TicTacToe Game Report

# Strategy

### Introduction to TicTacToe

TicTacToe is a traditionally paper and pencil game for two players, named “X” and “O”. Each player takes turns to make a mark on a 3x3 grid which serves as a board. If a straight line of their markings can be found, they win the game. This means much of the game revolves around blocking the opponent’s ability to obtain a win condition while making optimal moves to increasing the player’s chance of victory.

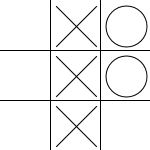
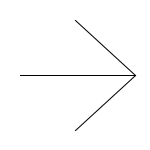
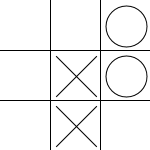
A perfect game can be played to an automatic draw in 100% of cases, provided both players make the optimal moves. There are three possible opening moves available to the first player “X”, which are centre, edge and corner. The second player “O”, essentially plays in reaction to the first player. The first player has the advantage in being able to always force a draw, with a chance of winning. The second player can only force a draw unless the first player has made a mistake.

Much of the game’s appeal is to teach young children the value of good sportsmanship since the simplicity of the game quickly leads said children to conclude that drawing is inevitable provided both parties play well.

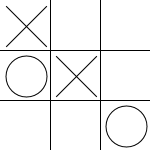
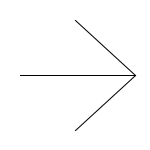
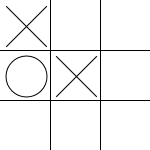
### Optimum Strategy

Win conditions have been documented at length, with a journal article in Cognitive Science (Crowley and Siegler, 1993) documenting the following process for determining the optimum strategy required for a win:

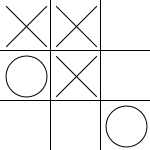
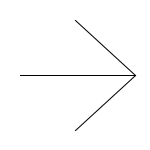
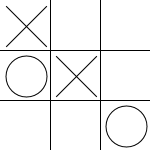
1. Play to win: If a player has two pieces in a row, placing a third to complete the row will win the game for the player



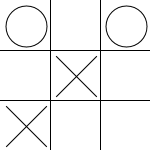
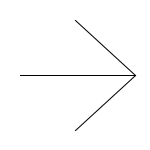
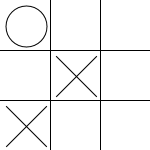
1. Blocking: If the opponent has two pieces in a row, placing a third to block the completion of the row will ensure the continuation of the game and possibly a draw



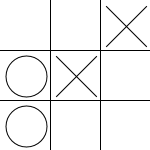
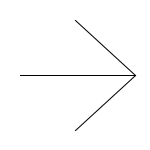
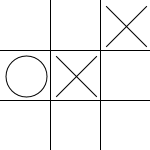
1. Forking: A piece will be placed to create the opportunity of two ways to win. This makes two rows of two pieces, guaranteeing a win in the next turn because the opponent can only block one row



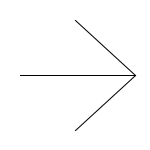
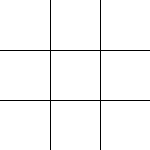
1. Blocking a possible fork:
   1. The player will place a piece to prevent the opponent by winning via a fork by creating a line of two pieces, forcing the opponent to use a turn to block
   2. If the opponent can make a fork, a piece must be played to block the possible fork



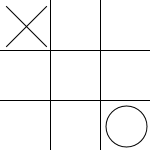
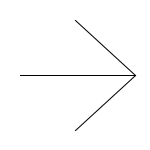
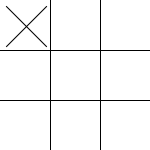
Or:



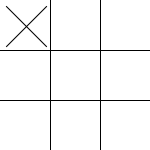
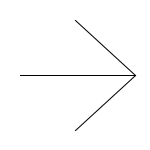
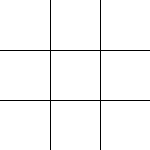
1. Centre: If the centre is empty, the centre of the board should be marked



