# Contents

[Contents 1](#_Toc48600214)

[Introduction 4](#_Toc48600215)

[0. Representing boardgames 5](#_Toc48600216)

[1.1. The state 5](#_Toc48600217)

[1.2. Actions 6](#_Toc48600218)

[1.3. Consequences 6](#_Toc48600219)

[1.4. Symmetrical games 7](#_Toc48600220)

[2. Boardgame AI 9](#_Toc48600221)

[2.1. MiniMax 9](#_Toc48600222)

[2.1.1. Algorithm 9](#_Toc48600223)

[2.2. NegaMax 10](#_Toc48600224)

[2.3. AlphaBeta 11](#_Toc48600225)

[2.3.1. Algorithm 12](#_Toc48600226)

[2.3.2. MiniMax 12](#_Toc48600227)

[2.3.3. NegaMax 13](#_Toc48600228)

[2.3.4. Killer heuristic 14](#_Toc48600229)

[2.3.5. Refutation tables 14](#_Toc48600230)

[2.3.6. Minimal window 14](#_Toc48600231)

[2.3.7. Aspiration search 14](#_Toc48600232)

[2.3.8. Iterative deepening 14](#_Toc48600233)

[2.3.9. Transposition Tables 15](#_Toc48600234)

[2.4. SSS\* 16](#_Toc48600235)

[2.4.1. Algorithm 17](#_Toc48600236)

[2.5. PVS, NegaScout, MTDf(n) 17](#_Toc48600237)

[2.6. Best Node Search (BNS) 20](#_Toc48600238)

[Monte-Carlo tree search ? 20](#_Toc48600239)

[3. Unity ML Agents 21](#_Toc48600240)

[3.1. Agent 22](#_Toc48600241)

[3.1.1. Behavior parameters 22](#_Toc48600242)

[3.2. Training 23](#_Toc48600243)

[3.2.1. PPO vs SAC 23](#_Toc48600244)

[3.2.2. Training parameters 24](#_Toc48600245)

[3.2.3. Self-play 26](#_Toc48600246)

[3.3. Results 26](#_Toc48600247)

[3.3.1. Tensor Board 27](#_Toc48600248)

[4. The demo 28](#_Toc48600249)

[4.1. Framework 28](#_Toc48600250)

[4.2. Algorithmic agents 28](#_Toc48600251)

[4.3. Machine learning agents 28](#_Toc48600252)

[4.3.1. Training 28](#_Toc48600253)

[5. Conclusion 34](#_Toc48600254)

[6. References 35](#_Toc48600255)

# Introduction

# Representing boardgames

Boardgames have a state, actions, and consequences. In these boardgames the entire state is defined by the board. For the games considered in the paper the only possible action is to pick a square on the board and change its state. The consequences are a number of rules that change the state depending on the actions chosen by the players, for example the win conditions.

## The state

The games have a board state which gets manipulated by the players until the game ends. The board consists of a matrix of cell which can have three possible states: empty, claimed by player one and claimed by player two.

The board will be represented by a “bit board”. The board state is saved in the bits of two ulong variables (64 bits, one for each player. A 0 bit means it is not claimed by the player and a 1 bit means it is claimed by the player. The 64-bit format limits the size of the matrix to 8x8.

Bit boards have some major advantages. Operations are efficient since they can check and manipulate multiple cells of the board at once and bit operations (see chapters Checking win conditions and Symmetric games). Memory wise bit boards take the minimum amount of space (Thill, 2008).

/// <summary>

/// Bits of the mask for a certain player

/// </summary>

public ulong bits;

Two bitmasks, one for the column and one for the row, are used to find link the bits to the cells:

private static readonly ulong[] columnMasks = new ulong[8]

{

0b\_00000001\_00000001\_00000001\_00000001\_00000001\_00000001\_00000001\_00000001,

0b\_00000010\_00000010\_00000010\_00000010\_00000010\_00000010\_00000010\_00000010,

0b\_00000100\_00000100\_00000100\_00000100\_00000100\_00000100\_00000100\_00000100,

0b\_00001000\_00001000\_00001000\_00001000\_00001000\_00001000\_00001000\_00001000,

0b\_00010000\_00010000\_00010000\_00010000\_00010000\_00010000\_00010000\_00010000,

0b\_00100000\_00100000\_00100000\_00100000\_00100000\_00100000\_00100000\_00100000,

0b\_01000000\_01000000\_01000000\_01000000\_01000000\_01000000\_01000000\_01000000,

0b\_10000000\_10000000\_10000000\_10000000\_10000000\_10000000\_10000000\_10000000

};

private static readonly ulong[] rowMasks = new ulong[8]

{

0b\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000\_11111111,

0b\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000\_11111111\_00000000,

0b\_00000000\_00000000\_00000000\_00000000\_00000000\_11111111\_00000000\_00000000,

0b\_00000000\_00000000\_00000000\_00000000\_11111111\_00000000\_00000000\_00000000,

0b\_00000000\_00000000\_00000000\_11111111\_00000000\_00000000\_00000000\_00000000,

0b\_00000000\_00000000\_11111111\_00000000\_00000000\_00000000\_00000000\_00000000,

0b\_00000000\_11111111\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000,

0b\_11111111\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000

};

To check whether a certain player has claimed the cell at column x, row y:

public bool GetBit(int x, int y)

=> bits & columnMasks[x] & rowMasks[y] != 0;

As a little side note, for checkers the maximum board size is actually 9x9 since you can turn the board 45°, making the moves straight rather than diagonal, and remove the white squares ending up with a board as displayed below, where the black player can play downwards and to the right and the white player can play upwards and to the left. The 9x9 format is not popular at all though.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | ● | ● |  |  |  |
|  |  | ● | ● | ● |  |  |  |
|  | ● | ● | ● |  |  |  |  |
| ● | ● | ● |  |  |  | ○ | ○ |
| ● | ● |  |  |  | ○ | ○ | ○ |
|  |  |  |  | ○ | ○ | ○ |  |
|  |  |  | ○ | ○ | ○ |  |  |
|  |  |  | ○ | ○ |  |  |  |

## Actions

As said before the actions for the games in the scope of paper are limited to changing the state of a single square, this can be done using the following function:

public void SetBit(int x, int y, bool bit)

{

if (bit)

bits |= columnMasks[position.x] & rowMasks[position.y];

else

bits &= ~(columnMasks[position.x] & rowMasks[position.y]);

}

## Consequences

The consequences can be many things, for example capturing a group in a game of Go or capturing a piece in a game of checkers, but here we will take the simplest and most necessary example of the win-condition.

Since the board is made from a bit matrix, we can use bit masks to scan over the board state to check if a win condition is met. We will use Tic-tac-toe or any three in a row game as an example. These games are won once three cells are connected in a line by one player, this can be done in four different ways, either horizontally, vertically, diagonally for going up or diagonally going down. These conditions can be represented by the following four bitmasks:

private readonly BitMask[] masks = new BitMask[4]

{

new BitMask(0b\_00000111, new Size(3, 1)), // -

new BitMask(0b\_00000001\_00000001\_00000001, new Size(1, 3)), // |

new BitMask(0b\_00000001\_00000010\_00000100, new Size(3, 3)), // /

new BitMask(0b\_00000100\_00000010\_00000001, new Size(3, 3)) // \

};

Now we can scan over the board looking for a three in a row for a certain player using:

public override bool GetIsWin()

{

for (int i = 0; i < masks.Length; ++i)

for (int x = 0 x <= board.size.x - masks[i].size.x; ++x)

for (int y = 0; <= board.size.y - masks[i].size.y; ++y)

{

ulong mask = masks[i].bits × columnMasks[x] & rowMasks[y];

if (bits & mask == mask)

return true;

}

return false;

}

This method is easy to mimic for other games since the only thing you have to do is update the list of masks. The GetIsWin() function works game independent. Of course, this is limited to games that have win conditions based on a certain local board state.

For example, checkers ends when a player is out of pieces, this can be tested by:

public override bool GetIsWin()

=> opponent.bits == 0;

For a game of Go it becomes extremely hard to define a win condition based on the board position, since the game end when the two players agree on ending the game and then the players need to agree on how to dived the territories and what the actual score is. If the players do not agree the game just continues.

