Randomised fast no-loss expert system to play tic-tac-toe like a human

eISSN 2517-7567
Received on 19th April 2020
Revised 25th July 2020
Accepted on 28th August 2020
E-First on 9th November 2020
doi: 10.1049/ccs.2020.0018
www.ietdl.org

Aditya Jyoti Paul[1,2] ✉
[1]*Cognitive Applications Research Lab, India*
[2]*Department of Computer Science and Engineering, SRM Institute of Science and Technology, Kattankulathur, Tamil Nadu - 603203, India*
✉ *E-mail: aditya_jyoti@srmuniv.edu.in*

Abstract: This study introduces a blazingly fast, no-loss expert system for tic-tac-toe using decision trees called T3DT, which tries to emulate human gameplay as closely as possible. It does not make use of any brute force, minimax, or evolutionary techniques, but is still always unbeatable. To make the gameplay more human-like, randomisation is prioritised and T3DT randomly chooses one of the multiple optimal moves at each step. Since it does not need to analyse the complete game tree at any point, T3DT is exceptionally faster than any brute force or minimax algorithm, this has been shown theoretically as well as empirically from clock-time analyses in this study. T3DT also does not need the data sets or the time to train an evolutionary model, making it a practical no-loss approach to play tic-tac-toe.

1 Introduction

Tic-tac-toe is one of the oldest and most played childhood games, dating back as early as 1300BC in Egypt. The board comprises a 3 × 3 matrix; two players alternatively place an X or an O in an empty cell and the first player to get three X's or O's in a row, column, or diagonal wins. It is a zero-sum game in which one player's win is the other's loss. It also is a full information game that is both the players have complete information about all the moves that are possible arising from a certain board state. Over the ages, people have come up with tricks or techniques to play it without being beaten. The operations in decreasing priority order are as follows:

(a) making a winning move, else
(b) blocking the opponent's winning move, else
(c) trying to make a fork, and lastly
(d) blocking an opponent's fork.

The main objective of this research work is the creation of a no-loss expert system that can play tic-tac-toe by understanding the rules of the game and maintain a high win-to-draw ratio. Here no-loss refers to the game never losing against an opponent, it would draw or preferably win the game, irrespective of whether it starts first or second. It should be able to execute the operations listed above just like a human would, so a decision-tree based algorithm (T3DT) was chosen. Humans' perception of games is fuzzy and chaotic in nature and the randomness hence produced makes them fun to be played with. T3DT tries to recreate the randomness to the maximum extent possible, thus making the algorithm a fun opponent to play with.

2 Literature review

Many heuristics and mathematical algorithms have been designed to win the game. In the field of two-player zero-sum games, pioneering work was done by John von Neumann, his first proof [1] of the minimax (MM) algorithm published in 1928 created ripples through the scientific and mathematical communities and is still considered one of the founding blocks of game theory. Kjeldsen [2] iterated the history of the development of Neumann's MM from his paper [1] in 1928 to his completely different proof in his book [3] with Morgenstern in 1944. In [2], Kjeldsen wrote that by considering mixed strategies and expressing player values in the bilinear form h, Neumann [1] had shown that for two-player zero-sum games, there always exists optimal mixed strategies ξ_0, η_0, such that

$$\max_{\xi}\ \min_{\eta} h(\xi,\ \eta) = \ \min_{\eta}\ \max_{\xi} h(\xi,\ \eta) = h(\xi,\ \eta) \qquad (1)$$

Kjeldsen [2] wrote that Neuman had actually proven a generalised version of MM considering a class of functions broader than the bilinear form h. However, for simpler two-player zero-sum games such as tic-tac-toe, the essence of this equation lies in the observation that the optimal strategy for both players involves pessimistically minimising the maximum damage that can be inflicted by the opponent.