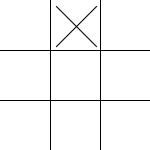
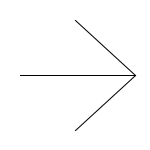
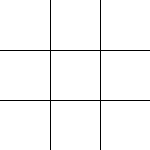
1. Opposite corner: If the opponent plays in a corner, the player plays in the opposite corner



1. Empty corner: The player places a piece in a corner



1. Empty side: The player places a piece into a side



### Initial Markings

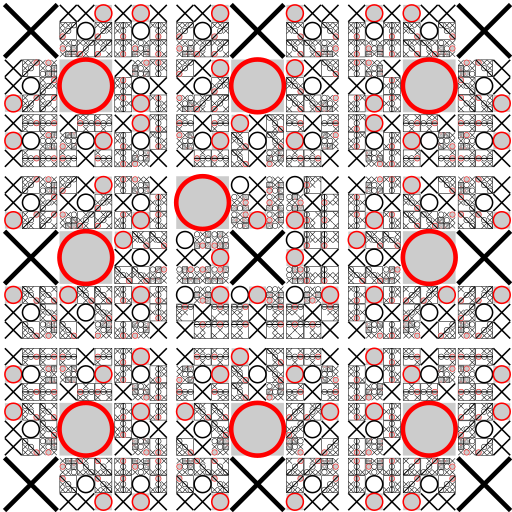
Due to rotation of the board, there are in effect three locations to place an initial “X”. These are the centre, a side or a corner. The first player can win or force a draw from any one of these positions, however a corner is the optimal strategy against an imperfect player because it gives the opponent the smallest number of squares available to force a draw.

The second player, placing an “O”, is in a reactive position in which they must place a piece to avoid a win by the first player. A corner opening should always be responded to with a centre marking and to a centre opening with a corner marking. An edge opening with a centre mark, a corner mark directly next to the X or an edge mark that is opposite the X. All other positions result in the ability of the “X” player to obtain a win.

### Further Play

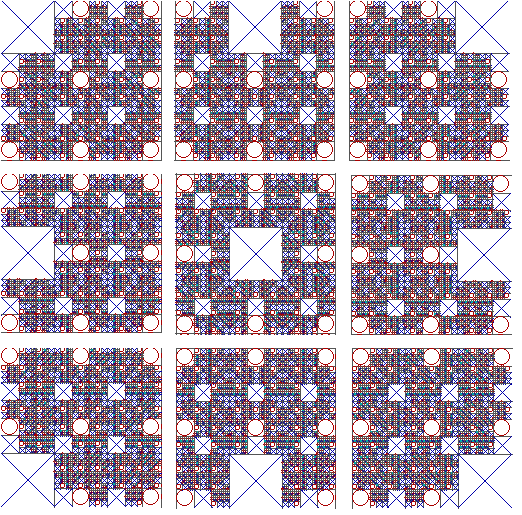
Numerous simulations of all possible moves have been performed, a notable example can be shown in the Scientific American article “A fractal guide to Tic-Tac-Toe” (Stewart, 2000). This article details fractal patterns such as Sherpinski’s gasket and the Sheffer Stroke, the patterns that emerge from such fractals leading to the creation of decision trees giving visualisations of extreme intricacy.

The following image example licensed under the Creative Commons license (nneonneo, 2017) shows an example of the possible moves that can be made:



The number of possible moves can be calculated by running through the possible number of markings a player can make in each turn of the game. The first player has 9 moves, the second player 8, the first then has 7 and so on until the board is filled with markings. This equates to 9! or just under 400,000 possible moves. However, in practice there are 255, 168 valid unique games possible. Of these 131, 184 games are win states for the “X” player, 77, 904 by the “O” player and 46, 080 draw states.

