Reinforcement Learning Project: Ultimate Tic-Tac-Toe

Lilian Bichot, Julien Bienvenu, Marie Bienvenu, Astrid Nilsson, Josselin Somerville,
https://github.com/JosselinSomervilleRoberts/UltimateTicTacToe-RL.git

***Abstract*—Tic-tac-toe is a relatively easy-to-master game. The same cannot be said for ultimate tic-tac-toe, a board game composed of nine tic-tac-toe boards arranged in a 3 × 3 grid, with some additional challenging rules. We develop here successively several reinforcement learning algorithms, each new one being introduced to overcome some drawbacks of the previous ones.**

I. Introduction

Ultimate tic-tac-toe is a 2-player strategic game which builds upon the concept of regular tic-tac-toe to make it more challenging. It is made of nine tic-tac-toe grids arranged in a 3 x 3 board. Victory is achieved by winning in three grids that are aligned in a horizontal, vertical or diagonal row. The first player puts a mark (X or O) in any of the empty spots. After each move, the opponent has to play in the local grid corresponding to the location of that last move in the last local grid. For example, putting a mark in the top left corner of any small grid sends the opponent in the top left small grid (wherever he can and wants inside this small grid). An additional and crucial rule is that when a local grid is won or filled completely and a player is sent to that grid, he can actually choose the grid in which to play next, instead of being forced to play inside this one.

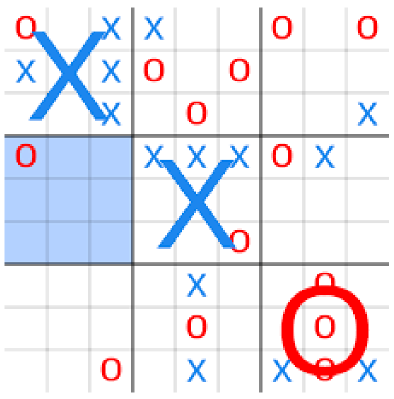


Fig. 1. A game of ultimate tic-tac-toe in which the red player has won one local grid and the blue player has won two

Compared to regular tic-tac-toe, there is no obvious winning strategy. Since it is a finite two-person game of perfect information in which the players move alternately and in which chance does not affect the decision making process, according to Zermelo's theorem [1], like for chess, there theoretically exists a winning strategy for one of the two players or a draw strategy for both of them. But, like for chess, these strategies are not known. Let us note that if we delete the last rule that allows a player to chose another grid if he is sent to an already won grid, we know a simple winning strategy for the first player, as shown in [2].

We thus face a game with no obvious winning strategy and a large state space - it has a size of 3^{81} (81 spots filled with X, O or no mark) - which means we cannot compute the Q-table for such a problem.

To tackle this challenge, a first idea is to use the minimax algorithm (which we will cover later on) as done by Gilbert[1]. More effective approaches usually involve neural networks to infer the best action for each state. Paruthi[2] based his neural network on the AlphaZero algorithm, which can play deterministic fully observable games such as go or chess. He performed a Monte Carlo Tree Search on the tree of states to determine a policy, which he then used for training his network.

Banerjee[3] tried a table-based temporal difference learning based approach, which did not prove very efficient because of the large space state. He observed better results when switching over to a neural network whose seed he based on the table computed previously.

Our approach[4] is the following. We designed our own environment to have two agents (AI or real players) be able to play ultimate tic-tac-toe. We then created different agents and compared their performance empirically. The agents we studied are:

- a random agent
- an agent based on the minimax algorithm (with and without alpha beta pruning)
- an agent using Monte Carlo tree search
- a deep Q-learning agent trained against the agents above

II. Background and Related Work

Minimax [3]

[1]https://github.com/Ian-Gilbert/Ultimate-Tic-Tac-Toe

[2]https://arnavparuthi.medium.com/playing-ultimate-tic-tac-toe-using-reinforcement-learning-892f084f7def

[3]https://medium.com/@shayak$_8$9588$/playing - ultimate - tic - tac - toe - with - reinforcement - learning - 7bea5b9d7252$

[4]Our code can be found here: https://github.com/JosselinSomervilleRoberts/UltimateTicTacToe-RL

A recommended agent for our problem is the so-called minimax algorithm. This algorithm needs an evaluation function, which allows any game configuration to be associated with a value. For example, in our case, it can be simply +100 if the player in question won in this configuration, -100 if the opponent won, and otherwise +10 for each small grid won by the player in question and -10 for each one won by the opponent. But we also tested more complicated functions, as detailed below.

