AI agent for Ultimate Tic Tac Toe Game

Artificial Intelligence final poject – winter 2013/14

Eran Amar
eran.amar@mail.huji.ac.il

Adi Ben Binyamin
adi.benbinyami@mail.huji.ac.il

***Abstract* — in this paper we describe our final project of AI course at the Hebrew University (winter 2013/14). Our project is focused on Adversarial Search, more specifically, Minimax and possible optimizations of that algorithm and the "cost" of those optimizations.**

Keywords—adversarial search; artificial intelligence; minimax; ultimate tic tac toe;

I. Description of the Game and Rules

Ultimate Tic Tac Toe (*UTTT*) is a nested version of the regular Tic Tac Toe (*TTT*) game. It is a two-player zero-sum game, which includes a matrix of 3x3 boards of TTT.

A square in the UTTT board (one of the nine TTT boards in the matrix) is called a *mini-board*, whereas a square inside each mini-board (in which a player places its move) is called a *cell*. To win a game, a player must win three mini boards in a row. Each player may only play inside a mini-board that corresponds to the cell that its opponent played last turn. The player who starts the game, is free to choose any cell. In addition, when a player is sent to a mini-board that is already decided (and therefore all its cells are "disabled") may choose any free cell in any non-decided mini-board, such a move called a *wildcard*.

II. The Goal of the Project

The game of UTTT is rather hard. Since the game consists of nine different mini-games of TTT which are being played in a parallel manner, a player of the game must take into consideration all of the different mini-boards when choosing his move. Not only that, but from the nature of choosing the mini-board a player must play in, he must also consider the long-term effects of his move, not only the immediate one. And we must not forget that the goal is achieving a row of won-mini-boards.

As we can see, this is a lot to consider for a human player, but not necessary for a computer. We thought of putting the methods learned from "Introduction to Artificial Intelligence" course to use in order to create computer agents that would be able to "consider" all of the different aspects of the game.

III. Our Approach to Solving the Game

The UTTT game fits perfectly to the Adversarial Search section from the course; therefore our project is focused on exploring and expanding that part of the course.

The main approach we took when designing the UTTT agent is the Minimax search. As we also wanted the agent to play "fast enough", we chose to combine some ideas from the technique of Simulated Annealing into a Minimax agent (we will discuss the different agents and methods on Chapter VII).

The main target of our project is to build an ultimate UTTT player that will have both high winning rate and runs in a reasonable amount of time.

To achieve that, we must achieve two main milestones in the implementation process: first, is the inner evaluation function uses to guide the search. Second, custom optimizations on the searching algorithm, some of them were learned in the course and some of them are unique ideas come during the process of developing the agent.

Specific details will follow, right after we explained some of the difficulties in solving a UTTT game.

IV. Why UTTT is difficult?

A. Minimum depth of the game tree

Consider minimal number of turns it takes to win a game: a player must fill at least three mini boards, each with at least three cells - total of nine cells in the UTTT board. Assuming that the opponent will not distract us from forming our win (which is a very simplifying assumption) and that we playing first, we need nine moves for ourselves and eight moves for our (naive) opponent. That is adds up to a minimum of seventeen moves.

Note that a tie in the game might be the worst case scenario. Assuming that no player win in any mini-board, all the eighty-one cells on the board are full and that will take eighty-one moves exactly.

B. Exploding branching factor

We will prove that the minimal average branching factor (ABF) possible for a UTTT game if greater or equals to 5: First, note that the highest branching factor possible for a mini board is 9 (for an empty mini board), and the minimal possible is 1 (not zero because we might visit each mini board at most 9 times, therefore we always have at least one available cell, besides the last turn in a game which ends with a tie). Because we are looking for a lower bound on the ABF, we are allowed to count any wildcard's branching factor as the number of unoccupied cells in the decided mini board that cause that wildcard (i.e. *ignore* the wildcard - this is a very simplified assumption that may only lower the bound for ABF). In addition, we will ignore the branching factor of the first turn in the game (which is always 81) because our agent might not be the first player on the game.