There is much research into the fractal nature of decision trees in TicTacToe, another image (Denis and Grim, 1997) shows an alternate fractal based on the game:



Keeping the number of combinations in mind, a solution should take the form of a set of conventional rules and not a decision tree that factors in every single combination. The rules I’ve outlined in an earlier section of this report are sufficient for any AI strategy with win conditions calculated under the system. Any implementation that would seek to calculate every possible position would be bloated and unnecessary.

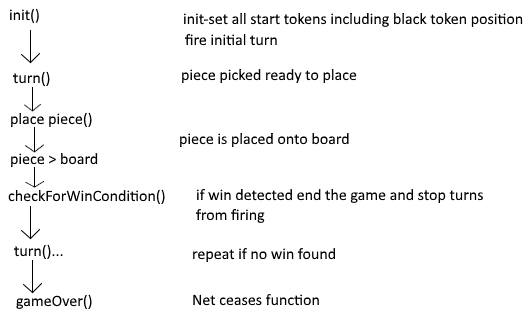
Once again, I must restate that the strategy outlined is optimum mathematically and proven to be correct. I don’t know how to pad this section out more outside of outlining the optimum moves available to a player at a given time.

# Petri Net

### Design

Before I began proper work on the petri net to be developed, I decided to use pen and paper to construct pseudo-code and a basic layout of what the net or set of nets would achieve.

My initial design plan was as follows:



I also sketched out a rough diagram of a basic petri net, which has been somewhat replicated in my final design. During this rough process, I focussed on making a turn loop that would limit and constrain the game to fit the number of turns that could be developed. The reasoning behind this was that there will never be more than 9 turns in the game and for each turn, a piece would need to be created and placed upon the board.

In any case, the initialisation function was to serve as a template for creating a first step that would create all necessary global variables and sub nets for the system with all variables within those initialised to a basic state. This would not include the black tokens and 0 values used within the system that could be placed within Renew.

Turn was to be a process in which the appropriate piece is picked and readied by the system for placement onto the board.

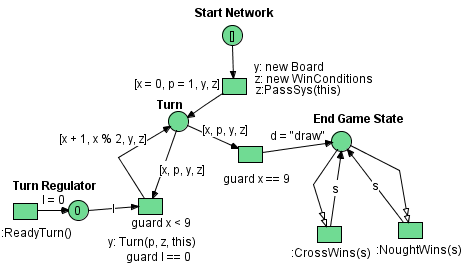
Place piece was to essentially be the process in which the user interacts with the system and places the piece on the board.

With the piece placed, a win condition would be checked and if found would present a message to the user informing them of which player had won the game.

Then turns would repeat until the win condition is found or it would transition to a game over state ending in a draw condition.

With the basic pseudo-code design process figured out, I decided to start work on one of several nets that would form the basis of the TicTacToe system, starting with the system net.

### System Net



The System net is the main net which initiates the system, generates turns and presents the end game state to the user. The initial black token that begins the system is placed at the start network place at the top.

A transition fires that creates a new board for the game and the win conditions section. A system value is passed to the win conditions for win conditions to fire back the victor if a win condition is found.

Moving onto the Turn place, initialised values are entered. “X” is the turn number, starting at 0 and ending in 9. 9 indicating that 9 turns have passed. “P” is the player in question, with crosses being 1 and noughts being appropriately 0. “Y” and “Z” are passed values required for system operation in a later step.