To choose what to play, this agent simulates all the "next" configurations from the current configuration up to a previously fixed depth, and assigns to each of the configurations of the last depth level a value according to the chosen evaluation function. For example, if the depth is 3, the agent will simulate all of its immediate choices, all of the opponent's choices in response to each of these choices, and all of its choices in response to each of the opponent's responses to each of its immediate choices.

Then, if the depth is odd, let us imagine that the agent rationally chooses, for each deepest choice of the adversary, the configuration of maximum depth that gives him the best score. Let us then imagine that the adversary, rationally, chooses then for each penultimate choice of the agent the child configuration that leads to the smallest maximum gain of the agent in the next round. And so on : at one degree of depth less, the agent chooses, for each penultimate choice of the adversary, the child configuration that maximises the minimal score that the adversary will rationally choose. And so we go back to depth zero, where the agent makes its final choice. And if the depth is even, the same applies, but the deepest step is then an opponent choice.

A drawback of this algorithm is that sometimes it performs terminal score calculations and tree ascents that are not necessary because it is known that the minimum or maximum is already reached in the previous calculations. Let us give an example [Fig. 2]. Let us say that the depth is 3, so the score calculations are done after an action of the minimax agent. Let us assume that, if the opponent had previously (in the move chaining simulation) performed action number 1, each of the actions then available to the agent was of score less than 3, with one of the configurations of score 3. In this case, if the first of the possible actions for the agent after the adversary has performed action number 2 has a score of 5, then the minimax agent will choose for this subtree a configuration with a score higher than or equal to 5, and thus the adversary, being rational, will have previously chosen preferentially his first choice (or another one, but in any case not the second one because it brings more points to the minimax agent). Therefore, it is useless to calculate the scores of the other configurations resulting from the opponent configuration number 2.

Pruning [4]

Alpha beta pruning lets us solve the issue above. Indeed, it consists in remembering the minimum score α and maximum score β encountered during the tree search. This lets us skip the exploration of sub-trees that would not modify either α nor β, thus speeding up the minimax algorithm without changing the resulting decision.

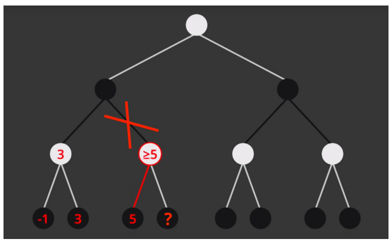


Fig. 2. Example of pruning in minimax taken from [3] where black maximizes the score and white minimizes it - whatever the value behind ?, black will choose a score ≥ 5 and thus white will choose the left branch (of value $3 < 5$)

A problem of the minimax algorithm is that we do need an evaluation function, however it is difficult to choose an evaluation function that adequately describes the lead that a player can have over his opponent. So, if possible, we therefore prefer to use an algorithm that does not need an evaluation function. The other way to evaluate a configuration is to run it to the end of the game. For this purpose, to avoid too much complexity, it is better to simulate the endgame randomly.

Pure Monte Carlo Game Search

Pure Monte Carlo Game Search is a heuristic search algorithm. We did not implement it, but explaining it is useful to understand Monte Carlo Tree Search.

At each step of the game, the algorithm simulates random game sequences from the current configuration for each possible initial move. For instance, if it has three possibilities for its next move, it will simulate -let's say- one hundred random games starting from each of the three configurations, and choose the configuration that has the best win rate on its one hundred random games.

A drawback of this algorithm is that in the one hundred simulations from a given configuration, we do not primarily study the win rates associated with the best sub-configurations of this configuration. Let us consider a configuration resulting in one great sub-configuration while the others are bad. It will have a lower win rate than a configuration that results in a set of poor sub-configurations. However the former configuration is clearly more advantageous for a smart player who chooses the great sub-configuration in the next round. That is why we prefer to use Monte Carlo Tree Search.

Monte Carlo Tree Search [5]

The difference with the previous algorithm is that while running the hundred simulations from the configuration, we prefer to choose initial moves that have already yielded appreciable win rates. We thus use a tree to remember states and their win rates. To choose these initial moves, we juggle between exploitation (choosing the set of initial moves that has for the moment the best victory rate) and exploration (testing a new move from a certain partial sequence of initial moves). This is the **selection** phase.