## Symmetrical games

A machine learning agent has no concept of symmetry and learns certain concepts multiple times which humans quickly would recognize as symmetrical. For example, the following Tic-tac-toe situations are completely equivalent to each other:

|  |  |  |
| --- | --- | --- |
|  | ○ |  |
| ○ |  |  |
|  | ● | ● |

|  |  |  |
| --- | --- | --- |
|  | ● | ● |
| ○ |  |  |
|  | ○ |  |

|  |  |  |
| --- | --- | --- |
|  | ○ |  |
| ● |  | ○ |
| ● |  |  |

|  |  |  |
| --- | --- | --- |
| ● |  |  |
| ● |  | ○ |
|  | ○ |  |
|  | ○ |  |
|  |  | ○ |
| ● | ● |  |

|  |  |  |
| --- | --- | --- |
| ● | ● |  |
|  |  | ○ |
|  | ○ |  |

|  |  |  |
| --- | --- | --- |
|  | ○ |  |
| ○ |  | ● |
|  |  | ● |

|  |  |  |
| --- | --- | --- |
|  |  | ● |
| ○ |  | ● |
|  | ○ |  |

We can help the agent by manipulating the matrix so that all equivalent problems are given as one. For this we use the following functions below, using more bit operations. These functions are independent of the board size and thus can also be used for other games than Tic-tac-toe.

public void MirrorHorizontally() // |

{

for (int x = 0; x < size.x / 2; ++x)

{

ulong column0 = bits & columnMasks[x];

ulong column1 = bits & columnMasks[size.x - x - 1];

bits &= ~(column0 | column1);

column0 <<= (size.x - 2 \* x - 1);

column1 >>= (size.x - 2 \* x - 1);

bits |= column0 | column1;

}

}

public void MirrorVertically() // -

{

for (int y = 0; y < size.y / 2; ++y)

{

ulong row0 = bits & rowMasks[y];

ulong row1 = bits & rowMasks[size.y - y - 1];

bits &= ~(row0 | row1);

row0 <<= 8 \* (size.y - 2 \* y - 1);

row1 >>= 8 \* (size.y - 2 \* y - 1);

bits |= row0 | row1;

}

}

public void MirrorDiagonally() // /

{

for (int x = 1; x < (size.x < size.y ? size.x : size.y); ++x)

for (int y = 0; y < x; ++y)

{

bool xy = GetBit(x, y);

bool yx = GetBit(y, x);

SetBit(x, y, yx);

SetBit(y, x, xy);

}

}

# Boardgame AI

Now that we have that game setup, we need an adversary for single player games. The MiniMax-algorithm lends itself for this kind of games. Many researches have suggested improvements on this algorithm. In this chapter we go in depth in the most well know variations on MiniMax. The algorithmic function described below all have a working version in the Unity-project. In this paper the same functionality is given in pseudo code for easier understanding.

All these algorithms need either a lot of time or good heuristic function (function that estimates a score for a certain board state) on games with a decent number of moves (both in depth as in possibilities). Writing a heuristic function needs a good understanding of the game at hand. To avoid this problem a machine learning solution might be a better option, this way the computer develops an understanding for the game and the developer does not need to think about a good heuristic function. In the next chapter we look at a machine learning solution using Unity ML Agents

## MiniMax

MiniMax is a simple algorithm that searches through the possible moves, picking the move with which leads to the best board position (MAX option in the image below) on the turn of the active player and picking the move with the worst board position (MIN option in the image below) for the active player when it is the opponents turn. The algorithm can also keep track of the number of moves played, deciding to return a heuristic value after a certain amount of moves instead of keep searching for an end state (which can take a lot of time).

|  |  |
| --- | --- |
| Active player  Opponent  Active player  Opponent |  |

MiniMax tree (Heineman et al., 2016, pp. 176)

### Algorithm

The MiniMax-algorithm takes in a state (node), the amount of recursion steps it is allowed to do (depth) and whether it needs to maximize or minimize (maximizingPlayer). It first checks whether is maximum depth is reached or if and end state is reached and if so, it returns the value for the state. Then it checks whether the maximizing player is making the next move, if that is the case it will call the minimax function again for each possible new state and pick the highest and return the highest scoring state, else it will pick the lowest scoring state. Following is the MiniMax-algorithm in pseudocode, the original call would be (Heineman et al., 2016, pp. 174–180):

minimax(origin, depth, TRUE)

 function minimax(node, depth, maximizingPlayer) is

if depth = 0 or node is a terminal node then

return the heuristic value for the maximizing player of node

if maximizingPlayer then

value := −∞

for each child of node do

value := max(value, minimax(child, depth − 1, FALSE))

return value

else (\* minimizing player \*)

value := +∞

for each child of node do

value := min(value, minimax(child, depth − 1, TRUE))

return value

## NegaMax

NegaMax is a simplified version of MiniMax, using some clever tricks.

First, we replace the maximizingPlayer variable with a variable f (float) that either holds -1 (minimizing player) or +1 (maximizing player), initialized with the value +1. We can now use f to remove the if statement, knowing that .

 function score (node, depth, f) is

if depth = 0 or node is a terminal node then

return the heuristic value of node for the maximizing player

value := f × -∞

for each child of node do

value := f × max(f × value, f × score(child, depth − 1, -f))

return value

The examined games are zero-sum, meaning that the score of both players always add up to zero. Either one player wins (+1) and one player loses (-1) adding up to zero, or the players draw (+0 for both players) adding up to zero. This characteristic can be represented as . Using this characteristic we can rewrite as .

 function score (node, depth, f) is

if depth = 0 or node is a terminal node then

return the heuristic value of node for the maximizing player

value := f × -∞

for each child of node do

value := f × max(f × value, -score(child, depth − 1, +1))

return value

Since the score function is now always called with +1 for the f parameter (both in the initial call as in the recursive calls) we can just remove the f parameter.

 function score (node, depth) is

if depth = 0 or node is a terminal node then

return the heuristic value of node for the maximizing player

value := -∞

for each child of node do

value := max(value, -score(child, depth − 1))

return value

We can add a color parameter that holds the color of the maximizing player (represented by -1 or +1) if we also make the heuristic value dependent on who is making the last move. After this step we have the NegaMax algorithm as it is best known (Heineman et al., 2016, pp. 180-183):

(\* Initial call for Player A's root node \*)

negamax(rootNode, depth, +1)

(\* Initial call for Player B's root node \*)

negamax(rootNode, depth, −1)

function negamax(node, depth, color) is

if depth = 0 or node is a terminal node then

return the heuristic value of node for the “color” player

value := −∞

for each child of node do

value := max(value, −negamax(child, depth − 1, −color))

return value

## AlphaBeta

+ <https://books.google.be/books?id=wyHSBQAAQBAJ&pg=SA38-PA13&redir_esc=y#v=onepage&q&f=fals> (Gonzalez et al., 2014, pp. 38-10, 38-12)

TEXT

(Heineman et al., 2016, pp. 183-189)

Minimax evaluates a player’s best move when considering the opponent’s counter‐ moves, but this information is not used while the game tree is generated! Consider the BoardEvaluation scoring function introduced earlier. Recall Figure 7-5, which shows the partial expansion of the game tree from an initial game state after X has made two moves and O has made just one move. Note how Minimax plods along even though each subsequent search reveals a los‐ ing board if X is able to complete the diagonal. A total of 36 nodes are evaluated. Minimax takes no advantage of the fact that the original decision for O to play in the upper-left corner prevented X from scoring an immediate victory. AlphaBeta defines a consistent strategy to prune unproductive searches from the search tree. After evaluating the sub game tree rooted at (1) in Figure 7-8, AlphaBeta knows that if this move is made the opponent cannot force a worse position than -3, which means the best the player can do is score a 3. When AlphaBeta gets to the game state (2), the first child game state (3) evaluates to 2. This means that if the move for (2) is selected, the opponent can force the player into a game state that is less than the best move found so far (i.e., 3). There is no need to check the sibling subtree rooted at (4), so it is pruned away. Using AlphaBeta, the equivalent expansion of the game tree is shown in Figure 7-9. As AlphaBeta searches for the best move in Figure 7-9, it remembers that X can score no higher than 2 if O plays in the upper-left corner. For each subsequent other move for O, AlphaBeta determines that X has at least one countermove that outper‐ forms the first move for O (indeed, in all cases X can win). Thus, the game tree expands only 16 nodes, a savings of more than 50% from Minimax. AlphaBeta selects the same move that Minimax would have selected, with potentially signifi‐ cant performance savings.

|  |  |
| --- | --- |
| Active player  Opponent  Active player  Opponent |  |

AlphaBeta tree (Heineman et al., 2016, pp. 184)

### Algorithm

The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta < alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

To illustrate this with a real-life example, suppose somebody is playing chess, and it is their turn. Move "A" will improve the player's position. The player continues to look for moves to make sure a better one has not been missed. Move "B" is also a good move, but the player then realizes that it will allow the opponent to force checkmate in two moves. Thus, other outcomes from playing move B no longer need to be considered since the opponent can force a win. The maximum score that the opponent could force after move "B" is negative infinity: a loss for the player. This is less than the minimum position that was previously found; move "A" does not result in a forced loss in two moves.

### MiniMax

Pseudo code with alphabeta(origin, depth, −∞, +∞, TRUE) as initial call:

function alphabeta(node, depth, α, β, maximizingPlayer) is

if depth = 0 or node is a terminal node then

return the heuristic value of node

if maximizingPlayer then

value := −∞

for each child of node do

value := max(value, alphabeta(child, depth − 1, α, β, FALSE))

α := max(α, value)

if α ≥ β then

break (\* β cut-off \*)

return value

else

value := +∞

for each child of node do

value := min(value, alphabeta(child, depth − 1, α, β, TRUE))

β := min(β, value)

if β ≤ α then

break (\* α cut-off \*)

return value

(Alpha-beta pruning, n.d.)

### NegaMax

Algorithm optimizations for [minimax](https://en.wikipedia.org/wiki/Minimax) are also equally applicable for Negamax. [Alpha-beta pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning) can decrease the number of nodes the negamax algorithm evaluates in a search tree in a manner similar with its use with the minimax algorithm.

The pseudocode for depth-limited negamax search with alpha-beta pruning follows: <https://en.wikipedia.org/wiki/Negamax>

(\* Initial call for Player A's root node \*)

negamax(rootNode, depth, −∞, +∞, 1)

function negamax(node, depth, α, β, color) is

if depth = 0 or node is a terminal node then

return color × the heuristic value of node

childNodes := generateMoves(node)

childNodes := orderMoves(childNodes)

value := −∞

foreach child in childNodes do

value := max(value, −negamax(child, depth − 1, −β, −α, −color))

α := max(α, value)

if α ≥ β then

break (\* cut-off \*)

return value

Alpha (α) and beta (β) represent lower and upper bounds for child node values at a given tree depth. Negamax sets the arguments α and β for the root node to the lowest and highest values possible. Other search algorithms, such as [negascout](https://en.wikipedia.org/wiki/Negascout" \o "Negascout) and [MTD-f](https://en.wikipedia.org/wiki/MTD-f), may initialize α and β with alternate values to further improve tree search performance.