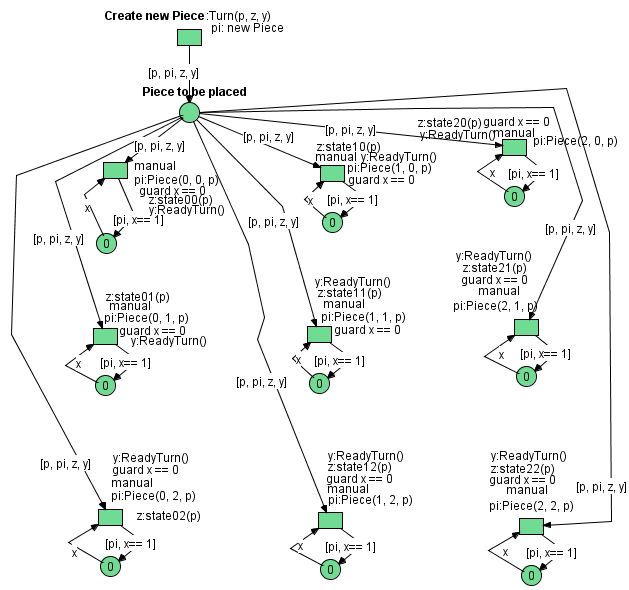
The turn loop first passes all necessary data to the turn transition. The loop is regulated by the turn number “X” needing to be below 9 to fire. The board “y” receives the player number and the system net and win conditions net references so that they may later interact with the other nets. “l” is used as a regulator to ensure that the turn loop can only fire if the turn is ready. Readiness is defined as there is no piece to be placed on the board. In other words, if there is a piece that has not been placed on the board, the turn loop will not fire.

The second part of the turn loop ticks the turn number “X” up by 1, it then calculates the player number based on a division calculation that works out the remainder of division by 2. In effect it uses an odd/even number checking calculation to do this. “y” and “z” as references are cycled through the system without effect so that they may be passed again in future to the turn to begin.

On the right side of the diagram, if the turn number hits 9, the transition displaying a draw may fire. This will provide the user with a draw condition for the end game state if there is no victory calculated. Two transitions each pass a string to the end game state displaying a victory condition achieved if it is calculated in the win conditions net. Clear arcs are used to eliminate an erroneous draw condition, should the simulation fire the draw condition before the final victory condition steps are set.

Overall, I think this net works very well for the job with this being in my opinion the finest bit of work on this coursework, a criticism I have is that I could implement a more stringent method of adding the draw text as if the player spams board pieces without running through the steps before each piece, 2 conditions may appear.

### Board Net



The board net creates and holds all the pieces in the system. It also feeds the win condition network with the information it needs and sends turn readiness information back to the system net.

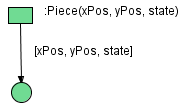
The initial entry point to the board net at the top begins by creating a new piece and passing information forwards to a piece to be placed area that holds piece information in situ while a player decides where to place a piece. “p” is the player number with “z” and “y” as net references. “pi” is the reference to a piece net that is created and passed forward.

The piece to be placed area is connected to 9 separate manual transitions that control the placement of pieces on the board. Because of the turn regulator in the system net, no more pieces may be created until a transition is fired. The connections to the transitions carry over the necessary information as previously discussed.

When a transition is fired by the user, a Piece(xPos, yPos, state) function is fired in the piece network. This passes the co-ordinates on the board and the player number (state) to the piece. A relevant function is passed the player number, which is sent to the win conditions network. A guard in place using the variable “x”, prevents the transition being fired again and the piece is stored in the place below the transition. Finally, ReadyTurn () is fired on the system net which creates a “0” value token which allows new turns to be created and another piece to be generated.

Some criticism I have of the board network is that the actual visual design of it is very cluttered. Functionally I think it works very well but it is not pleasing to the eye and the mess of calls and assigning of variables is a little confusing.