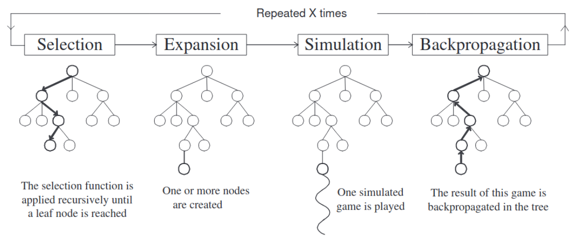

Repeated X times
Selection
Expansion
Simulation
Backpropagation
The selection function is applied recursively until a leaf node is reached
One or more nodes are created
One simulated game is played
The result of this game is backpropagated in the tree

Fig. 3. Outline of the Monte Carlo Tree Search algorithm, taken from [5]

Then, we **expand** that node by applying a random valid action to get a child node. From that node, we compute a random **simulation** of the game until its end. The result - win or loss - is **backpropagated** from the expanded node to the root to update each win rate.

In order to balance exploitation and exploration, we can use the UCT (Upper Confidence bound applied to trees) formula : to choose the node that has the highest value of $\frac{w_i}{n_i} + c.\sqrt{\frac{\log N_i}{n_i}}$ with w_i the number of wins for the node considered after the i-th move, n_i the number of simulations for the node considered after the i-th move, N_i the total number of simulations after the i-th move run by the parent node of the one considered, and c the empirically chosen exploration parameter.

Deep Q-learning [6]

Deep Q-learning consists in using a neural network to learn the value $Q(s, a)$ of applying action a at the state s of the game for all actions and states. Let us define: We thus use a neural network q_w with parameter w such that for a given state s with possible actions $a_1, ..., a_{n_s}$, $q_w(s) = [Q(s, a_i)]_{i=1}^{i=n_s}$. During the learning phase, the network plays against another agent (the minimax agent for example). Parameter w is updated using the following equation:

$$w = w + \alpha \left[target(w) - pred(w)\right] \nabla_w [q_w(S_t)]_{A_t}$$

where

$$target(w) = (R_t + \gamma \max_{a \in \mathcal{A}} [q_w(S_{t+1})]_a)$$

$$pred(w) = [q_w(S_t)]_{A_t}$$

with

- α the learning rate
- R_t, S_t and A_t the reward, state and action at step t
- $\gamma \in (0, 1)$ representing the importance of future rewards $\mathcal{A}$ the space of possible actions

Once the network is trained, the associated agent can simply play action $a = \text{argmax}_{a \in \mathcal{A}} Q(s, a)$ at a given state s.

III. The Environment

State and action spaces

The board consists in 9 squares made of 3 x 3 tic-tac-toe grids which are either empty or contain a X or O mark depending on the player. A state is defined by the contents of the 81 cells at a given moment. This means that theoretically, there are 3^{81} possible states. In practice, the space of valid states is smaller as many states are impossible to reach in game.

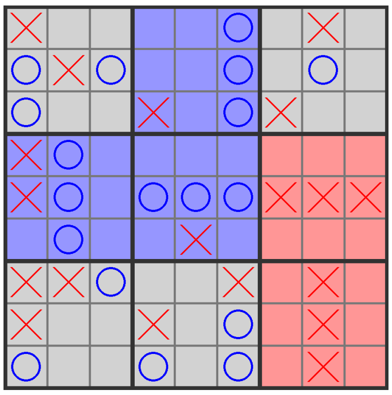


Fig. 4. Our implementation of ultimate tic-tac-toe

An action consists in putting a mark (X or O depending on the player) in an empty cell. Notice that the action space at a given moment in the game depends on the previous move: the small grid in which the player moves is determined by the position of the opponent's previous mark in its grid. Either the small grid in which the player is sent is neither full nor won, in which case they must play there, or it is, in which case the player can put a mark in any empty cell.

Reward function

Keep in mind that this is only an example of a (relatively complicated) reward function. We have tested other reward functions, including the simple one mentioned above.

The reward function is non-trivial as it is unclear for most states whether they will lead to a victory for the first or second player. This function should take into account the usefulness of winning a certain small grid. It should therefore favor winning grids that:

- can be involved in a lot of winning combinations. For instance, the middle grid can be in four winning lines (the horizontal and vertical lines as well as the two diagonals) and thus is more valuable than any other small grid.
- bring the player closer to a victory. For example, the top right grid is more valuable if the player has already won the top left and the top middle grids.

