THE WORKING DOCUMENT

[**Topic**](#_anxrkb4j5ld9) **2**

[**The Game**](#_rlbrgo1e5zh9) **2**

[Introduction](#_nuv4mms6slc6) 2

[Rules](#_xkia23acrnlj) 2

[Gameplay](#_uohl865tol94) 2

[Computer implementations](#_u8mt7fm6pd4b) 3

[**Overview**](#_jw2futbp7edo) **4**

[Arm](#_42xsgy4j3yr3) 4

[Head](#_utj04svz983m) 4

[**Implementation**](#_10tpl9ho5dsp) **4**

[The Head : Vision](#_cdfumo30lqkg) 5

[The Arm : Forward and Inverse Kinematics](#_ofax92y9sjn1) 6

[Groot : The Two who are One, The One that is All](#_ro4qtw8b0rvr) 8

[The skeleton of the Hardware implementation](#_4iqo8rbipg9v) 8

[The skeleton of the Software implementation](#_9q9mlpq0qbub) 8

[Generic](#_iy13e1eimlbn) 8

[Vision-based](#_u9ebuvurwmhn) 10

[Hardware-based](#_hfmpad6erzvx) 10

[**Further... (if time permits)**](#_6qpucn2nlgnl) **11**

[**References**](#_rw7myy99qh50) **11**

[**Outcome**](#_j6gizahv8ply) **11**

# Topic

A robotic arm (of ‘Groot’) to play the mega-tic-tac-toe.

# The Game

Ref: <https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe>

## Introduction

The Mega/Ultimate tic-tac-toe is a board game composed of nine [tic-tac-toe](https://en.wikipedia.org/wiki/Tic-tac-toe) boards arranged in a 3-by-3 grid. Players take turns playing in the smaller tic-tac-toe boards until one of them wins in the larger tic-tac-toe board. Compared to traditional tic-tac-toe, strategy in this game is conceptually more difficult, and has proven more challenging for computers.

## Rules

Each small 3-by-3 tic-tac-toe board is referred to as a local board, and the larger 3-by-3 board is referred to as the global board.

The game starts with X playing wherever they want in any of the 81 empty spots. This move 'sends' their opponent to its relative location. For example, if X played in the top right square of their local board, then O needs to play next in the local board at the top right of the global board. O can then play in any one of the nine available spots in that local board, each move sending X to a different local board.

If a move is played so that it is to win a local board by the rules of normal [tic-tac-toe](https://en.wikipedia.org/wiki/Tic-tac-toe), then the entire local board is marked as a victory for the player in the global board.

Game play ends when either a player wins the global board or there are no legal moves remaining, in which case the game is a draw.

## Gameplay

Ultimate tic-tac-toe is significantly more complex than most other variations of tic-tac-toe, as there is no clear strategy to playing. This is because of the complicated [game branching](https://en.wikipedia.org/wiki/Branch_(computer_science)) in this game. Even though every move must be played in a local board, equivalent to a normal tic-tac-toe board, each move must take into account the global board in several ways:

1. **Anticipating the next move:** Each move played in a local board determines where the opponent's next move may be played. This might make moves that may be considered bad in normal tic-tac-toe viable, since the opponent is sent to another local board, and may be unable to immediately respond to them. Therefore, players are forced to consider the larger game board instead of simply focusing on the local board.
2. **Visualizing the game tree:** Visualizing future branches of the [game tree](https://en.wikipedia.org/wiki/Game_tree) is more difficult than single board tic-tac-toe. Each move determines the next move, and therefore reading ahead—predicting future moves—follows a much less linear path. Future board positions are no longer interchangeable, each move leading to starkly different possible future positions. This makes the game tree difficult to visualize, possibly leaving many possible paths overlooked.
3. **Winning the game:** Due to the rules of ultimate tic-tac-toe, the global board is never directly affected. It is only governed by actions that occur in local boards. This means that each local move played is not intended to win the local board, but to win the global board. Local wins are not valuable if they cannot be used to win the global board—in fact, it may be strategic to sacrifice a local board to your opponent in order to win a more important local board yourself. This added layer of complexity makes it harder for humans to analyze the relative importance and significance of moves, and consequently harder to play well.