When negamax encounters a child node value outside an alpha/beta range, the negamax search cuts off thereby pruning portions of the game tree from exploration. Cut offs are implicit based on the node return value. A node value found within the range of its initial α and β is the node's exact (or true) value. This value is identical to the result the negamax base algorithm would return, without cut offs and without any α and β bounds. If a node return value is out of range, then the value represents an upper (if value ≤ α) or lower (if value ≥ β) bound for the node's exact value. Alpha-beta pruning eventually discards any value bound results. Such values do not contribute nor affect the negamax value at its root node.

This pseudocode shows the fail-soft variation of alpha-beta pruning. Fail-soft never returns α or β directly as a node value. Thus, a node value may be outside the initial α and β range bounds set with a negamax function call. In contrast, fail-hard alpha-beta pruning always limits a node value in the range of α and β.

This implementation also shows optional move ordering prior to the [foreach loop](https://en.wikipedia.org/wiki/Foreach_loop) that evaluates child nodes. Move ordering[[2]](https://en.wikipedia.org/wiki/Negamax#cite_note-Schaeffer-2) is an optimization for alpha beta pruning that attempts to guess the most probable child nodes that yield the node's score. The algorithm searches those child nodes first. The result of good guesses is earlier and more frequent alpha/beta cut offs occur, thereby pruning additional game tree branches and remaining child nodes from the search tree. <https://en.wikipedia.org/wiki/Negamax>

### Killer heuristic

In competitive two-player games, the **killer heuristic** is a technique for improving the efficiency of [alpha-beta pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning), which in turn improves the efficiency of the [minimax algorithm](https://en.wikipedia.org/wiki/Minimax_algorithm). This algorithm has an [exponential](https://en.wikipedia.org/wiki/Exponential_time) search time to find the optimal next move, so general methods for speeding it up are very useful. [Barbara Liskov](https://en.wikipedia.org/wiki/Barbara_Liskov) created the general [heuristic](https://en.wikipedia.org/wiki/Heuristic) to speed [tree search](https://en.wikipedia.org/wiki/Tree_search) in a computer program to play [chess endgames](https://en.wikipedia.org/wiki/Chess_endgame) for her Ph.D. thesis.[[1]](https://en.wikipedia.org/wiki/Killer_heuristic#cite_note-1)

[Alpha-beta pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning) works best when the best moves are considered first. This is because the best moves are the ones most likely to produce a *cutoff*, a condition where the game playing program knows that the position it is considering could not possibly have resulted from best play by both sides and so need not be considered further. I.e. the game playing program will always make its best available move for each position. It only needs to consider the other player's possible responses to that best move, and can skip evaluation of responses to (worse) moves it will not make.

The killer heuristic attempts to produce a cutoff by assuming that a move that produced a cutoff in another branch of the [game tree](https://en.wikipedia.org/wiki/Game_tree) at the same depth is likely to produce a cutoff in the present position, that is to say that a move that was a very good move from a different (but possibly similar) position might also be a good move in the present position. By trying the *killer move* before other moves, a game playing program can often produce an early cutoff, saving itself the effort of considering or even generating all legal moves from a position.

In practical implementation, game playing programs frequently keep track of two killer moves for each depth of the game tree (greater than depth of 1) and see if either of these moves, if legal, produces a cutoff before the program generates and considers the rest of the possible moves. If a non-killer move produces a cutoff, it replaces one of the two killer moves at its depth. This idea can be generalized into a set of [refutation tables](https://en.wikipedia.org/wiki/Refutation_table).

A generalization of the killer heuristic is the *history heuristic*. The history heuristic can be implemented as a table that is indexed by some characteristic of the move, for example "from" and "to" squares or piece moving and the "to" square. When there is a cutoff, the appropriate entry in the table is incremented, such as by adding *d²* or *2d* where *d* is the current search depth. This information is used when ordering moves.

Killer Heuristic <https://en.wikipedia.org/wiki/Killer_heuristic>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9124&rep=rep1&type=pdf>

### Refutation tables

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9124&rep=rep1&type=pdf>

### Minimal window

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9124&rep=rep1&type=pdf>

### Aspiration search

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9124&rep=rep1&type=pdf>

### Iterative deepening

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9124&rep=rep1&type=pdf>

Move ordering <https://en.wikipedia.org/wiki/Negamax>

It is possible to make some further improvements on the Alpha-Beta algorithm, even without losing accuracy. A first method is to increase the number of skipped branches. This can be done to check those branches that might lead to skipping other branches first (high potential moves). This can also be done by checking those moves first that scored the highest when testing for the previous move (Alpha-beta pruning, n.d.). Another trick is the killer heuristic, here the last move that lead to a skipped branch at the same level will be checked first since it has a high chance of leading to more branch skipping (Killer heuristic, n.d.).

Killer + move ordering: [https://www.adrian.idv.hk/2019-03-15-Tic-tac-toe/](https://www.adrian.idv.hk/2019-03-15-tictactoe/) (Tam, 2019)

Alpha–beta search can be made even faster by considering only a narrow search window (generally determined by guesswork based on experience). This is known as *aspiration search*. In the extreme case, the search is performed with alpha and beta equal; a technique known as [*zero-window search*](https://en.wikipedia.org/wiki/MTD-f#Zero-Window_Searches), *null-window search*, or *scout search*. This is particularly useful for win/loss searches near the end of a game where the extra depth gained from the narrow window and a simple win/loss evaluation function may lead to a conclusive result. If an aspiration search fails, it is straightforward to detect whether it failed *high* (high edge of window was too low) or *low* (lower edge of window was too high). This gives information about what window values might be useful in a re-search of the position.

Over time, other improvements have been suggested, and indeed the Falphabeta (fail-soft alpha-beta) idea of John Fishburn is nearly universal and is already incorporated above in a slightly modified form. Fishburn also suggested a combination of the killer heuristic and zero-window search under the name Lalphabeta ("last move with minimal window alpha-beta search").

### Transposition Tables

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9124&rep=rep1&type=pdf>

[Transposition tables](https://en.wikipedia.org/wiki/Transposition_table) selectively [memoize](https://en.wikipedia.org/wiki/Memoization" \o "Memoization) the values of nodes in the game tree. *Transposition* is a term reference that a given game board position can be reached in more than one way with differing game move sequences.

When negamax searches the game tree, and encounters the same node multiple times, a transposition table can return a previously computed value of the node, skipping potentially lengthy and duplicate re-computation of the node's value. Negamax performance improves particularly for game trees with many paths that lead to a given node in common.

The pseudo code that adds transposition table functions to negamax with alpha/beta pruning is given as follows: <https://en.wikipedia.org/wiki/Negamax>

(\* Initial call for Player A's root node \*)

negamax(rootNode, depth, −∞, +∞, 1)

function negamax(node, depth, α, β, color) is

alphaOrig := α

(\* Transposition Table Lookup; node is the lookup key for ttEntry \*)

ttEntry := transpositionTableLookup(node)

if ttEntry is valid and ttEntry.depth ≥ depth then

if ttEntry.flag = EXACT then

return ttEntry.value

else if ttEntry.flag = LOWERBOUND then

α := max(α, ttEntry.value)

else if ttEntry.flag = UPPERBOUND then

β := min(β, ttEntry.value)

if α ≥ β then

return ttEntry.value

if depth = 0 or node is a terminal node then

return color × the heuristic value of node

childNodes := generateMoves(node)

childNodes := orderMoves(childNodes)

value := −∞

for each child in childNodes do

value := max(value, −negamax(child, depth − 1, −β, −α, −color))

α := max(α, value)

if α ≥ β then

break

(\* Transposition Table Store; node is the lookup key for ttEntry \*)

ttEntry.value := value

if value ≤ alphaOrig then

ttEntry.flag := UPPERBOUND

else if value ≥ β then

ttEntry.flag := LOWERBOUND

else

ttEntry.flag := EXACT

ttEntry.depth := depth

transpositionTableStore(node, ttEntry)

return value

Alpha/beta pruning and maximum search depth constraints in negamax can result in partial, inexact, and entirely skipped evaluation of nodes in a game tree. This complicates adding transposition table optimizations for negamax. It is insufficient to track only the node's *value* in the table, because *value* may not be the node's true value. The code therefore must preserve and restore the relationship of *value* with alpha/beta parameters and the search depth for each transposition table entry.

Transposition tables are typically lossy and will omit or overwrite previous values of certain game tree nodes in its tables. This is necessary since the number of nodes negamax visits often far exceeds the transposition table size. Lost or omitted table entries are non-critical and will not affect the negamax result. However, lost entries may require negamax to re-compute certain game tree node values more frequently, thus affecting performance. <https://en.wikipedia.org/wiki/Negamax>

## SSS\*

**SSS\*** is a [search algorithm](https://en.wikipedia.org/wiki/Search_algorithm), introduced by George Stockman in 1979, that conducts a [state space search](https://en.wikipedia.org/wiki/State_space_search) traversing a [game tree](https://en.wikipedia.org/wiki/Game_tree) in a [best-first](https://en.wikipedia.org/wiki/Best-first_search) fashion similar to that of the [A\* search algorithm](https://en.wikipedia.org/wiki/A*_search_algorithm).