Assume that any mini board have a unique Id number from 1 to 9. Denote s as the full history of branching factors from a valid game sequence (we always talk about valid games). Meaning, s is a series of $(miniB\_id, bf)_i$ whereas i is the turn's index (on the first turn $i = 0$) and bf is the *branching factor* of the only $miniB\_id$ that was legal in that specific turn (well defined when ignoring wildcards, as we did). The ABF for a given series s is calculated by the formula:

$$\text{I.} \qquad ABF(s) = \frac{1}{|s|}\left(\sum_{(miniB,bf)\in s} bf\right)$$

Another way to calculate the ABF is by weighted average. Given a series of branching factor s, for all $k \in [9]$ denote:

$$MB_k = \{(miniB\_id, bf)_i \in s \mid miniB\_id = k\}$$

Meaning, MB_k is the sub-series of s filtered only to the k^{th} mini board. And therefore:

$$\text{II.} \qquad ABF(s) = \sum_{k=1}^{9} ABF(MB_k)\left(\frac{|MB_k|}{|s|}\right)$$

If there some mini-board id t with no elements in s, then $MB_t = \emptyset$, and the formula is well defined. From now on, we will work only with formula number II.

Key observation: For any series s (from a valid game), and $t \in [9]$, if $(t, x) \in s$ then $(t, x') \in s$ for all x' such that $x \le x' \le 9$. That because, to reach a branching factor of x in mini board t, we (and\or our opponent) must have decrease the branching factor one-by-one from 9 down to t, by playing moves in that mini board.

Therefore, for any mini board t appeared in s, with x as its minimal branching factor for that mini board in s (which is not lower than 1), then MB_t must be in the form $(t, 9), (t, 8), \dots, (t, x)$. Meaning that:

$$ABF(MB_t) = \frac{1}{|MB_t|}\left(\sum_{(t,bf)\in MB_t} bf\right) =$$

$$= \frac{1}{9-(x-1)}\left(\sum_{k\in[9]\setminus[x-1]} k\right) =$$

$$= \frac{1}{9-(x-1)}\left((9-(x-1))\frac{9+x}{2}\right) =$$

$$= \frac{9+x}{2} \ge 5$$

Assigning that in formula II and we get:

$$ABF(s) = \sum_{k=1}^{9} ABF(MB_k)\left(\frac{|MB_k|}{|s|}\right) \overset{*}{\ge} \sum_{k=1}^{9} 5\left(\frac{|MB_k|}{|s|}\right) =$$

$$= 5\sum_{k=1}^{9}\frac{|MB_k|}{|s|} = 5\frac{|s|}{|s|} = 5$$

$$\Rightarrow ABF(s) \ge 5$$

* For any mini board k not in s, $|MB_k| = 0$, therefore we still allowed to replace $ABF(MB_k)$ with 5.

Note that the proof examines the situation of no wildcards, but as for the branching factor, wildcards doesn't help either: Taking into account that when a player sends its opponent to an already-decided mini-board, that opponent may choose any available cell on the UTTT board - the branching factor of that turn could easily reach 25 (approximately five available cell on approximately five available mini boards).

Combined together, a search-without-optimization approach will reach a tremendous number of search-states in any time the agent is required to decide on a single move (and that is not less than 17 times, as showed above). More specifically, denote $b = 5$ as the average branching factor, $d = 17$ the minimal depth of the game tree (which must be end with a winning of one of the players because tie requires much more turns and can reach depth of 81). Let i be the sequence number of the current move, starting from $i = 1$ and ends at $i = d$. The amount of calculation that classical search algorithm such as Minimax will do during one game is equal to $\sum_{i=0}^{d}(b^{d-i})$ and that is equivalent to $\frac{(b^{d+1}-1)}{b-1}$. Assigning the minimal possible values mentioned above will reach $O(b^d) \sim 5^{17} \sim 2^{40}$ calculations. When considering the worst case scenario, the maximal depth of the tree is 81 and the average branching factor is 5, that sum up to $\sim 2^{188}$ calculations for a single game.

On Chapter VIII we will demonstrate running time of Minimax agent with Alpha-Beta pruning as an optimization mechanism, and that will convince us that more creative optimizations are required.