## Computer implementations

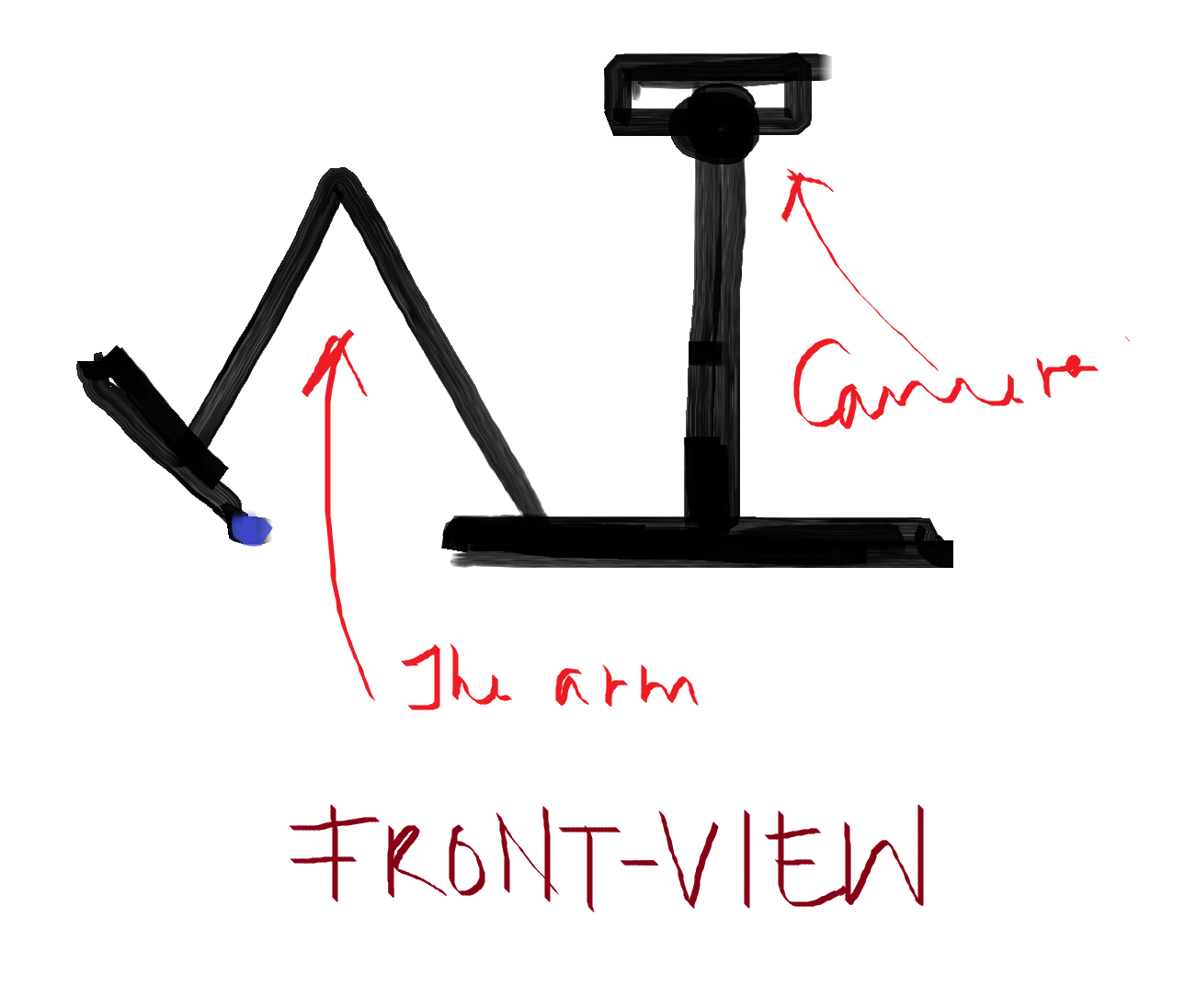
While tic-tac-toe is elementary to solve, and can be done nearly instantly using [depth-first search](https://en.wikipedia.org/wiki/Depth-first_search), ultimate tic-tac-toe cannot be reasonably solved using any brute force tactics. Therefore, more creative computer implementations are necessary in order to play this game.