SSS\* is based on the notion of [solution trees](https://en.wikipedia.org/w/index.php?title=Solution_tree&action=edit&redlink=1). Informally, a solution tree can be formed from any arbitrary game tree by pruning the number of branches at each [MAX](https://en.wikipedia.org/wiki/Minimax) node to one. Such a tree represents a complete strategy for MAX, since it specifies exactly one MAX action for every possible sequence of moves made by the opponent. Given a game tree, SSS\* searches through the space of partial solution trees, gradually analyzing larger and larger subtrees, eventually producing a single solution tree with the same root and Minimax value as the original game tree. SSS\* never examines a node that [alpha-beta pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning) would prune, and may prune some branches that alpha-beta would not. Stockman speculated that SSS\* may therefore be a better general algorithm than alpha-beta. However, [Igor Roizen](https://en.wikipedia.org/w/index.php?title=Igor_Roizen&action=edit&redlink=1) and [Judea Pearl](https://en.wikipedia.org/wiki/Judea_Pearl) have shown[[1]](https://en.wikipedia.org/wiki/SSS*#cite_note-1) that the savings in the number of positions that SSS\* evaluates relative to alpha/beta is limited and generally not enough to compensate for the increase in other resources (e.g., the storing and sorting of a list of nodes made necessary by the best-first nature of the algorithm). However, [Aske Plaat](https://en.wikipedia.org/w/index.php?title=Aske_Plaat&action=edit&redlink=1), [Jonathan Schaeffer](https://en.wikipedia.org/wiki/Jonathan_Schaeffer), Wim Pijls and Arie de Bruin have shown that a sequence of null-window alpha-beta calls is equivalent to SSS\* (i.e., it expands the same nodes in the same order) when alpha-beta is used with a [transposition table](https://en.wikipedia.org/wiki/Transposition_table), as is the case in all game-playing programs for chess, checkers, etc. Now the storing and sorting of the OPEN list were no longer necessary. This allowed the implementation of (an algorithm equivalent to) SSS\* in tournament quality game-playing programs. Experiments showed that it did indeed perform better than [Alpha-Beta](https://en.wikipedia.org/wiki/Alpha-beta_pruning) in practice, but that it did not beat [NegaScout](https://en.wikipedia.org/wiki/NegaScout" \o "NegaScout).[[2]](https://en.wikipedia.org/wiki/SSS*#cite_note-2)

The reformulation of a best-first algorithm as a sequence of depth-first calls prompted the formulation of a class of null-window alpha-beta algorithms, of which [MTD-f](https://en.wikipedia.org/wiki/MTD-f) is the best known example.

[https://en.wikipedia.org/wiki/SSS\*](https://en.wikipedia.org/wiki/SSS*)

<https://books.google.be/books?id=wyHSBQAAQBAJ&pg=SA38-PA13&redir_esc=y#v=onepage&q&f=fals> (Gonzalez et al., 2014, pp. 38-12, 38-13)

### Algorithm

There is a [priority queue](https://en.wikipedia.org/wiki/Priority_queue) OPEN that stores states {\displaystyle (J,s,h)} or the nodes, where {\displaystyle J} - node identificator ([Dewey's notation](https://en.wikipedia.org/w/index.php?title=Dewey%27s_notation&action=edit&redlink=1) is used to identify nodes, {\displaystyle \epsilon } is a root), {\displaystyle s\in \{L,S\}} - state of the node {\displaystyle J} (L - the node is live, which means it's not solved yet and S - the node is solved), {\displaystyle h\in (-\infty ,\infty )} - value of the solved node. Items in OPEN queue are sorted descending by their {\displaystyle h} value. If more than one node has the same value of {\displaystyle h}, a node left-most in the tree is chosen.

[https://en.wikipedia.org/wiki/SSS\*](https://en.wikipedia.org/wiki/SSS*)

OPEN := { (e, L, inf) }

while true do // repeat until stopped

pop an element p=(J, s, h) from the head of the OPEN queue

if J = e and s = S then

STOP the algorithm and return h as a result

else

apply Gamma operator for p

Gamma operator for p (J, s, h)  is defined in the following way:

if s = L then

if J is a terminal node then

(1.) add (J, S, min(h, value(J))) to OPEN

else if J is a MIN node then

(2.) add (J.1, L, h) to OPEN

else

(3.) for j=1..number\_of\_children(J) add (J.j, L, h) to OPEN

else

if J is a MIN node then

(4.) add (parent(J), S, h) to OPEN

remove from OPEN all the states that are associated with the children of parent(J)

else if is\_last\_child(J) then // if J is the last child of parent(J)

(5.) add (parent(J), S, h) to OPEN

else

(6.) add (parent(J).(k+1), L, h) to OPEN

// add state associated with the next child of parent(J) to OPEN

## PVS, NegaScout, MTDf(n)

Principal variation search (sometimes equated with the practically identical NegaScout) is a [negamax](https://en.wikipedia.org/wiki/Negamax" \o "Negamax) algorithm that can be faster than [alpha-beta pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning" \o "Alpha-beta pruning). Like alpha-beta pruning, NegaScout is a directional search algorithm for computing the [minimax](https://en.wikipedia.org/wiki/Minimax) value of a node in a [tree](https://en.wikipedia.org/wiki/Tree). It dominates alpha-beta pruning in the sense that it will never examine a node that can be pruned by alpha-beta; however, it relies on accurate node ordering to capitalize on this advantage.

NegaScout works best when there is a good move ordering. In practice, the move ordering is often determined by previous shallower searches. It produces more cutoffs than alpha-beta by assuming that the first explored node is the best. In other words, it supposes the first node is in the [principal variation](https://en.wikipedia.org/wiki/Variation_(game_tree)" \o "Variation (game tree)). Then, it can check whether that is true by searching the remaining nodes with a null window (also known as a scout window; when alpha and beta are equal), which is faster than searching with the regular alpha-beta window. If the proof fails, then the first node was not in the principal variation, and the search continues as normal alpha-beta. Hence, NegaScout works best when the move ordering is good. With a random move ordering, NegaScout will take more time than regular alpha-beta; although it will not explore any nodes alpha-beta did not, it will have to re-search many nodes.

[Alexander Reinefeld](https://en.wikipedia.org/wiki/Alexander_Reinefeld) invented NegaScout several decades after the invention of alpha-beta pruning. He gives a proof of correctness of NegaScout in his book.[[1]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-1)

Another search algorithm called [SSS\*](https://en.wikipedia.org/wiki/SSS*) can theoretically result in fewer nodes searched. However, its original formulation has practical issues (in particular, it relies heavily on an OPEN list for storage) and nowadays most chess engines still use a form of NegaScout in their search. Most chess engines use a transposition table in which the relevant part of the search tree is stored. This part of the tree has the same size as SSS\*'s OPEN list would have.[[2]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-2) A reformulation called MT-SSS\* allowed it to be implemented as a series of null window calls to Alpha-Beta (or NegaScout) that use a transposition table, and direct comparisons using game playing programs could be made. It did not outperform NegaScout in practice. Yet another search algorithm, which does tend to do better than NegaScout in practice, is the best-first algorithm called [MTD-f](https://en.wikipedia.org/wiki/MTD-f), although neither algorithm dominates the other. There are trees in which NegaScout searches fewer nodes than SSS\* or MTD-f and vice versa.

NegaScout takes after SCOUT, invented by [Judea Pearl](https://en.wikipedia.org/wiki/Judea_Pearl) in 1980, which was the first algorithm to outperform alpha-beta and to be proven asymptotically optimal.[[3]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-3)[[4]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-4) Null windows, with β=α+1 in a negamax setting, were invented independently by J.P. Fishburn and used in an algorithm similar to SCOUT in an appendix to his Ph.D. thesis,[[5]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-5) in a parallel alpha-beta algorithm,[[6]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-6) and on the last subtree of a search tree root node.[[7]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-7)

Most of the moves are not acceptable for both players, so we do not need to fully search every node to get the exact score. The exact score is only needed in principal variation (a sequence of moves acceptable for both players) which is expected to propagate down to the root. In iterative deepening search, the previous iteration has already established such a sequence. In a node that has a score that ends up being inside the window, which is called PV-node, we search the first move, which is deemed best, in a full window to establish a bound. That is needed to prove other moves are unacceptable. We conducted a zero window search to test if a move can be better. Since a zero window search is much cheaper, it can save a lot of effort. If we find that a move can raise alpha, we do a re-search with the full window to get the exact score. [[8]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-8) [[9]](https://en.wikipedia.org/wiki/Principal_variation_search#cite_note-9)

Principal variation search <https://en.wikipedia.org/wiki/Principal_variation_search>

function pvs(node, depth, α, β, color) is

if depth = 0 or node is a terminal node then

return color × the heuristic value of node

for each child of node do

if child is first child then

score := −pvs(child, depth − 1, −β, −α, −color)

else

score := −pvs(child, depth − 1, −α − 1, −α, −color) (\* search with a null window \*)

if α < score < β then

score := −pvs(child, depth − 1, −β, −score, −color) (\* if it failed high, do a full re-search \*)