V. The Characteristics were considered and Pitfalls

Two guiding parameters were considered during the developing process: average running time for a single move, and the winning-rate. A good agent, as we see it, is an agent who wins at last 80% of games he participated in. We define a reasonable average running time as any amount of time that is less than a half of second. When focusing only on the winning rate factor, Minimax algorithm is the best solution (assuming we already have a good evaluation function), because it looks ahead multiple turns and chooses the best action on the long run under any possible situation. Therefore, it guarantees us the best possible result from the chosen move even on the worse case.

There are two key pitfalls regarding that approach:

First of all, we must have a good evaluation function - which is always not so easy, and also in nested TTT. Second, assuming that we already have a good evaluation function, the quality of Minimax algorithm depends on the depth it "looks" into the future - how we choose the correct depth?

One might think that running Minimax with an unlimited depth (until the game ends) is a worthwhile solution. It may be true when considering only the winning-rate parameter, but when taking the time resource into account - it is absolutely not. Let's see why:

Minimax running time is $O(b^m)$, and the average branching factor is 5, and the minimal game sequence is 17 moves (the minimal depth of the game tree), then the cost of one call to the Minimax algorithm will takes *at last*

$\sim 2^{40}$ calculations. This will result in awful running time of the agent.

Therefore, we wanted to solve this trade off in winning-rate vs. running time, which led us to taking different approaches, which will be represented later in Chapter VII.

VI. Evaluation Functions

We implemented three evaluation functions for assessing a given state of an UTTT board:

A. *CellsWeightHeu:*

This approach considers the existing close-strategy (an optimal policy) in the regular TTT game. It looks on the given mini-board and estimate the board's desirability according to the last move played.

1) If the last move played in the given mini-board was a winning move, then this board gets the highest score possible.
2) Otherwise, if the last move blocked the opponent's immediate win (i.e. the opponent had 2 cells in a row, and the move was in the third cell in that row), then the state gets some lesser value, but still high one.
3) Otherwise, the state is ranked according to a weight function we created. This weight function captures the desirability of a given cell, and the heuristic returns the sum of the weights for the player's cells. Obviously, we will prefer a cell which is a part of the most winning rows possible (i.e. the center cell is the top ranked one, then the corner cells, and lastly the cells from the adjacent squares to the center cell), which is achieved by us with this functions.

B. *RecursiveWeightHeu:*

This is a more complex evaluation function. The first thing is to ask: is the UTTT game has a winner? If so, we get some constant large positive score if the winner is us and constant large negative score if the winner is our opponent. When the board is decided as a tie we get neutral score. Otherwise, the heuristic is recursively called for each mini board, the returns heuristic value is the sum of the values for each of the mini board, multiplied by the weight of the mini board itself.

The weight of the mini board is the number of winning rows which he is a part of (i.e. the center board's weight is 4, 3 for each of the corners, the 2 for the rest).

The evaluation of a mini board is done in the following manner:

1) If the board is won by us, we return the sum of all cell's weights, which is 24.
2) If the mini board is won by opponent, we return -24.
3) If the board is with tie, then it gets a zero.
4) Otherwise, we add each of the cells of the mini boards' weight if it is owned by us, subtract its weight if it is owned by the opponent. An empty cell gets no score.

This evaluation function is overall good. It clearly prefers winning and not losing immediately, and it performs a sophisticated evaluation for the other cases.

C. *WinningPossibilitiesHeu:*

This evaluation function suggests an improvement over the Recursive Weight Heuristic. The evaluation is done in a similar manner: if the board is won by us we get the highest score, if it is won by the opponent we get the lowest score, and zero for a tie. For the other options the calculation is more sophisticated.

First, we evaluate the big UTTT board as if it was a TTT board itself. The evaluation will be explained soon. The value of the UTTT as a TTT is multiplied by a weight we have given to it, as it is preferred to win the UTTT board over winning a mini board. To this value we add the value of the evaluation function for each mini board that is still playable (not full or already decided).