The most common [artificial intelligence](https://en.wikipedia.org/wiki/Artificial_intelligence) (AI) tactic, [minimax](https://en.wikipedia.org/wiki/Minimax), may be used to play ultimate tic-tac-toe, but has difficulty playing this. This is because, despite having relatively simple rules, ultimate tic-tac-toe lacks any simple [heuristic evaluation function](https://en.wikipedia.org/wiki/Evaluation_function). This function is necessary in minimax, for it determines how good a specific position is. Although elementary evaluation functions can be made for ultimate tic-tac-toe by taking into account the number of local victories, these largely overlook positional advantage that is much harder to quantify. Without any efficient evaluation function, most typical computer implementations are weak, and therefore there are few computer opponents that can consistently outplay humans.

However, artificial intelligence algorithms that don't need evaluation functions, like the [**Monte Carlo tree-search algorithm**](https://en.wikipedia.org/wiki/Monte_Carlo_tree_search), have no problem in playing this game. The Monte Carlo tree-search relies on random simulations of games in order to determine how good a position is instead of a positional evaluation, and is therefore able to accurately assess how good a current position is. Therefore, computer implementations using these algorithms tend to outperform minimax solutions, and can consistently beat human opponents.

# Overview

Groot has 2 parts/components; the first being the arm and the second being the eye/head.



## Arm

The arm holds the marker and marks an X/O in the box (of the 81 boxes) that it is supposed to.

## Head

Just comprises of a camera. That looks at the board constantly. If there is a difference from the previous image (ie; the human opponent has played), then it leans forward to take better (more accurate) picture of the board

# Implementation

1. The Head : Vision
2. The Arm : Forward and Inverse Kinematics
3. Groot : The Two who are One, The One that is All

## 

## The Head : Vision

Input : the current image of the board

Output : If there is a difference from the previous image (ie; the human opponent has played), then return which of the 81 boxes would be played next

Procedure:

Would be split into 2 parts

Let the human mark with red and Groot mark with blue.

F1 : An image is captured every 10 milliseconds, whenever the full board is visible, and compared with the previous image captured. The subtraction of the two images is checked for differences in the red marking.

Case 1 : No difference

GOTO F1;

Case 2 : Spotted difference

Go to the Lean-in position

Capture new\_picture

Go back to initial position

GOTO F2(new\_picture);

The function F2 follows:

int F2(new\_picture)