In the same way, at the local level, the function should value cells that lead more often to victory.

Let us denote $\mathcal{L}$ as the different lines in a tic-tac-toe grid: the three horizontal lines, the three vertical lines and the two diagonals. Let us consider a grid g - either the global one or a small one - that has not been won yet and a player $i \in 1,2$ (note that his opponent is $3-i$). For any line l, we define its value for player i as

$$v_{line}(l,i) = \begin{cases} 1 & \text{if } i \text{ has won 2 cells and 3-i none in } l \\ 0 & \text{otherwise} \end{cases}$$

We give any cell c a value $v_{cell}(c)$: 2 for the middle cell, 1.5 for the corners and 1 for the rest. We also denote as $c[g]$ the content of cell c: 0 if it is empty, or $i \in 1,2$ for the player who took it. Using the above functions, we define the value of the grid g for player i as the following:

$$v_{grid}(g,i) = 5\min\left[2, \sum_{l\in\mathcal{L}} v_{line}(l,i)\right] + \sum_{c\in g} v_{cell}(c)\mathbb{1}_{g[c]=i}$$

and its global value as $v_{grid}(g) = v_{grid}(g,1) - v_{grid}(g,2)$.

Using the previous evaluation function for the reward

It is now important to take into account both the situation of the player on the macro-scale and on the micro(scale for each grid in order to compute its reward. In fact a cross not winning any cell can be better than winning a cell if well placed. We use the following reward function at time t:

- if the agent wins the game, the reward is 400.
- if the agent wins a large cell, the reward is computed as such:
$$R_t = [v_{grid}(G_t) - v_{grid}(G_{t-1})]$$
- if the agent does not win a large cell, we multiply how much he advanced in winning that cell by how much winning this large cell would improve his situation on the large grid:

$$R_t = [v_{grid}(G_t) - v_{grid}(G_{t-1})]\,[v_{grid}(g_t) - v_{grid}(g_{t-1})]$$

where G_t is the large grid and g_t is the small grid in which the player just played at step t. The reward for winning is 400.

Main challenges

The main challenge posed by the environment is the size of the state space ($\approx 3^{81}$ for the 81 cells states and the big cells playable). It leads us to define a reward function that may not be optimal for the agent to win, as it is not obvious whether a state has a high probability of leading to a victory. As this is a 2-player game, another challenge is the ability to predict the opponent's moves.

IV. The Agent

Minimax algorithm

We have implemented the minimax agent as described above. As explained, it is difficult to find a satisfactory evaluation function (of an intermediate state). We have tested several of them, notably the simple evaluation of the number of small grids ahead of the opponent, but also a more elaborate function giving points to each partially filled small grid according to the presence of advantageous configurations (well aligned, in particular) according to weights fixed by hand in the measure of our good sense.

Note a trade-off here: for a fixed target execution time for the algorithm, the more complex the evaluation function, the less deep the minimax algorithm can be. Therefore, a more naive evaluation function may sometimes be more appropriate. For example, the complex reward function discussed above has twenty times the time complexity of the simple reward function above on our computers, so an algorithm with a fixed execution time needs to explore less deeply to compensate.

Finally, let us note an implementation trick: the minimax tree is supposed to keep in memory the intermediate configurations during the descent. This can take up considerable memory space. For this reason, we preferred to keep in memory only the individual moves that lead from the previous stage to the current stage. Thus, to have the state at a given stage, it is deduced from the moves executed from the current configuration, whose description is stored in memory.

As mentioned above, the difficulty is to find a satisfactory evaluation function. Hence the attempt to use an algorithm that does not need an evaluation function.

Monte Carlo Tree Search algorithm

Monte Carlo Tree Search (MCTS) agents are famous for their performance in 2-player games such as Go. We use a MCTS agent for ultimate tic-tac-toe as it does not require any reward function. This means it only needs to know what are the winning states and does not rely on a potentially poorly chosen reward function. Furthermore, it does not need a training phase.

The issue with MCTS is the tuning of two parameters:

- a constant c that lets us balance exploration versus exploitation
- a constant n_{iter} which defines the number of iterations of the tree search. Choosing a big value helps the agent perform better but is quite costly in terms of time.