The value of a mini board is determined in the following way: If the mini board is not playable, it gets no value. Otherwise, we estimated the winning possibilities of the board. The key improvement over the previous function is that here we take into account the fact that some of the cells in the winning possibility are already occupied by our opponent; hence, there is no reason to highly compensate them. For example, consider the middle cell in a mini-board, and let's assume that it is occupied by our player. That cell's initial value is 4 because there are four possible winning-rows that cross that cell. But, if our opponent occupied one of the cells from the corners of the mini-board, the middle-cell's value is decreased by one, because there is one less possible winning-row for our player, which uses that middle cell.

VII. Agent's Structure and Optimizations

The agents are composed from two base layers, and another optional layer: First is the "search engine" which is the Minimax algorithm with Alpha-Beta pruning (as a basic optimization). Then, there is the specific evaluation function used by the Minimax (i.e. Alpha-Beta) search. The third layer apply only to part of the agents - a random jump mechanism which played as a main optimization technique.

The first two layers (search algorithm and evaluation function) used as the basic structure of an agent. We started by implementing 3 Minimax agents (from now on, any reference to Minimax means Minimax with Alpha-Beta pruning), one for each evaluation function, and testing their winning rate and average time per move.

The searching depth of all the agents were fix to 2. Less than that we miss the point of Minimax looking moves ahead, more than that reach running time way beyond 1 second per move (as we saw during the developing process, and decided to decrease the depth).

In reality, even with search depth of 2 (and Alpha-Beta pruning) all of the three agents reaches average move of more than half a second, and that is not a reasonable

amount of time - different optimization techniques is needed.

Our first suggestion was to alter the inner "searching engine" from Minimax search to greedy search, meaning, choosing the most promising action (according to the same evaluation functions) in the current state and do not look ahead.

Our thought was, that we may play greedily (and independently) in each mini-board without consider the consequences on the other mini-boards. In other words, we wanted to check if playing with an "optimal policy" locally (which is the base idea behind evaluation function A) will result with an optimal policy globally.

Another optimization, inspired from the sub-field of local searches, is to combine Simulated Annealing mechanism with the Minimax algorithm. More specifically, defines a "temperature" variable which is getting "colder" as long as we get deeper in the game (passes more turns). Each time our agent should decides on a move, it flip a coin with a probability related to the current temperature, and according to the result takes a random jump (i.e. random action) OR searches with the Minimax algorithm. The colder the temperature, the less probability for a random jump. The rationale behind that mechanism is, that at the beginning of the game the agent's choices are less critical than choices made deeper in the game, therefore, we allowed random moves with higher probability on earlier stage. Our hypothesis is, that such a random jumps with decreasing probability will not impair the winning rate severely, and on the same time improves the average time per move dramatically. We explored that hypothesis on the next chapter.

VIII. Experiments and Results

All in all, we designed 11 different agents:

- **Reflex Agent** - which chooses always the first legal action from the list of legal actions (prefers free cells from left to right and from top to bottom).
- **Random Agent** - which chooses random action from the list of legal actions.
- Three different **Greedy Agents**, each of them choose the best current action according to one of the evaluation functions discussed in Chapter VI.
- Three regular **Minimax** (Alpha-Beta) agents with depth-limit of 2. Each agent uses one of the three evaluation functions discussed in Chapter VI.
- For each regular-Minimax agent, we constructed a version with **Random Jump** mechanism. These agents are using coin-flip to choose between their Minimax strategy and a random action (the actions a chosen with uniform probability).

To test our agents, we run a batch of 100 matches between any possible combinations of two agents, except running an agent against himself. Since there may be some hidden advantage for the player starting in the game, for each two agents a_1 and a_2, we run both the batch for a_1 against a_2, and a_2 against a_1.

There are total of 11×10 different game possibilities, and for each we run a session of 100 independent games.

We wanted to develop an objective way of comparing our agents' performance. For that, we used the following calculation:

For the Winning Rate calculation, we counted the total number of games won by an agent, and divided it by the total number of games he participated in. To eliminate the effect of the very few sessions who ended in a tie, these games were rematches (while completely discarding the information gained from the tie match).

Since each agent participated in 2000 different games (10 sessions where he is the first player and 10 in which he is the second, each sessions consists of 100 different independent matches). In case you wondering if "luck" distort the results of part of the games, The Law of Large Numbers promise us that the averages will still converge towards the expected value of the results (that is one reason we chose large sessions).