{

x = local box in which the mark was made by the opponent/human

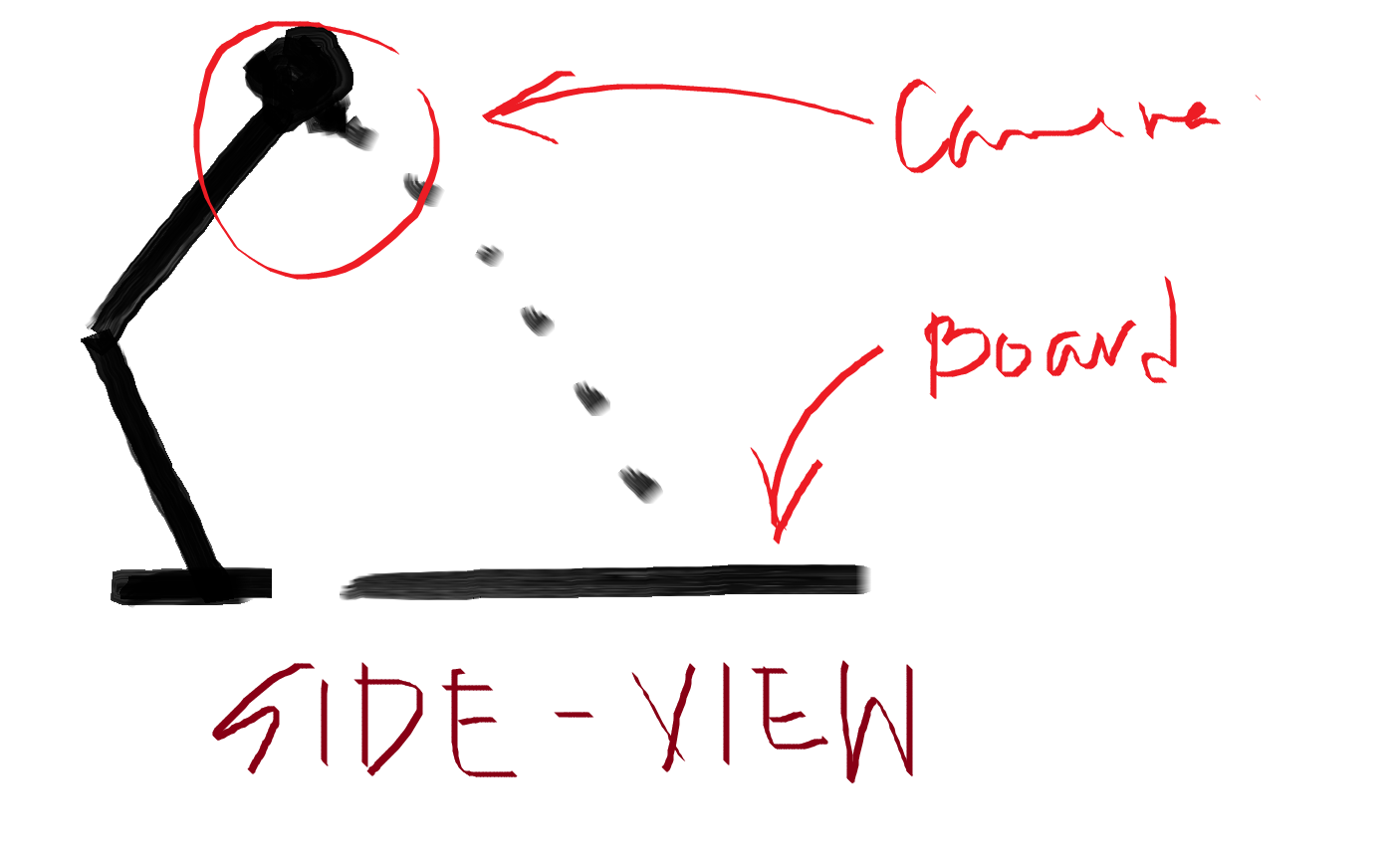
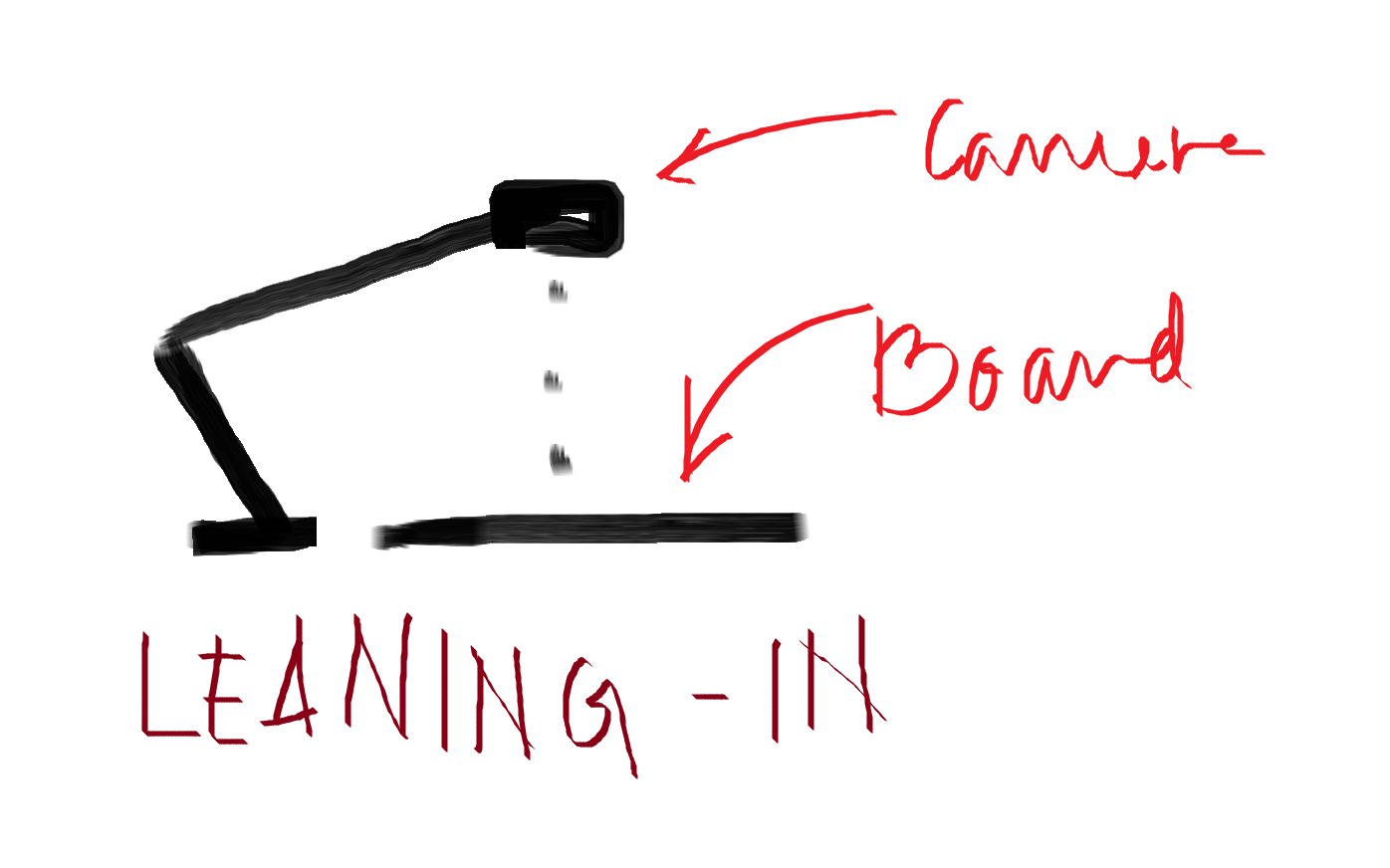
y = global box in which the mark has to be made accd. to x