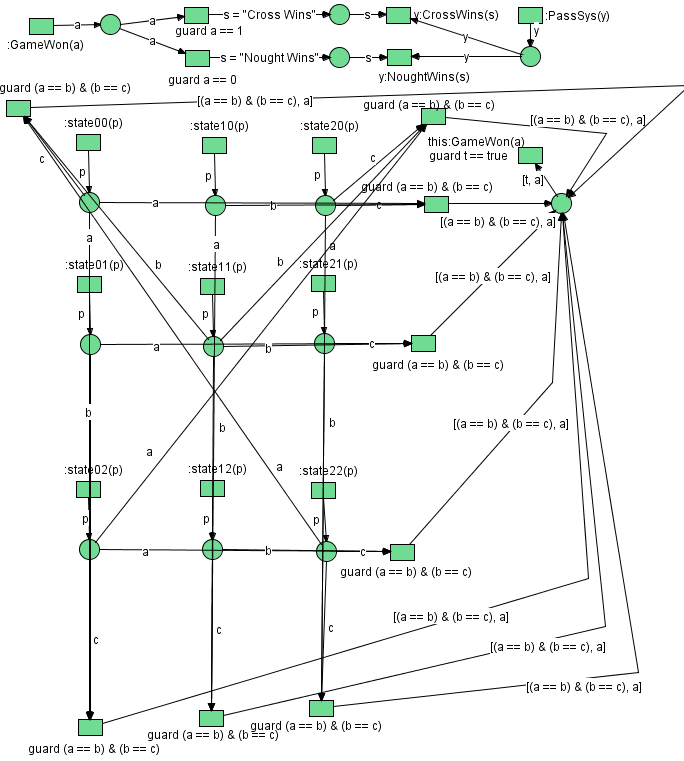
### Piece Net



The piece net is very simple and contains only the coordinates of a piece and the state or player number of the piece. Since the player number is passed to the win conditions network, no information needs to be drawn out of this network. I feel this was a very sensible decision to have made, my initial design of the piece net involved functions to pull data out, but it seemed inappropriate when I could send such data at a higher level.

I think the piece net is poorly implemented, however this is due to the necessity of the coursework demands placed upon me. Due to the win conditions network I have developed, the piece network is mostly redundant. Player markings are contained within the win conditions. Pieces are still recorded and the details of their position and player marking are contained within its network, yet they are never accessed, only assigned once. All in all, I have effectively designed out the piece net while retaining all the functionality of a proper system.

### Win Conditions Net



The win conditions net contains all the information necessary to calculate and process victory conditions. In some ways, due to the simple store of what player has placed a piece into a place on the board, it can be used as a visual representation of the board which is otherwise cluttered by a large amount of function calls and the containing of piece objects.

Each of the 9 statexy(playerNumber) transitions passes the player number to the relevant place in the centre of the network. Each place is connected along with two other places on the same axis to a transition at the end of a row of 3 places, which evaluates the status of said places. The transition will not fire unless it finds that each of the places has an equal value to the others. If the transition fires due to the 3 places in a row having identical values, a “true” value is found and passed with the player number to a holding place.

The next transition which fires the GameWon(playerNumber) is guarded by a true state to ensure it does not fire unless a successful match of 3 places is found. The player number is then passed to the top of the network to be prepared for the end game state message.

Depending on the player number passed, a transition will fire that then passes a string with the relevant victory condition. This string is then sent into a CrossWins or NoughtWins function. This function requires the system net reference which was passed to the win conditions network during initialisation. Once the message is passed back to the system net, the win condition is presented to the user in the end game state place.

The win conditions network is in my opinion one of the best parts of this work, considering the amount of connections required, it does look good in a visual sense. I also think the implementation of the victory conditions was conducted in a good and concise way. However, the top of the network could be improved by removing the places before the victory message call to the system network, this is somewhat redundant but at the least it keeps it clear.

# Testing

To evaluate the system, I will conduct a test of all major functions within the system. This will focus on testing each transition and place in turn systematically and correcting any defects if found.