We first take $n_{iter} = 500$ to get a good balance between speed and efficency. Using this, we pit MCTS against the 5-step Minimax Agent for several values of c and choose the best. We then use that fixed constant ($c = \sqrt{2}$) to test distinct values of n_{iter} (see results in section V). Our implementation of MCTS uses a custom tree structure, with each node storing information like how many times it was visited and how many times it led to a win.

TABLE I
WIN RATE (OUT OF 100 GAMES) OF MCTS FOR SEVERAL VALUES OF THE EXPLORATION PARAMATER C

Parameter c	**0**	**0.5**	**$\sqrt{2}$**	**3**	**5**
Win rate of MCTS (%)	8	17	**23**	15	11

Deep Q-learning

Deep Q-learning is a reinforcement learning method allowing to model big state spaces, which is the case of our study: the game has 3^{81} possible states. Its idea is to estimate the Q table through a neural network, which will take the state of the game as input, and will produce as outputs the estimated values of each action.

In our case, a big problem came from the fact that not all actions are possible. Among all the 81 possible actions - mark a specific tile - only a handful are usually allowed by the rules of the game. However, since the network's architecture is fixed - especially the number of outputs - the agent can choose illegal actions. To solve this issue, we implemented two solutions:

- During the learning process, we check that an action is possible and if the agent chooses an illegal action a highly negative reward is given to the agent.
- During the play phase, we use on top of the prediction a filter removing all the actions not legal. This way the agent chooses the most likely legal action.
- Finally, to help the learning process, we apply randomly the filter to the prediction (with probability ϵ). This is used to make the agent play so that it does not get stuck trying to play an illegal action. When the agent chooses an illegal action we increase ϵ, and when it play several legal moves, we decrease ϵ.

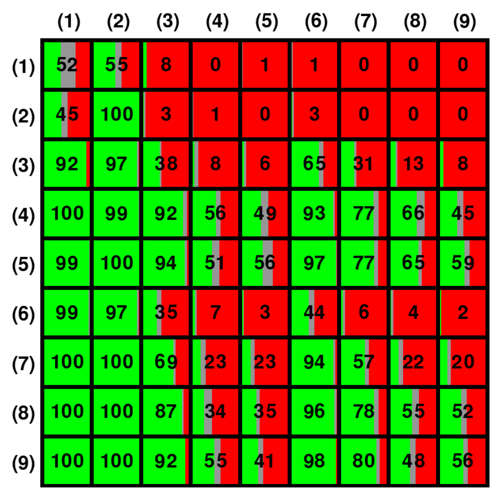
Deep Q Learning agents need to be trained before evaluating their performances; that can be seen as either an advantage - once the learning process is finished, we can save our agent's network to load it later in order to perform meaningful actions in the game with a very small computing time - or an inconvenient - the learning process can be extremely costly in time, which means that our agent will only be as smart as the amount of time we have at our disposal.

To train it, we made it play against opponents. First against the random agent to make it learn legal moves. We chose the random agent as it is the fastest one. Then against the Minimax agent with 5 steps ahead as it is quite a good agent with a relatively fast computation time.

The architecture of our agent's neural network was designed with simplicity in mind : two hidden layers fully connected to each other and to the inputs/outputs. More complexes networks were tried, but they did not perform any better than this one and had the major inconvenient of greatly slowing the learning process. If we had more time, we would have liked to design and evaluate convolutional neural networks.

V. Results and Discussion

In order to evaluate the performance of each agent, let's confront all of them. For two agent opposed, we assign a score $s(i,j)$ for the agent i against the agent$j$, computed as follows: $s(i,j) = 100{*}(w(i,j){+}0.5{*}e(i,j))/n$ with $w(i,j)$ and $e(i,j)$ the number of win and draws of i over j.



	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
(1)	52	55	8	0	1	1	0	0	0
(2)	45	100	3	1	0	3	0	0	0
(3)	92	97	38	8	6	65	31	13	8
(4)	100	99	92	56	49	93	77	66	45
(5)	99	100	94	51	56	97	77	65	59
(6)	99	97	35	7	3	44	6	4	2
(7)	100	100	69	23	23	94	57	22	20
(8)	100	100	87	34	35	96	78	55	52
(9)	100	100	92	55	41	98	80	48	56

