two consecutive states is a matter of a single xor, making it much faster than recomputing the whole board. In practice, however, this agent produced only a negligible advantage over a memory-less agent, winning 52.5% of games (*n*=100, p-value>0.76).

In conclusion, using transposition tables did not prove to be advantageous. This is probably due to the exponential number of possible boards growing in each turn.

VIII. Learning

Our next family of agents uses learning to try and improve performance. Our main direction is reinforcement learning, and Q-learning in particular.

Our basic attempt was a learner for state-action pairs. We initialized a dictionary for every pair. After every move, we updated the value of the pair using the difference between the values of the two consecutive positions, multiplied by some alpha value and a discount value. The score of a given position was defined as 500 for a winning position and -500 for a losing position. In all other positions, the score is the number of small boards won minus the number of small boards lost. (This is the same as heur1). We divided the games into two stages – in the first half the player plays a random move with a certain probability, and chooses the best stat-action pair with the complement probability. In the second half the player always issues his policy.

The results were disastrous – this player won only 0.5% of games against a regular alpha-beta agent (*n*=1,000, p-value<10^{280}). This is due to insufficient feedback – only rarely will a player visit a state twice, and so the q-values are not able to propagate properly.

We then tried learning with features. Instead of a dictionary with action-state pairs, we learn features of advantageous states, and these features can be shared by many states.

We initialized a weight vector for many features, previously described in the heuristics section (heur2). This time, we updated the weight vector using the difference between state scores. However, this agent also failed miserably against an alpha-beta agent. Again, it won just 0.4% of games (*n*=1,000, p-value<10^{280}).

It turns out that single transition Q-learning isn't fit for this game, since the rewards are so scarce the agent cannot learn in reasonable time. In contrast to games like Pacman, where rewards are frequent enough so the player has a chance to learn the correct weights for the features.

Further improvements are possible. It's possible to remember the whole history of a game, then update the feature vector at the end of the game with all moves played during the game, not just the last one. It's also possible to learn values for all possible binary feature vectors. We did not implement these agents, but conjecture they will show better performance.

IX. Should you go first or second?

We can also use the agents developed to determine other questions about the nature of the game.

In game theory, Zermelo's theorem states that for every deterministic two-player game with perfect information, either one of the players has a winning strategy, or both players have strategies that guarantee a tie. The theorem only proves the existence of a strategy but does not include a constructive proof.

Some games have been analyzed and such strategies have been found. Regular Tic-Tac-Toe, for example, is a tied game if both players play optimally. Other games, like Chess, are harder to analyze. Does Ultimate Tic-Tac-Toe have a winning strategy? Is it better to go first or second? We cannot give an analytical model, but we can try to find a statistical answer. We ran two identical alpha-beta time-bounded agents (0.1 seconds), and tested which player has a better chance of winning. The first player won 56.17% of games (*n*=300, p-value<0.05), meaning he has a statistically significant advantage.

[1] https://en.wikipedia.org/wiki/Binomial_test

[2] Test of Equal or Given Proportions, http://stat.ethz.ch/R-manual/R-patched/library/stats/html/prop.test.html

[3] XKCD: Tic-Tac-Toe http://xkcd.com/832/