## System net testing:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Area** | **Purpose of Test** | **Expected Value** | **Actual Value** | **Corrective Action Needed?** |
| Start Place | Check that black token generates and works properly | Black token present | Black token present | No |
| Turn Place | Check that x is created and set to 0 | 0 | 0 | No |
| Turn Place | Check that p is created and set to 1 | 1 | 1 | No |
| Turn Place | Check that y is a newly created board | Y is a board network | Y is a board network | No |
| Turn Place | Check that z is a newly created win conditions | Z is a win conditions network | Z is a win conditions network | No |
| Start transition | Check the created win conditions is passed the system net reference | The win conditions network receives a system network reference | The win conditions network receives a system network reference | No |
| Turn Regulator | Check to see if a new “l” variable is created by the turn regulator | “l” = 0 | “l” = 0 | No |
| Turn Place | Check if x is incremented for a new turn | x = 0 and after one cycle is 1 | x = 0 and after one cycle is 1 | No |
| Turn Place | Check to see if the player number is updated appropriately on a new turn | X = 1 and then switches to 0 and back to 1 after two turns | X = 1 and then switches to 0 and back to 1 after two turns | No |
| End Game State | Ensure a draw is set after 9 turns and no win is found | “Draw” is present | “Draw” is present | No |
| End Game State | Ensure Cross Wins can fire on a win for crosses | “Cross Wins” is present | “Cross Wins” is present | No |
| End Game State | Ensure Nought Wins can fire on a win for noughts | “Nought Wins” is present | “Nought Wins” is present | No |

### Board Net Testing:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Area** | **Purpose of Test** | **Expected Value** | **Actual Value** | **Corrective Action Needed?** |
| Piece to be Placed, Place | Check to see if the player number is passed correctly | P = 1 | P = 1 | No |
| Piece to be Placed, Place | Check to see if the piece is created and passed correctly | Created Piece | Created Piece | No |
| Piece to be Placed, Place | Check to see if the win conditions reference is passed correctly | Win conditions reference present | Win conditions reference present | No |
| Piece to be Placed, Place | Check to see if the system net reference is passed correctly | System net reference present | System net reference present | No |
| Place 0, 0 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 0, 0 | Check to see if x value is updated on trigger | X = 1 | X = false | Yes |
| Place 0, 0 | Check to see if x value is updated on trigger | X = 1 | No enabled bindings found | Yes |
| Place 0, 0 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |
| Place 1, 0 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 1, 0 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |
| Place 2, 0 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 2, 0 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |
| Place 0, 1 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 0, 1 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |
| Place 1, 1 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 1, 1 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |
| Place 2, 1 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 2, 1 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |
| Place 0, 2 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 0, 2 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |
| Place 1, 2 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 1, 2 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |
| Place 2, 2 | Check to see if piece is created on placement | Piece created and present in place | Piece created and present in place | No |
| Place 2, 2 | Check to see if x value is updated on trigger | X = 1 | X = 1 | No |

### Piece Net Testing:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Area** | **Purpose of Test** | **Expected Value** | **Actual Value** | **Corrective Action Needed?** |
| Piece Place (0, 0) | Check to see if x, y and state values are correctly set | X = 0, y = 0, p = 1 | X = 0, y = 0, p = 1 | No |
| Piece Place (1, 0) | Check to see if x, y and state values are correctly set | X = 1, y = 0, p = 1 | X = 1, y = 0, p = 1 | No |
| Piece Place (2, 0) | Check to see if x, y and state values are correctly set | X = 2, y = 0, p = 1 | X = 2, y = 0, p = 1 | No |
| Piece Place (0, 1) | Check to see if x, y and state values are correctly set | X = 0, y = 1, p = 1 | X = 0, y = 1, p = 1 | No |
| Piece Place (1, 1) | Check to see if x, y and state values are correctly set | X = 1, y = 1, p = 1 | X = 1, y = 1, p = 1 | No |
| Piece Place (2, 1) | Check to see if x, y and state values are correctly set | X = 2, y = 1, p = 1 | X = 2, y = 1, p = 1 | No |
| Piece Place (0, 2) | Check to see if x, y and state values are correctly set | X = 0, y = 2, p = 1 | X = 0, y = 2, p = 1 | No |
| Piece Place (1, 2) | Check to see if x, y and state values are correctly set | X = 1, y = 2, p = 1 | X = 1, y = 2, p = 1 | No |
| Piece Place (2, 2) | Check to see if x, y and state values are correctly set | X = 2, y = 2, p = 1 | X = 2, y = 2, p = 1 | No |