α := max(α, score)

if α ≥ β then

break (\* beta cut-off \*)

return α

Memory-enhanced Test Driver <https://en.wikipedia.org/wiki/MTD-f>

**MTD(f)** is a [minimax](https://en.wikipedia.org/wiki/Minimax) search algorithm developed in 1994 by [Aske Plaat](https://en.wikipedia.org/w/index.php?title=Aske_Plaat&action=edit&redlink=1), [Jonathan Schaeffer](https://en.wikipedia.org/wiki/Jonathan_Schaeffer), [Wim Pijls](https://en.wikipedia.org/w/index.php?title=Wim_Pijls&action=edit&redlink=1), and [Arie de Bruin](https://en.wikipedia.org/w/index.php?title=Arie_de_Bruin&action=edit&redlink=1). Experiments with tournament-quality [chess](https://en.wikipedia.org/wiki/Chess), [checkers](https://en.wikipedia.org/wiki/Checkers), and [Othello](https://en.wikipedia.org/wiki/Reversi) programs show it to be a highly efficient minimax algorithm. The name MTD(f) is an abbreviation for MTD(n,f) (Memory-enhanced Test Driver with node n and value f). It is an alternative to the [alpha-beta pruning](https://en.wikipedia.org/wiki/Alpha-beta_pruning) algorithm.[[1]](https://en.wikipedia.org/wiki/MTD-f#cite_note-F%C3%BCrnkranzKubat2001-1)

MTD(f) was first described in a University of Alberta Technical Report authored by Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin,[[2]](https://en.wikipedia.org/wiki/MTD-f#cite_note-2) which would later receive the ICCA Novag Best Computer Chess Publication award for 1994/1995. The algorithm MTD(f) was created out of a research effort to understand the [SSS\*](https://en.wikipedia.org/wiki/SSS*) algorithm, a best-first search algorithm invented by [George Stockman](https://en.wikipedia.org/w/index.php?title=George_Stockman&action=edit&redlink=1) in 1979.[[3]](https://en.wikipedia.org/wiki/MTD-f#cite_note-GonzalezDiaz-Herrera2014-3) SSS\* was found to be equivalent to a series of [alpha-beta](https://en.wikipedia.org/wiki/Alpha-beta_pruning) calls, provided that alpha-beta used storage, such as a well-functioning [transposition table](https://en.wikipedia.org/wiki/Transposition_table).

The name MTD(f) stands for Memory-enhanced Test Driver, referencing [Judea Pearl](https://en.wikipedia.org/wiki/Judea_Pearl)'s Test algorithm, which performs Zero-Window Searches. MTD(f) is described in depth in Aske Plaat's 1996 PhD thesis.

MTD(f) derives its efficiency by only performing zero-window alpha-beta searches, with a "good" bound (variable beta). In [NegaScout](https://en.wikipedia.org/wiki/NegaScout" \o "NegaScout), the search is called with a wide search window, as in AlphaBeta(root, −INFINITY, +INFINITY, depth), so the return value lies between the value of alpha and beta in one call. In MTD(f), AlphaBeta fails high or low, returning a lower bound or an upper bound on the minimax value, respectively. Zero-window calls cause more cutoffs, but return less information - only a bound on the minimax value. To find the minimax value, MTD(f) calls AlphaBeta a number of times, converging towards it and eventually finding the exact value. A [transposition table](https://en.wikipedia.org/wiki/Transposition_table) stores and retrieves the previously searched portions of the tree in memory to reduce the overhead of re-exploring parts of the search tree.[[4]](https://en.wikipedia.org/wiki/MTD-f#cite_note-ai-4)

[NegaScout](https://en.wikipedia.org/wiki/NegaScout) calls the zero-window searches [recursively](https://en.wikipedia.org/wiki/Recursion_(computer_science)). MTD(f) calls the zero-window searches from the root of the tree. Implementations of the MTD(f) algorithm have been shown to be more efficient (search fewer nodes) in practice than other search algorithms (e.g. NegaScout) in games such as chess [[1]](http://portal.acm.org/citation.cfm?id=240998&coll=GUIDE&dl=GUIDE&CFID=50049687&CFTOKEN=16181537), checkers, and Othello. For search algorithms such as NegaScout or MTD(f) to perform efficiently, the [transposition table](https://en.wikipedia.org/wiki/Transposition_table) must work well. Otherwise, for example when a hash-collision occurs, a subtree will be re-expanded. When MTD(f) is used in programs suffering from a pronounced odd-even effect, where the score at the root is higher for even search depths and lower for odd search depths, it is advisable to use separate values for f to start the search as close as possible to the minimax value. Otherwise, the search would take more iterations to converge on the minimax value, especially for fine grained evaluation functions.

Zero-window searches hit a cut-off sooner than wide-window searches. They are therefore more efficient, but, in some sense, also less forgiving, than wide-window searches. Because MTD(f) only uses zero-window searches, while [Alpha-Beta](https://en.wikipedia.org/wiki/Alpha-beta_pruning) and NegaScout also use wide window searches, MTD(f) is more efficient. However, wider search windows are more forgiving for engines with large odd/even swings and fine-grained evaluation functions. For this reason some [chess engines](https://en.wikipedia.org/wiki/Chess_engine) have not switched to MTD(f). In tests with tournament-quality programs such as [Chinook](https://en.wikipedia.org/wiki/Chinook_(draughts_player)) (checkers), [Phoenix](https://en.wikipedia.org/w/index.php?title=Phoenix_(chess_program)&action=edit&redlink=1) (chess), and [Keyano](https://en.wikipedia.org/w/index.php?title=Keyano&action=edit&redlink=1" \o "Keyano (page does not exist)) (Othello), the MTD(f) algorithm outperformed all other search algorithms.[[4]](https://en.wikipedia.org/wiki/MTD-f#cite_note-ai-4)[[6]](https://en.wikipedia.org/wiki/MTD-f#cite_note-6)

Recent algorithms like [Best Node Search](https://en.wikipedia.org/wiki/Best_Node_Search) is suggested to outperform MTD(f).

function MTDF(root, f, d) is

g := f

upperBound := +∞

lowerBound := −∞

while lowerBound < upperBound do

β := max(g, lowerBound + 1)

g := AlphaBetaWithMemory(root, β − 1, β, d)

if g < β then

upperBound := g

else

lowerBound := g

return g

f

First guess for best value. The better the quicker the algorithm converges. Could be 0 for first call.

d

Depth to loop for. An iterative deepening depth-first search could be done by calling MTDF() multiple times with incrementing d and providing the best previous result in f.[5]

PSEUDOCODE FROM: (Gonzalez et al., 2014, pp. 38-13)

<https://books.google.be/books?id=wyHSBQAAQBAJ&pg=SA38-PA13&redir_esc=y#v=onepage&q&f=false>

## Best Node Search (BNS)

(Rutko, 2011, pp. 94-99)

<https://www.bjmc.lu.lv/fileadmin/user_upload/lu_portal/projekti/bjmc/Contents/770_7.pdf>

Best node search (BNS) is a [minimax](https://en.wikipedia.org/wiki/Minimax) [search algorithm](https://en.wikipedia.org/wiki/Search_algorithm), developed in 2011. Experiments with [random trees](https://en.wikipedia.org/wiki/Random_tree) show it to be the most efficient [minimax](https://en.wikipedia.org/wiki/Minimax) algorithm. This [algorithm](https://en.wikipedia.org/wiki/Algorithm" \o "Algorithm) does tell which move leads to minmax, but does not tell the evaluation of [minimax](https://en.wikipedia.org/wiki/Minimax).[[1]](https://en.wikipedia.org/wiki/Best_node_search#cite_note-1)

[MTD-f](https://en.wikipedia.org/wiki/MTD-f) guesses the minimax by calling zero-window [alpha-beta prunings](https://en.wikipedia.org/wiki/Alpha-beta_pruning" \o "Alpha-beta pruning). BNS calls search that tells whether the minimax in the [subtree](https://en.wikipedia.org/wiki/Subtree" \o "Subtree) is smaller or bigger than the guess. It changes the guessed value until [alpha](https://en.wikipedia.org/wiki/Alpha" \o "Alpha) and [beta](https://en.wikipedia.org/wiki/Beta" \o "Beta) is close enough or only one [subtree](https://en.wikipedia.org/wiki/Subtree" \o "Subtree) allows minimax value bigger than guessed value.

The default nextGuess function above may be replaced with one which uses statistical information for improved performance.

PSEUDOCODE (Rutko, 2011, pp. 95)

function nextGuess(α, β, subtreeCount) is

return α + (β − α) × (subtreeCount − 1) / subtreeCount

function bns(node, α, β) is

subtreeCount := number of children of node

do

test := nextGuess(α, β, subtreeCount)

betterCount := 0

for each child of node do

bestVal := −alphabeta(child, −test, −(test − 1))

if bestVal ≥ test then

betterCount := betterCount + 1

bestNode := child

(update number of sub-trees that exceeds separation test value)

(update alpha-beta range)

while not (β − α < 2 or betterCount = 1)

return bestNode

## Monte-Carlo tree search ?

# Unity ML Agents

To make the framework even more adaptable it would be nice if you could just plug in any game, add the rules, and get a good AI without needing to code heuristics or algorithms. Machine Learning might make this possible.

Unity ML Agents seemed to fit this project, as Dr. Danny Lange (VP of AI and Machine Learning at Unity Technologies) puts it: “Unity ML-Agents offers a flexible way to develop and test new AI algorithms quickly and efficiently across a new generation of robotics, games, and beyond”.

The main advantage is that the Unity community is very active so in normal circumstances a lot of resources and examples can be found on the internet. But during this project it became clear it is not really the case for ML Agents. Although ML Agents appeared in 2017 it is still in beta and it thus exists for three years now, the framework is still buggy, and the documentation is far from on point.