MM surprised people by providing an algorithm which can never be beaten, and most of the earlier approaches for tic-tac-toe relied on some form of game tree search. Claude Shannon realised the importance of variable depth search in 1949 [4], and best-first approaches with both fixed [5] and variable depth [6] have been implemented since. In general, much of the early research on game-tree search algorithms such as alpha-beta pruning [7–9], Scout [10], NegaScout [11], SSS* [12–15], fixed and dynamic node ordering [16] and aspiration windows [17, 18] make the same choice as full-width fixed depth MM. On the other hand, algorithms such as B* [19, 20], min–max approximation [21], conspiracy search [22, 23], meta-greedy search [24], singular extensions [25], and risk assessment [26] might not always do so, searching some strategies more deeply than others. Improved hardware and algorithms supporting parallelism [27–30] have made tree generation and search even faster. Here, just a cursory description of game tree search methods has been given as a detailed account is outside the scope of this paper, but [31–33] compare and contrast these methods in much greater detail for a clearer understanding.

Ignoring the symmetry of the board, there are 255,168 unique games of tic-tac-toe [34], and move-generation and analysis of all these moves at runtime are tedious and time-consuming. Humans are skilled at simplifying the search process by pruning the options at hand by having an intrinsic understanding of the game and selecting a few main candidates [35]. This task is challenging for computers and thus other approaches that do not rely on game tree search were developed and have been described below.

Fogel [36] laid the groundwork for using evolution in developing neural agents to play tic-tac-toe. Chellapilla and Fogel [37] demonstrated how intelligence is developed in evolutionary

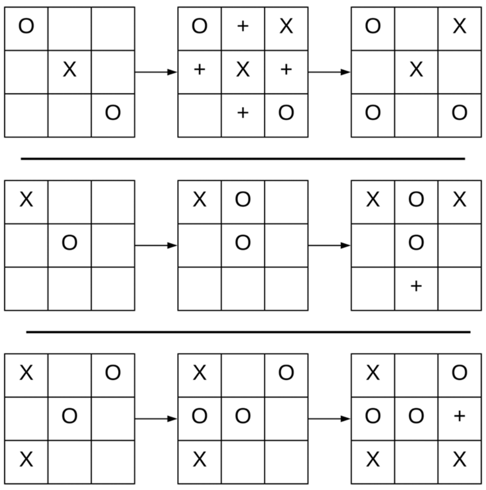


Fig. 1 *Three examples in which Soedarmadji's solution [39], playing with X, loses the game [42]*

neural networks, and how they can learn to play any arbitrary game without depending upon human expertise or being explicitly programmed to do so. Hochmuth [38] described how this game could be played by a genetic algorithm. Soedarmadji [39] suggested a decentralised approach comprising nine agents, one for each cell in the board and one manager to choose which move to do make based on the priority number generated by each independent agent. Yau *et al.* [40] compared the implementations of various adaptations and related parameters in evolving agents to play games such as tic-tac-toe. Yau *et al.* [41] comprehensively discussed the use of co-evolution and Pareto evolution in the development of neural agents and how success is strongly dependent on the initialisation of the co-evolution process, and related empirical data is discussed.

Bhatt *et al.* [42] did some commendable work, trying to find out no-loss strategies for the game using a customised genetic algorithm, focusing on a high win-to-draw ratio. They succeeded in finding 72,657 such strategies, and also came to some other interesting conclusions, which verify our existing notions about the game. Mohammadi *et al.* [43] presented a co-evolution and interactive fitness-based genetic algorithm to build a human-competitive player. Ling *et al.* [44] used a double transfer function and Rajani *et al.* [45] applied the Hamming distance classifier, on neural networks to demonstrate the advantages of each method. Singh *et al.* [46] devised a deterministic mathematical model to solve tic-tac-toe as a nine-dimensional problem. Karamchandani *et al.* [47] discussed the techniques used in [44, 45] and presented a very basic rule-based algorithm, which always targets the centre at the start and has a set of special moves, but it can still be beaten by an opponent in some cases. Garg *et al.* [48, 49] formally defined the tic-tac-toe problem and formulated a winning strategy for the same using a multi-tape turing machine and automata theory.