For the *Running Time* measurement, we set a timer to start before each of the agent's turns, and turn it off immediately after the agent chose his next move. We then divided the time by the total number of turns the agent had in these games.

After obtaining these objective measurements, we can easily compare the agents.

In addition, we explored different "decreasing temperature" function for the random jump mechanism, and finally chose $f(t) = \frac{5}{t^2}$, starting with $t = 3$, and increments t by 0.1 for each turn. This function achieved high value for smalls t i.e. $f(t) \sim 0.55$, and decrease slowly (not linearly) to $f(t) \sim 0.20$ for later moves (i.e. around turn No.20).

Results

As we thought, the Minimax based agents perform quite well on the Winning Rate scale, and they are pretty bad on the Running Time scale. With an average Turn Time of 1.009 sec (between all three different Minimax agents), and average Winning Rate of 75% of the games played.

In Table I there is a brief summary of the Winning Rate and Run Time of the different agents. Each agent is labeled with a prefix of `MM` or `G` which represents the Minimax search-engine and the Greedy search-engine accordingly. That prefix is followed by the name of the evaluation function, as shown at Chapter VI, then, there will be a `#R` suffix but only for agents with the Random Jump mechanism.The only exceptions are the reflex agent and the random agent.

TABLE I. SUMMERY OF THE RESULTS

Agent Name	Winning Rate (%)	Average Move (ms)
MM_Winning	92.562708	1515.329
MM_Winning#R	85.378563	816.9631
MM_Recursive	79.045149	897.0043
MM_Recursive#R	70.149016	496.6588
MM_Cells	58.726002	616.6018
MM_Cells#R	54.405278	392.3125
G_Winning	39.798924	3.32677
reflex	21.810606	0.017565
G_Cells	19.046377	2.298451
G_Recursive	17.353463	2.237743
random	11.723913	0.017117

First, let us discuss the result of the Greedy optimization introduced in the previous chapter. It turns out that Greedy Agent improved the Running Time dramatically, but impaired the winning rate more than we are willing to accept. With a winning rate of ~20% it does not matter much that the average time per turn is ~2.2 ms.

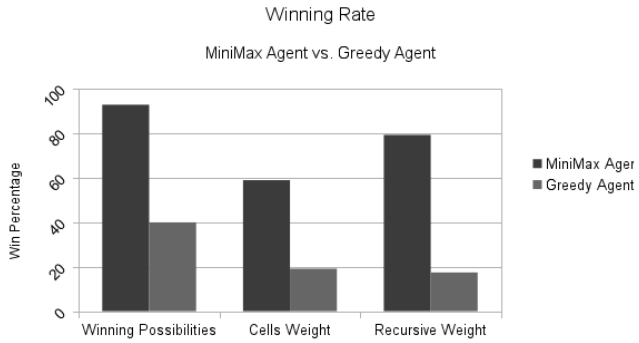

Fig. 1. Winning rate (%) of Basic Minimax vs. Greedy agents, per evaluation function.

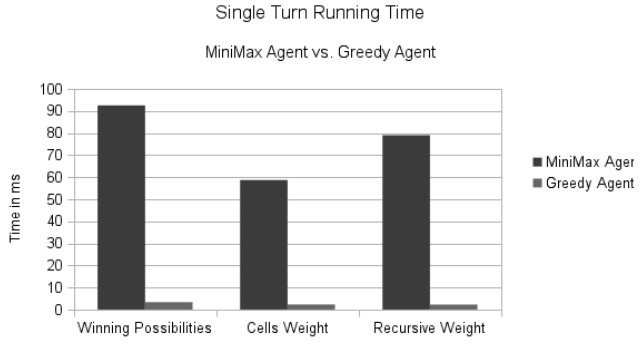

Fig. 2. Average time per move (ms) of Basic Minimax vs. Greedy agents, per evaluation function.