Agents can be trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning methods through a simple-to-use Python API. Unity additionally offers implementations (based on TensorFlow) of state-of-the-art algorithms to enable game developers and hobbyists to easily train intelligent agents for 2D, 3D and VR/AR games. These trained agents can be used for multiple purposes, including controlling NPC (non-player character) behavior (in a variety of settings such as multi-agent and adversarial), automated testing of game builds and evaluating different game design decisions pre-release.

Use Cases

Unity ML-Agents can benefit:

Academic researchers interested in studying complex multi-agent behavior in realistic competitive and cooperative scenarios.

Industry researchers interested in large-scale parallel training regimes for robotics, autonomous vehicle, and other industrial applications.

Game developers interested in filling virtual worlds with intelligent agents each acting with dynamic and engaging behavior.

Features

The Unity ML-Agent Toolkit is an open-source solution with the following features:

Unity environment control from Python

10+ sample Unity environments

Support for multiple environment configurations and training scenarios

Train memory-enhanced agents using deep reinforcement learning

Easily definable Curriculum Learning scenarios

Broadcasting of agent behavior for supervised learning

Built-in support for Imitation Learning

Flexible agent control with On Demand Decision Making

Visualizing network outputs within the environment

Simplified set-up with Docker

Wrap learning environments as a gym

Required software (20/07/2020)

* Unity 2018.4 or Later
* Unity MLAgents package
* Python 3.6/3.7 (64 bit)

Creating a Python virtual environment

* Create a folder for your virtual environments

*C:\python-envs*

* Create a new virtual environment

*python -m venv C:\python-envs\mlagents-env*

Using the virtual environment

* Activate using

*C:\python-envs\mlagents-env\Scripts\activate*

* Deactivate using

*deactivate*

Prepare using mlagents

* Install mlagents

*pip3 install mlagents*

<https://github.com/Unity-Technologies/ml-agents/blob/release_4_docs/docs/Getting-Started.md>

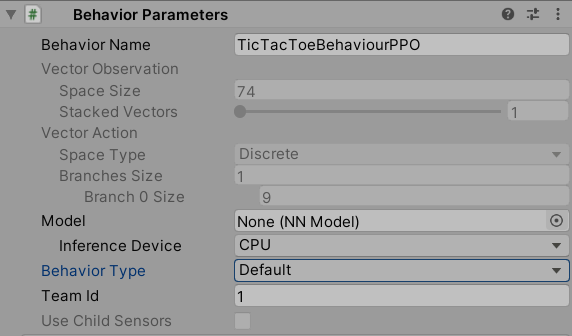
<https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.Agent.html#Unity_MLAgents_Agent_CollectObservations_Unity_MLAgents_Sensors_VectorSensor_>

<https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md>

## Agent

### Behavior parameters

When adding an agent script to a game object a Behavior Parameters component will appear as seen in the image below. Following the image is a short description for each behavior parameter.



Behaviour parameters (Screenshot)

#### Behavior name

The name of the behavior, this links up to a behavior in the .yaml file. If the behavior name does not link up to a behavior in the .yaml file, the agent will not train.

#### Vector observation

This parameter defines the structure of the observations. The space size is the number of observed variables per observation. The stacked vectors parameter is the amount of old vector observations that are used as inputs for the neural network. For example, a stacked vectors parameter of 2 will feed both the current observations and the previous observations to the network.

#### Vector action

This parameter defines the action space. This action space can either be discrete (a fixed amount of actions) or continuous (an infinite amount of actions based on a float input value). The branches size sub parameter represents the amount of actions being taken while the branch size represents the amount of possible actions for each action being taken.

#### Model

This parameter holds a reference to a trained neural network, using the network to determine the float[] actionVector in the Agent.OnActionRecieved(float[] actionVector ) function.

#### Behavior type

The behavior type enum has three possible states: default, heuristic only and interference only. The default value is used for training, heuristic only will make the agent fall back on the Agent.Heuristic(float[] actionsOut) function (which should be defined in the agent’s script) and the interference only will use the neural network without training it.

#### Team ID

This variable represents the team of the agent. Agents that work together have the same team ID. The team ID should be equal or greater than zero, otherwise the agent will not train in some cases.

#### Use child sensors

This parameter is there when using ray perception sensors or render texture sensors. If the parameter is toggled to true, the agent will look for sensors in the components of its child transforms.

## Training

### PPO vs SAC

<https://blogs.unity3d.com/2019/11/11/training-your-agents-7-times-faster-with-ml-agents/>

The primary section of the trainer config file is a set of configurations for each Behavior in your scene. These are defined under the sub-section behaviors in your trainer config file. Some of the configurations are required while others are optional. To help us get started, below is a sample file that includes all the possible settings if we are using a PPO trainer with all the possible training functionalities enabled (memory, behavioral cloning, curiosity, GAIL, and self-play). You will notice that curriculum and environment parameter randomization settings are not part of the behavior’s configuration, but in their own section called environment\_parameters.

Below is a typical yaml file for a PPO setup:

behaviors:

BehaviorPPO:

trainer\_type: ppo

hyperparameters:

batch\_size: 1024

buffer\_size: 10240

learning\_rate: 3.0e-4

learning\_rate\_schedule: linear

beta: 5.0e-3

epsilon: 0.2

lambd: 0.95

num\_epoch: 3

network\_settings:

vis\_encoder\_type: simple

normalize: false

hidden\_units: 128

num\_layers: 2

memory:

sequence\_length: 64

memory\_size: 256

max\_steps: 5.0e5

time\_horizon: 64

summary\_freq: 10000

keep\_checkpoints: 5

checkpoint\_interval: 50000

threaded: true

init\_path: null

Below is a typical SAC setup:

behaviors:

BehaviorSAC:

trainer\_type: sac

hyperparameters:

batch\_size: 1024

buffer\_size: 10240

learning\_rate: 3.0e-4

learning\_rate\_schedule: linear

buffer\_init\_steps: 0

tau: 0.005

steps\_per\_update: 10.0

save\_replay\_buffer: false

init\_entcoef: 0.5

reward\_signal\_steps\_per\_update: 10.0

network\_settings:

vis\_encoder\_type: simple

normalize: false

hidden\_units: 128

num\_layers: 2

memory:

sequence\_length: 64

memory\_size: 256

max\_steps: 5.0e5

time\_horizon: 64

summary\_freq: 10000

keep\_checkpoints: 5

checkpoint\_interval: 50000

threaded: true

init\_path: null

To use self-play the following parameters should be added (possible with PPO and SAC):

self\_play:

window: 10

play\_against\_latest\_model\_ratio: 0.5

save\_steps: 50000

swap\_steps: 2000

team\_change: 100000

### Training parameters

When training an agent, you must set the training parameters in the .yaml file. These parameters determine both the agent’s neural network as how it is trained. The .yaml file is used by the Python environment to do the training. Following are the training parameters for the PPO setup with a short description of how to use them (Mattar et al., 2018).

#### Gamma

The gamma value (gamma) is the discount factor for future rewards. If the agent is more interested in future rewards this value should be bigger, if the agent is more interested in short term rewards the value should be smaller. Typical values are ranging from 0.8 to 0.955.

#### Lambda

The lambda value (lambd) is used when calculating the General Advantage Estimate (GAE). The value determines how much weight the agent gives to the current value estimate when updating based on new training data. Low values lead to giving more importance on the current value (which can lead to a high bias) while high values lead to giving more importance to the new data (which can lead to a high variance). It is important to find a good balance, common values range from 0.9 to 0.95.

#### Buffer Size

The buffer size (buffer\_size) is the amount of experiences (observations, actions, and rewards) that should be obtained before updating the model. The value should be a multiple of the batch size (see below). Typical values range between 2048 to 409600.

#### Batch Size

The batch size (batch\_size) is the amount of experiences (observations, actions, and rewards) used for one iteration of gradient decent update. This should be a factor of the buffer size. For agents with a continuous action space this should be a large value (typically between 512 and 5120), while for agents with a discrete action space thee value should be lower (typically between 32 and 512).

#### Number of Epochs

The number of epochs (num\_epoch) represent the amount of experience buffer passes during gradient descent. A larger batch size makes it acceptable to increate this number. The higher the number the faster the agent will learn, but at the cost of less stable updates. This number typically ranges between 3 and 10.

#### Learning Rate

The learning rate (learning\_rate) is the strength of the gradient descent update step. If the training is unstable or if the reward does not consistently increase this value should be decreased. The learning rate typically ranges from 10-5 to 10-3.

#### Time Horizon

The time horizon (time\_horizon) defines to the amount of experience steps that need to be collected per agent before sending them to the experience buffer. If the end of an episode is reached before the end of an episode an expected reward will be used based on a value estimate of the agent’s current stage. The time episode should at least be big enough to capture the important behavior of a sequence of an agent’s actions. For our games this corresponds to the length of a single game.

#### Max Steps

Max steps (max\_steps) represents the number of steps the training process will take. More complex problems require more steps for the agent to get fully trained. Typically, this value ranges from 5 \* 105 to 107.

#### Beta

The beta value (beta) links to the entropy regularization which determines the randomness of the policy. The randomness makes sure that all possible actions are explored during training. Increasing beta will add more random actions. The entropy of the training (see TensorBoard) should decrease slowly over the training, if it drops to fast, increase beta, if it drops to slow, decrease beta. Values commonly range between 10-4 and 10-2.

#### Epsilon

Epsilon represents the acceptable threshold of divergence between old and new policies when updating using gradient descent. The smaller the value the more stable the updates but the slower the training. Typical values range from 0.1 to 0.3.