z = random local box of the global box that is empty

res = z in terms of 0-80 boxes

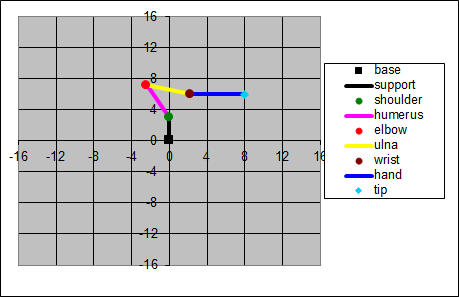
return res;

}

## The Arm : Forward and Inverse Kinematics

Lynx robot-arm



|  |  |  |
| --- | --- | --- |
| **Arm Geometry** | |  |
| H (height of base) | 3.00 |  |
| L1,L2 (hum & ulna) | 4.75 |  |
| L3 (hand) | 5.75 |  |
| S (range -s to s deg) | 80.00 |  |
|  |  |  |
| **Input** |  |  |
| X (hor) | 8.00 |  |
| Z (ver) | 6.00 |  |
| P (ang of hand wrt gnd deg) | 0.00 |  |
|  |  |  |
| **Computation** | | **Formula Used** |
| Xb | 0.24 | (X - L3 \* cos(P)) / (2 \* L1) |
| Zb | 0.32 | (Z - H - L3 \* sin(P)) / (2 \* L1) |
| Q | 2.33 | sqrt(1 / (Xb^2 + Zb^2) - 1) |
| P1 | 119.88 | atan2(Xb - Q \* Zb, Zb + Q \* Xb) |
| P2 | -13.62 | atan2(Xb + Q \* Zb, Zb - Q \* Xb) |
| T1 | 29.88 | P1 - 90 |
| T2 | -43.50 | P2 - T1 |
| T3 | 13.62 | P - P2 |
|  |  |  |
| **Output** |  |  |
| T1 (shoulder ang) | 29.88 | T1 - round(T1/360)\*360 |
| T2 (elbow ang) | -43.50 | T2 - round(T1/360)\*360 |
| T3 (wrist ang) | 13.62 | T3 - round(T1/360)\*360 |