As for the Random Jump optimizations, the results are a bit more promising. We found that the Random Jump mechanism cuts the running time by 42.36% on average, as demonstrated by the following diagram:

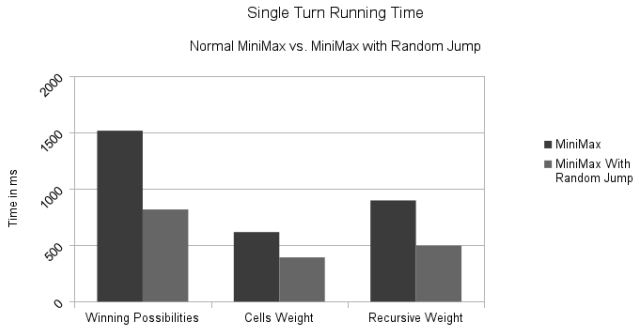

Fig. 3. Average time per move (ms) of Basic Minimax vs. Random-Jump Minimax agent, per evaluation function.

As one can obviously see, the Running Time improved dramatically with our version of the Random Jump Minimax Agent. We are left to wonder - at what cost? Clearly there must be some payoff in the Winning Rate.

From our statistics, it follows that the Random Jump Minimax agents performs on average as little as 8.79% worse than the normal Minimax agent, as this diagram demonstrates:

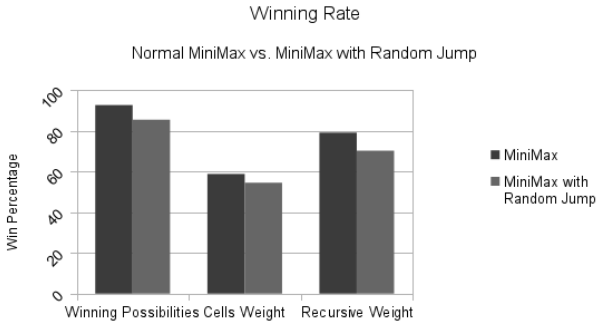

Fig. 4. Winning rate (%) of Basic Minimax vs. Random-Jump Minimax agent, per evaluation function.

To calculate the improvement and the regression of an agents' performance, we normalized the results to his corresponding non-random-jump agent. That is, if the Winning Rate of an agent is w_{normal}, and the Winning rate of the corresponding Random Jump agent is w_{jump}, then the improvement was calculated with the formula $\frac{(w_{normal} - w_{jump})}{w_{normal}}$. The same way of comparing was used also for the Running Time comparison using $\frac{(t_{normal} - t_{jump})}{t_{normal}}$ when t_{normal} is the average time (ms) per move of normal Minimax agent, and t_{jump} is the average time (ms) per move of Minimax with Random-Jump mechanism.

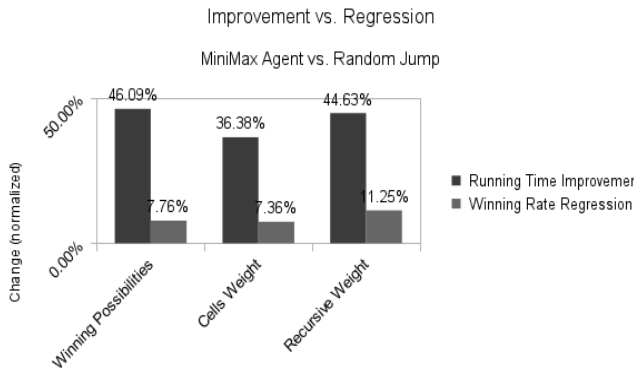

Fig. 5. Improvement (%) in average time for move and regressing (%) in winning rate, Basic Minimax vs. Minimax with Random-Jump, grouped by evaluation function.

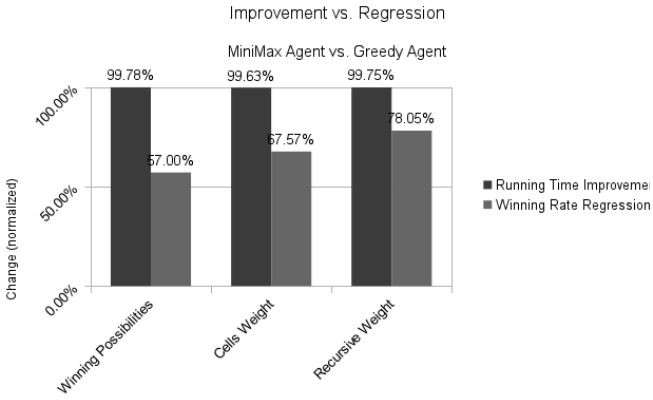

Fig. 6. Improvement (%) in average time for move and regressing (%) in winning rate, Basic Minimax vs. Minimax with Greedy search-engine, grouped by evaluation function.