#### Normalize

Normalize is a bool parameter that represents whether normalization should be applied to observation inputs. The normalization is based on the running average and the variance of the vector observation. Normalization might be harmful with simple discrete control problems but might be helpful when the problem is more complex and continuous.

#### Number of Layers

The number of layers (num\_layers) defines the depth of the neural network. The more complex the problem, the more layer might be required. More layers require more training time. The number of layers typically ranges from 1 to 3

#### Hidden Units

The number of hidden units (hidden\_units) defines the width of the neural network, the number of neurons in each layer. The more interaction there is between the observation variables the more hidden units are required. A typical neural network has between 32 and 512 hidden units.

#### Recurrent Neural Network Hyperparameters

These hyperparameters are optional and are only used when use\_recurrent is set to true. The parameters are the sequence length (sequence\_lenght) and the memory size (memory\_size). The sequence length is the length of the experience sequence that is passed to the network for training. The length should be big enough to contain all important information from the beginning of the episode until the end of the episode. The memory size is the size of the float array that holds the hidden state of the recurrent neural network (RNN). The value of the memory should always be a multiple of 4.

#### Intrinsic Curiosity Module Hyperparameters

Only when use\_curiosity is set to true these hyperparameters are used. The curiosity encoding size (curiosity\_enc\_size) represents the size of the hidden layer used for encoding the observations within the curiosity module. A typical value for the encoding size lays between 64 and 256. The curiosity strength (curiosity\_strength) represents the magnitude of the generated intrinsic reward. The value should be balanced to the extrinsic reward signals so that neither of them overwhelms the other. Typical values are between 0.1 and 0.001.

### Self-play

Self-play

<https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-agents/>

Improvements

<https://www.codeproject.com/Articles/5160398/A-Tic-Tac-Toe-AI-with-Neural-Networks-and-Machine>

## Results

When the maximum amount of steps is reached the training is done and ML Agents will create a .nn file for each agent. This file contains the trained neural network. To test the neural network you drag the file into the model parameter of the agent. If you think the agents needs more training you can continue training the network by increasing the maximum amount of steps in the .yaml file and just resume the training (using the –resume command). If you want more information about what happened during the training process you can use Tensor Board.

### Tensor Board

To open Tensor Board after the training ended, drop the following line in the command prompt:

tensorboard --logdir=results

This will start a host a local webpage which can be opened in any browser by copy-pasting “localhost:6006” in the address bar. On this webpage you will find graphs of how certain parameters evolved during training for the different agents you trained. The following parameters are displayed (AurelianTactics, 2018; Mattar et al., 2018):

#### Cumulative Reward

The cumulative reward is total amount of rewards the agent collected each step. In a normal training scenario this should consistently increase, but in a game scenario when the agent is playing against another training agent the reward might go down because the other agent is improving faster.

#### Entropy

Entropy represents the randomness in the agents decisions. This should decrease over time. If the entropy goes down to fast or does not decrease when using a discrete action space, the beta parameter should be adjusted in the .yaml file.

#### Learning Rate

In [machine learning](https://en.wikipedia.org/wiki/Machine_learning) and [statistics](https://en.wikipedia.org/wiki/Statistics), the **learning rate** is a [tuning parameter](https://en.wikipedia.org/wiki/Hyperparameter_(machine_learning)) in an [optimization algorithm](https://en.wikipedia.org/wiki/Mathematical_optimization) that determines the step size at each iteration while moving toward a minimum of a [loss function](https://en.wikipedia.org/wiki/Loss_function).[[1]](https://en.wikipedia.org/wiki/Learning_rate#cite_note-1) Since it influences to what extent newly acquired information overrides old information, it metaphorically represents the speed at which a machine learning model "learns".[[2]](https://en.wikipedia.org/wiki/Learning_rate#cite_note-towardsdatascience2-2) In the [adaptive control](https://en.wikipedia.org/wiki/Adaptive_control) literature, the learning rate is commonly referred to as **gain**.[[3]](https://en.wikipedia.org/wiki/Learning_rate#cite_note-3)

<https://en.wikipedia.org/wiki/Learning_rate>

#### Policy Loss

The mean magnitude of policy loss function. Correlates to how much the policy (process for deciding actions) is changing. The magnitude of this should decrease during a successful training session. These values will oscillate during training. Generally they should be less than 1.0.

#### Value Estimate

The mean value estimate for all states visited by the agent. Should increase during a successful training session. These values should increase as the cumulative reward increases. They correspond to how much future reward the agent predicts itself receiving at any given point.

#### Value Loss

The mean loss of the value function update. Correlates to how well the model is able to predict the value of each state. This should increase while the agent is learning, and then decrease once the reward stabilizes. These values will increase as the reward increases, and then should decrease once reward becomes stable.

# The demo

## Framework

The demo is a Unity framework that allows to quickly build board games.

The framework allows to quickly create a board by changing the colors on the Square-prefab and linking the game in the Board-component. The board will construct itself based on the board size of the game.

It allows to quickly build a game by creating a subclass of Game. There are a only few abstract functions that need to be implemented. List<Position> GetPossibleMoves(Board board, IGameAgent gameAgent) has to return the possible moves based on the board state. State GetState(Board board, IGameAgent gameAgent) needs to return the board state (win/loss/draw/playing) of a certain position based on who is the active agent. float GetScore(Board board, IGameAgent gameAgent, State gameState) is the heuristic function used by the algorithmic agents.

Agents can be added by making an derived class from GameAgent or one of it’s subclasses (AlgorithmAgent/MLAgent). The framework also contains some premade agents, most of them can be plugged into any game, while the others can be used for reference.

### Algorithmic agents

All the described algorithms described in this paper also have a functional agent in the framework. Most of these agents are game independent, so they can be quickly used when adding a new game. To use one of the agents just drag and drop the agent game object in the list of players in the game component.

### Machine learning agents

To add a machine learning agent you have to implement the MLAgent class, a derived class from the ML Agents agent class. There are two subclasses that already have some extra functionality, the ActorAgent and the EvaluatorAgent. The actor agent uses a discreet action space that represents the possible actions and plays the chosen action/move, while the evaluator agent uses a continuous action space that represents the score for a certain possible move and then plays the highest scoring move. The actor agent trains faster, but the evaluator agent might be interesting to do further research or when you want to rank the possible moves from bad to worse.

### Training

Working on a machine learning agent can be very challenging for a beginner. There are many parameters and training methods and each training session takes a lot of time. When the results are not as expected there is always the doubt if it was due to the training parameters or because the training session was too short. The whole process is very trail-and-error based. Luckily, while the agent is training there is a lot of time to browse through the forums for new ideas.

In a way these ideas sometimes lead to a big improvement, other times they do not seem to change anything or even worsen the agent’s performance. Testing takes time and thus sometimes a good idea might be thrown away because there was no immediate result. During this research it became clear though that making a machine learning AI solution takes way more time than creating an algorithm-based AI. When working with a lot of resources and time on a complex game, like the AlphaGo project, it is definitely worth using machine learning. A machine learning AI might come with moves that go beyond human creativity.

- Starting agent vs Responding agent

- Agent vs Random

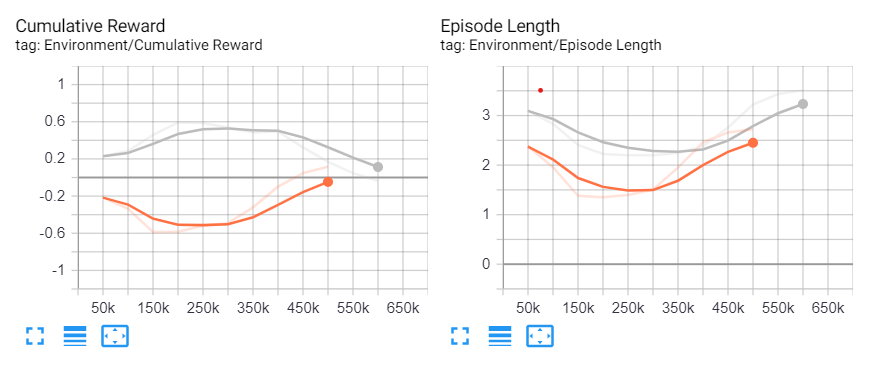
- Continuous evaluation instead of discrete actions

- Self play

- SAC instead of PPO

- Tournament play

Tic Tac Toe:



Behavior A trained against Behavior B with Behavior A always being the starting player.

|  |  |  |  |
| --- | --- | --- | --- |
| 1000 < Games | Beginner wins | Draw | Beginner loses |
| Perfect vs Perfect | 100.0% | 0.0% | 0.0% |
| Random vs Random | 58.2% | 11.7% | 30.2% |
| Random vs Behavior A | 21.4% | 5.0% | 73.6% |
| Random vs Behavior B | 49.9% | 16.5% | 33.6% |
| Behavior A vs Random | 96.0% | 2.2% | 1.8% |
| Behavior A vs Behavior A | 97.5% | 1.7% | 0.8% |
| Behavior A vs Behavior B | 21.0% | 49.4% | 29.6% |
| Behavior B vs Random | 81.0% | 10.6% | 8.4% |
| Behavior B vs Behavior A | 67.9% | 17.1% | 14.9% |
| Behavior B vs Behavior B | 80.4% | 10.4% | 9.2% |

These are some weird results, at least it is visible the training actually worked. Some remarks:

- You would expect Behavior vs Behavior matches to have only one outcome, but they seem to be creative. As for now we are clueless why this is happening.