**In groot’s case;**

def above\_to\_code (**X, Z, P**)

{

Calc….

return **[T1, T2, T3]**

}

def our\_func (**x, y, z**)

{

ang\_marker\_handle = 50 ……………?# to be set

base\_ang = atan2(x,y) ……………?# is it atan2(x,y) or atan2(y,x)

X = sqrt ( (x\*x) + (y\*y) )

[sld, elb, wrt] = above\_to\_code (X, Z, ang\_marker\_handle)

return **[base\_ang, sld, elb, wrt]**

}

## Groot : The Two who are One, The One that is All

### The skeleton of the Hardware implementation

1. RaspberryPi (1)
2. Camera (1)
3. The manipulator
   1. Servos (4 for the arm, 2 for the head)
   2. Links (6)
4. Metal or Foam/Mount-Board for the base, joints, links etc
5. 2 markers (1 for the human and 1 for the arm ; preferably broad-tip, blue and red so that the vision part becomes simpler)
6. The board with the grid - white board, black tape, stamp : X,O

### The skeleton of the Software implementation

#### Generic

def **start\_here**()

{

curr\_state = **capture\_picture**()

// the human opponent plays first

while(True)

{

delay(100 ms)

if ( **is\_endgame**(curr\_state) )

{

break

}

[board\_visible, nxt\_state] = **capture\_picture**()

if (board\_visible && (!**isequal\_states**(cur\_state, nxt\_state))

{

curr\_state = **groot’s\_turn**(cur\_state, nxt\_state)

}

}

}

def **groot’s\_turn**(cur\_state, nxt\_state)

{

diff\_loc\_box = **diff\_img**(cur\_state, nxt\_state)

det\_pic = **capture\_detailed\_picture**()

box\_no = **take\_decision**(diff\_loc\_box, det\_pic)

**make\_move**(box\_no)

[board\_visible, curr\_state] = **capture\_picture**()

// ‘board\_visible’ will always be one in this case

return curr\_state

}

def **capture\_detailed\_picture**()

{

**bend\_head\_forward**()

[board\_visible, det\_pic] = **capture\_picture**()

// ‘board\_visible’ will always be one in this case

**take\_head\_back**()

return det\_pic

}

def **take\_decision**(diff\_loc\_box, det\_pic)

{

[lx, ly, gx, gy, glb\_box] = **conv\_conv**(loc\_box)

l = **list\_empt\_boxes**(glb\_box, det\_pic)

i = **rand\_int**(0,len(l))

res\_loc\_box = l[i]

return res\_loc\_box

}

def **conv\_conv**(loc\_box)

{

// convention-converter

loc\_box\_x = loc\_box/9; // truncated division

loc\_box\_y = loc\_box%9;

glb\_box\_x = loc\_box\_x/3; // truncated division

glb\_box\_y = loc\_box\_x%3;

glb\_box = glb\_box\_y\*3 + glb\_box\_x;

return [loc\_box\_x, loc\_box\_y, glb\_box\_x, glb\_box\_y, glb\_box]

}

def **rand\_int**(start, exc\_end)

{

number = ...

return number

}

def **is\_endgame**(curr\_state)

{

}

def **list\_empt\_boxes**(glb\_box, det\_pic)

{

returns list of empty local boxes in the global box

}

#### Vision-based

def **diff\_img**(cur\_state, nxt\_state)

{

}

def **isequal\_states**(cur\_state, nxt\_state)

{

}

#### Hardware-based

def **make\_move**(box\_no)

{

}

def **bend\_head\_forward**()

{

}

def **take\_head\_back**()

{

}

def **capture\_picture**()

{

}

# Further... (if time permits)

[Monte Carlo tree-search algorithm](https://en.wikipedia.org/wiki/Monte_Carlo_tree_search), to predict the best move and makes Groot play to win.

# References

1. <https://en.wikipedia.org/wiki/Ultimate_tic-tac-toe>
2. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.461.1097&rep=rep1&type=pdf>
3. <https://deepblue.lib.umich.edu/bitstream/handle/2027.42/6192/bac6709.0001.001.pdf>
4. <http://www.lynxmotion.com/images/html/proj058.htm>

# Outcome