### Win Conditions Testing:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Test Area** | **Purpose of Test** | **Expected Value** | **Actual Value** | **Corrective Action Needed?** |
| State (0, 0) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| State (1, 0) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| State (2, 0) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| State (0, 1) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| State (1, 1) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| State (2, 1) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| State (0, 2) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| State (1, 2) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| State (2, 2) | Check to see if the player value is passed correctly | P = 1 | P = 1 | No |
| Top Horizontal Victory Condition | Check if condition properly fires | T = true, a = 1 | T = true, a = 1 | No |
| Middle Horizontal Victory Condition | Check if condition properly fires | T = true, a = 1 | T = true, a = 1 | No |
| Bottom Horizontal Victory Condition | Check if condition properly fires | T = true, a = 1 | T = true, a = 1 | No |
| Left Vertical Victory Condition | Check if condition properly fires | T = true, a = 1 | T = true, a = 1 | No |
| Middle Vertical Victory Condition | Check if condition properly fires | T = true, a = 1 | T = true, a = 1 | No |
| Right Vertical Victory Condition | Check if condition properly fires | T = true, a = 1 | T = true, a = 1 | No |
| Top Left to Bottom Right Diagonal Victory Condition | Check if condition properly fires | T = true, a = 1 | T = true, a = 1 | No |
| Top Right to Bottom Left Diagonal Victory Condition | Check if condition properly fires | T = true, a = 1 | T = true, a = 1 | No |
| GameWon Place | Check if a is passed correctly | A = 1 | A = 1 | No |
| CrossWins Place | Check if CrossWins is passed the correct string | S = “Cross Wins” | S = “Cross Wins” | No |
| NoughtWins Place | Check if NoughtWins is passed the correct string | S = “Nought Wins” | S = “Nought Wins” | No |

# AI Player

For an AI Player, the strategy detailed in the beginning of this report is the strategy I would use in implementation. However, I have not implemented an AI player due to difficulties with the Java code integration. While I have not made a presentable attempt, I do know exactly how I would implement the AI player.

To put it simply, there are two connective points I would use with the network to a java program:

1. The manual transitions available to the user, made available to a computer program to activate said transitions
2. The values contained on the win conditions network, showing the layout of the pieces placed by each player used to calculate the position to play to

Said AI using the strategy outlined in this document would in effect play to an optimum position. Apologies for the lack of AI, it’s hard to get the Java code working.

Without Java code, implementation is possible but far more difficult, I would estimate about 3-4 times the complexity of the current network setup would need to be used. I would use a similar method to detecting win conditions to analyse the current disposition of markings on the board. This would give a mathematical result to a set of switches which would be linked to the implementation of strategy, access to which would be ranked according to the strategy earlier in the report. Put simply, if a winning move was available that would fire but if not, the decision making would effectively bypass the win move and instead look to block and so on and so forth.

# References

Crowley, K. and Siegler, R. (1993). Flexible Strategy Use in Young Children's Tic-Tac-Toe. *Cognitive Science*, 17(4), pp.531-561.

Stewart, I. (2000). A Fractal Guide to Tic-Tac-Toe. *Scientific American*, 283(2), pp.86-88.

nneonneo (2017). *Optimal decision tree for player O in Tic-Tac-Toe. Based on "A Fractal Guide to Tic Tac Toe" by By Ian Stewart on August 1, 2000 Scientific American*. [image] Available at: https://upload.wikimedia.org/wikipedia/commons/2/2f/Tictactoe-O.svg [Accessed 17 Nov. 2017].

Denis, P. and Grim, P. (1997). Fractal Images of Formal Systems. *Journal of Philosophical Logic*, 26(2), pp.181-222.