Fig. 5. Score for each agent against each agent for 100 games. The results should be read line by line, displaying in green the proportion of wins, grey of draws and red losses.
(1) Random Agent
(2) DQN Agent (trained for 15k games)
(3) 1-step Minimax Agent (random)
(4) 3-step Minimax Agent (random)
(5) 5-step Minimax Agent (random)
(6) MCTS Agent ($N = 100$)
(7) MCTS Agent ($N = 500$)
(8) MCTS Agent ($N = 1000$)
(9) MCTS Agent ($N = 2000$)

From the simulations it seems that the 5-step Minimax Agent is the best agent (using complex rewards with randomization of the best moves). Note that we chose the complicated reward function because its minimax algorithm turned out to win largely against the simple reward minimax for an equivalent execution time (thus with a higher depth for the simple reward minimax). Let's highlight a few facts shown in these results:

The DQN agent was not trained for long enough (even though it took more than 15 hours). For comparison, the famous AlphaZero algorithm took about 20 million games before it finally beat humans. Surprisingly it is even worse than the random agent but this does not mean that it has not learnt. One issue was the fact that for a given state most actions are considered illegal (as they are not in the right board), however after the training the agent chose only 21% of the time an illegal action (which is not bad as less than 9 actions over 81 are usually legal for a given state). Moreover, even though it only had a score of 45 against the random agent, the DQN agent playing first won every single game against another DQN Agent, demonstrating that a pattern has been learnt.

For the other agents, beginning does not seem to make a big difference as most scores on the diagonal are close to 50. *(Of course since only 100 games were played for each duels, these numbers have high intervals of uncertainty).* In this graph, only Minimax Agents with randomization were tested but doing the same thing without the randomization leads to significantly different results. For the same number of states, the agent starting will nearly always win. And for two Minimax agents with different number of steps, the agent with highest number of steps wins in more than 80% of the games.

In the end, MCTS seems to lead to promising results but the agent is much slower than a Minimax agent with similar performances (The MCTS Agent with $N = 2000$ is more than 4.5 times slower than the 5-step Minimax Agent).

Using the previous evaluation function for the reward

Choosing an appropriate reward function was a real challenge. For the minimax, with the simple reward only giving point for a victory, the agent plays randomly at the beginning since it cannot win with certainty in a few steps. However, with the complex reward, the computation is much longer, so at the end it can miss opportunities that requires a few more steps to plan the victory. This is why we implement the **improved minimax**. At the beginning, the agent uses the complex reward function in order to play in good spots. After a certain number of plays (N_{start}), the agent first uses the simple reward with a few more steps that what he do with the complex reward (as it is faster to compute), if a positive reward is found, that means that a given path leads to victory so it plays that move, otherwise it uses the complex reward.

VI. Conclusion

Producing a good agent for ultimate tic-tac-toe is actually tricky, despite the original tic-tac-toe game having an easy winning strategy. The minimax and MCTS agents need a trade-off between decision time and performance. Meanwhile, the DQN seems to require a long training phase - 15 000 games are not enough for the agent to beat the random agent. By playing the game ourselves, we were able to beat almost all of the agents quite easily. But the minimax algorithm of depth 6 (maximum for our computers) is relatively intelligent. Moreover, using the improved minimax provides a really good agent. Over 20 games, only 2 people were able to beat it while it only took a few seconds to compute each move (so it could be even better with more time for each move). Try to beat it! In the future, we could try training a DQN using games between real players for it to get a better sense of the strategies involved in the game. We could also think of other reward functions that might help agents like the minimax produce better results.

References

[1] E. Zermelo. Über eine anwendung der mengenlehre auf die theorie des schachspiels. https://web.archive.org/web/20131217224959if_/http://www.socio.ethz.ch/publications/spieltheorie/klassiker/Zermelo_Uber_eine_Anwendung_der_Mengenlehre_auf_die_Theorie_des_Schachspiels.pdf, 1913.

[2] G. Bertholon et al. At Most 43 Moves, At Least 29 - Optimal Strategies and Bounds for Ultimate Tic-Tac-Toe

[3] Minimax principle. *Encyclopedia of mathematics*, 1994.

[4] S. Lague. Algorithms Explained – minimax and alpha-beta pruning. https://www.youtube.com/watch?v=l-hh51ncgDI

[5] G. Chaslot et al. Progressive strategies for Monte-Carlo tree search. https://dke.maastrichtuniversity.nl/m.winands/documents/pMCTS.pdf, 2008.

[6] In Lecture VII - Reinforcement learning III. *INF581 Advanced Machine Learning and Autonomous Agents*, 2022.