We are fairly impressed with the results of the Random Jump mechanism, and we were happy that we had the chance of combining different idea learned in the "Intro to AI" course, constructing a unique agent which suggest a fair tradeoff in winning performance vs. time performance.

IX. Conclusions

As we saw in the previous section, *WinningPossibiliesHeu* is the best evaluation function among the implemented evaluation functions, as it achieved the highest winning rate (better than any other agent in both the deterministic-Minimax version and the random-jump version).

Regarding the average running time factor - no agent which played a move in less than half a second was still able to maintain winning rate higher than 80%. Unfortunately, none of the agents we implemented were able to match both criteria of winning rate and the bound of average calculation time per move.

Considering the optimization mechanisms, Random Jump turns out to be a worthwhile technique with better tradeoff between running time improvement and winning rate loss, than the greedy technique. We assumes that random jump works in UTTT, because moves on earlier stages of the game are less critical than any move done ten, even five, turns later. That allows us to make fast-and-random moves on the beginning of the game which improves the average running time dramatically but barely damaged the winning rate.

X. Suggestions for Further Research & Limitations

There are a number of issues we could not explore due to lack of time and the limited scope of this project. First, is implementing a learning agent - it could be very interesting to explore the winning rate of a learning agent which learned against the all the Minimax agents. In such a case we might have to think of another way to measure "average time for move" which consider the learning sessions as well. Second, there is more optimizations and adjustments possible in the already implemented Minimax agents, such as choosing randomly between two different evaluation function (each time the agent should decide on a move), or add a time limit mechanism that will stop exploring the game tree within a predefined amount of time and return the best action found so far. And third, explore the current implementation with different parameters, i.e. experiment other formulas for the "temperature function", and compare its effect on the Random Jump trade off.

Major limitation of our results, target the winning rate factor. The fact that the *Minimax_WinningPossibilites* agents wins 92% of the game does not necessarily means that it is an ultimate opponent, there might be other evaluation function which could achieve higher winning rate, and even constantly overcome the *Minimax_WinningPossibilites*. That should be bear in mind while reviewing the results.

Another thing to consider is our use in Python as a programming language. Since Python is considered to be a "slow" programming language, then it is somehow safe to assume that the same evaluation functions and methods, will achieve much better run time performance while written in C++, for example.

XI. Usage of the Code

1) Download all the code from: https://github.com/amarn/UTTT to a local folder on your computer.
2) Open a command line window (terminal) and locate it inside the folder of the project.
3) Run: `python uttt_game_engine.py <arg1> <arg2>`

Fill the arguments according to Table II. `<arg1>` is the agent for the X player, and `<arg2>` is the agent for the O player.

Note: The project was written in python 2.7 and might not be compatible with python 3.0 or higher versions.

TABLE II. YOU MAY CHOOSE ANY AGENTS FROM THE FOLLOWING OPTIONS (CASE SENSITIVE):

Argument String	Description
human	the user chooses the moves
random	chooses moves randomly
reflex	choose the most top-left available cell
MM_Cells#R	Minimax player (with CellsWeightHeu) and random jump optimization
MM_Cells	Minimax player (with CellsWeightHeu)
MM_Recursive#R	Minimax player (with RecursiveWeightHeu) and random jump optimization
MM_Recursive	Minimax player (with RecursiveWeightHeu)
MM_Winning#R	Minimax player (with WinningPossibilitiesHeu)
MM_Winning	Minimax player (with WinningPossibilitiesHeu) and random jump optimization
G_Cells	Greedy player with CellsWeightHeu
G_Recursive	Greedy player with RecursiveWeightHeu
G_Winning	Greedy player with WinningPossibilitiesHeu

XII. BIBLIOGRAPHY

[1] Artificial Intelligence: A Modern Approach, by Stuart Russell and Peter Norvig, Third Edition, 2010.