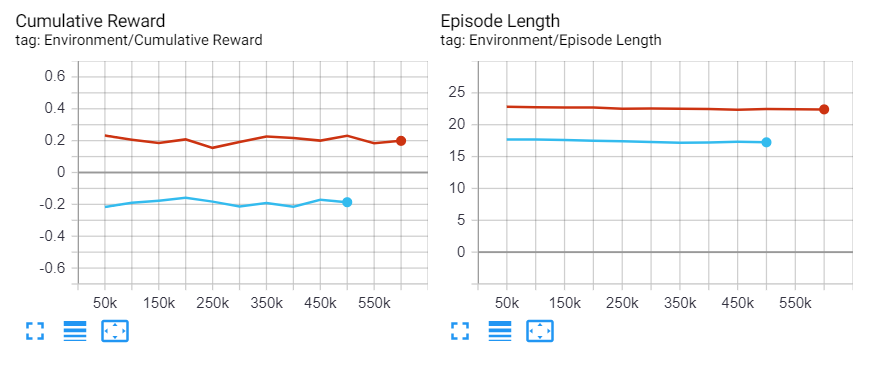
- Although Behavior B was trained by always responding to the moves of Behavior A it also performs okay in the starting position.

- Behavior B became an expert in drawing/winning to Behavior A (79% of the matches) but has more trouble drawing/winning against random moves (50% of the matches) which is clearly a case of overfitting.

- Although these results look good these behaviors still miss finishing/blocking every three in a row. The behaviors do not understand the game of Tic-tac-toe, they just have an incomplete statistical idea of which moves are good. Many previous tests are not reported because the results looked worthless while probably, they would have given a win rate just above Random vs Random.

The next agent is an evaluator that gives a score to each board position. These evaluations then get used to select the best move.

-> try rotations etc.

-> try spectator view  


|  |  |  |  |
| --- | --- | --- | --- |
| 1000 < Games | Beginner wins | Draw | Beginner loses |
| Perfect vs Perfect | 100.0% | 0.0% | 0.0% |
| Random vs Random | 58.2% | 11.7% | 30.2% |
| Random vs Behavior A | 50.1% | 7.9% | 42.0% |
| Random vs Behavior B | 49.1% | 9.8% | 41.0% |
| Behavior A vs Random | 68.2% | 9.1% | 22.7% |
| Behavior A vs Behavior A | 71.2% | 10.0% | 18.8% |
| Behavior A vs Behavior B | 55.8% | 11.8% | 32.4% |
| Behavior B vs Random | 69.1% | 11.7% | 19.2% |
| Behavior B vs Behavior A | 61.1% | 11.8% | 27.1% |
| Behavior B vs Behavior B | 60.7% | 12.4% | 27.0% |

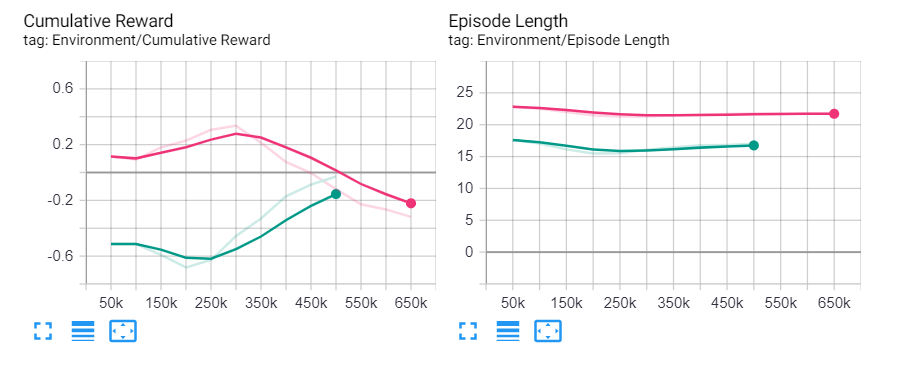
Agent that turns the board around so that it only has should evaluate moves on the following positions:

000

0X0

XX0

Because of the symmetry…



|  |  |  |  |
| --- | --- | --- | --- |
| 1000 < Games | Beginner wins | Draw | Beginner loses |
| Perfect vs Perfect | 100.0% | 0.0% | 0.0% |
| Random vs Random | 58.2% | 11.7% | 30.2% |
| Random vs Behavior A |  |  |  |
| Random vs Behavior B | 27.6% | 21.9% | 50.5% |
| Behavior A vs Random | 89.4% | 4.0% | 6.6% |
| Behavior A vs Behavior A |  |  |  |
| Behavior A vs Behavior B |  |  |  |
| Behavior B vs Random | 78.7% | 11.8% | 9.5% |
| Behavior B vs Behavior A |  |  |  |
| Behavior B vs Behavior B |  |  |  |

Less training (only 50.000 steps 32 neurons x 3 layers)

A vs R 73.1% 10.0% 16.9%

A vs R 77.5% 9.1% 13.4%

A vs R 64.1% 11.0% 24.8%

Second agent lags behind while this should not be happening

*C:\python-envs\mlagents-env\Scripts\activate*

cd C:\Github\BoardGameAI\Four\Assets\ML-Agents

mlagents-learn C:\Github\BoardGameAI\Four\Assets\ML-Agents\Tic-tac-toeBehavior.yaml --run-id=Temp0

Self-play introducing a lot of new problems

self\_play:

save\_steps: 5000

team\_change: 10000

swap\_steps: 2000

window: 10

When training the random agent will always win some games since the agent itself has some random moves when training, although the amount of wins by the random agent should decrease over time.   
After 1.000.000 games both agents were clearly improving, yet they still weren’t able to stop/finish every three in a row. After 1.265.545 games a random agent was added to avoid overfitting. It should be mentioned that adding extra agents slows down the training.

First result was:

TicTacToeCNNAgentPPO: 344

TicTacToeCNNAgentSAC: 471

RandomAgent: 80

Draws: 105

This clearly means that there is still room to improve, yet the agents lose only 8% of the games vs the random agent.

At step 3.381.018 the results were:

TicTacToeCNNAgentPPO: 335

TicTacToeCNNAgentSAC: 467

RandomAgent: 7

Draws: 192

The agents became really good at removing any win chances from the random agent. Yet the agents weren’t drawing against each other. It could have been the case that the agents were prioritizing winning, even by playing bad moves when countered correctly, since the random agent would not counter correctly in most cases. Therefor a new agent was added, a perfect player, using the NegaMax with AlphaBeta and transposition tables. This agent will punish opportunistic behavior. The first results with this agent added are:

RandomAgent: wins: 11 - losses: 465

TTNegaMaxAlphaBetaAgent: wins: 169 - losses: 0

TicTacToeCNNAgentSAC: wins: 243 - losses: 26

TicTacToeCNNAgentPPO: wins: 161 - losses: 93

Draws: 417

Notice how the new agent has a very small amount of wins, although it is a perfect player. This is because the agent does not optimize for win-chances but takes the first branch that leads to the highest possible reward, instead of taking the branch where a non-perfect player has a lower chance of winning. The SAC agent seems to be able to predict moves with a high probability for the opponent to make mistakes.

# Conclusion

The framework provided with this paper makes it very easy to implement a board game that first the given requirements: a maximum size of 8 by 8.

# References

Wikipedia contributors. (2020, July 2). Alpha–beta pruning. In *Wikipedia, The Free Encyclopedia*.

<https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=965591416>

Wikipedia contributors. (2019, June 27). Killer heuristic. In *Wikipedia, The Free Encyclopedia*.

<https://en.wikipedia.org/w/index.php?title=Killer_heuristic&oldid=903761614>

Wikipedia contributors. (2020, June 21). Minimax. In *Wikipedia, The Free Encyclopedia*.

<https://en.wikipedia.org/w/index.php?title=Minimax&oldid=963711709>

Heineman, G. T., Pollice, S., & Selkow, G. (2016). Algorithms in a Nutshell (2nd edition). O’Reilly. <http://www.r-5.org/files/books/computers/algo-list/common/Heineman_Pollice_Selkow-Algorithms_in_a_Nutshell-EN.pdf>

AurelianTactics. (2018, 13 december). Understanding PPO Plots in TensorBoard. Medium.

<https://medium.com/aureliantactics/understanding-ppo-plots-in-tensorboard-cbc3199b9ba2>

Gonzalez, T., Diaz-Herrera, J., & Tucker, A. (2014). Computing Handbook, Third Edition. Taylor & Francis. <https://books.google.be/books?id=wyHSBQAAQBAJ&printsec=frontcover&hl=nl#v=onepage&q&f=false>

Mattar, M., Juliani, A., Berges, V., Pang, D., Esh, Ward, J., & Shih, J. (2018, October 19). *Training with Proximal Policy Optimization*. GitHub. <https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md>

Rutko, D. (2011). *Fuzzified Algorithm for Game Tree Search with Statistical and Analytical Evaluation*. Scientific Papers, University of Latvia, 770, 90–111.

<https://www.bjmc.lu.lv/fileadmin/user_upload/lu_portal/projekti/bjmc/Contents/770_7.pdf>

Tam, A. S. (2019, March 15). *Tic-tac-toe using AI from the last century.* www.adrian.idv.hk. [https://www.adrian.idv.hk/2019-03-15-Tic-tac-toe/](https://www.adrian.idv.hk/2019-03-15-tictactoe/)

Thill, M. (2008, February 14*). Constructing Agents for Connect-4: Board Representations. markusthill.github.io.* <https://markusthill.github.io/programming/connect-4-board-representations/>