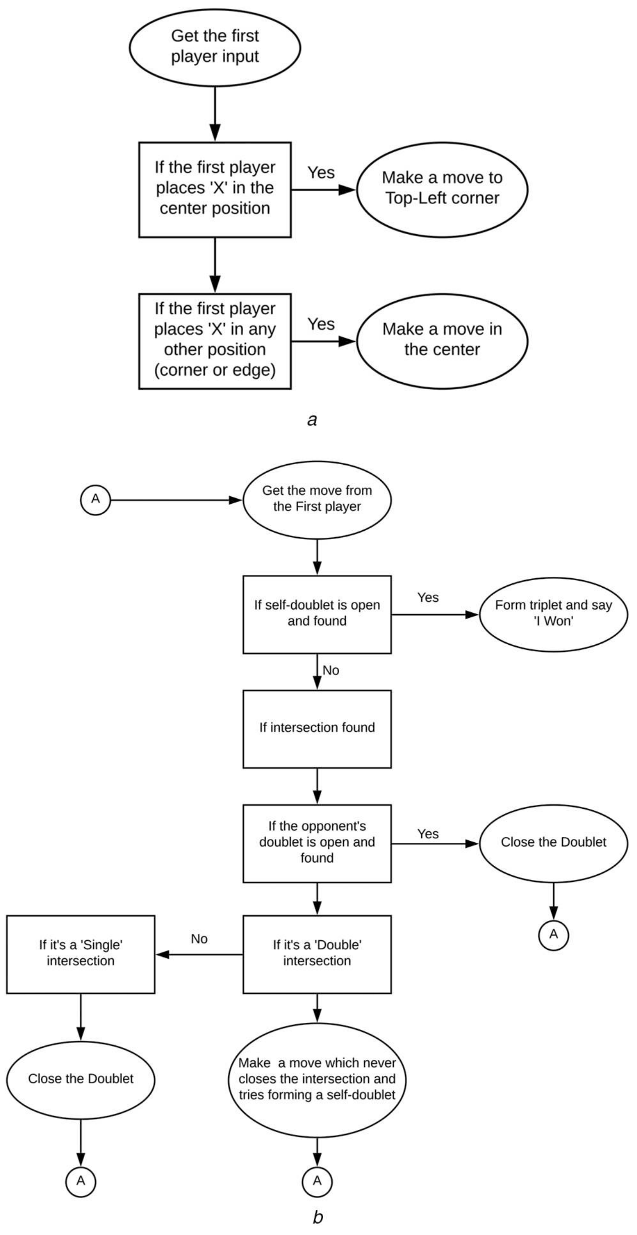
Initial approaches by Hochmuth [38] and Soedarmadji [39] were innovative but not very effective. Hochmuth's method does not guarantee any loss for all possible tic-tac-toe games and was an exercise demonstrating the effectiveness of genetic algorithms, rather than the development of a perfect no-loss algorithm. Soedarmadji's heuristic approach loses in at least three board states as described in [42]. The three examples are explained through Fig. 1, in which, optimal positions for X are marked with ‘+’. In the first example, X should have marked any of the edge-centres, not in the top-right corner which opens a fork for the opponent, the second example is more evident as X fails to block a two O's in a row and similarly in the third example X again fails to block O.

It is discussed in [40, 42] and in Section 4 of this paper itself that the optimal strategy for the first and second player are different from each other. The first player has a greater chance of victory (around 1.68:1 [42]). In [50], a non-randomised decision tree approach is discussed by Sriram *et al.* They demonstrated some interesting observations, such as how MM fails to move optimally in as many as 25–40 cases. However, the algorithm was unclear in some aspects, such as it was not discussed how the algorithm would make a move if it had to start the game itself, even though the strategies are different for the two players as discussed above. Also, when the opponent makes the first move, it was suggested to always mark the centre if possible else mark the top-left corner, according to the tree in Fig. 2*a*, ignoring the edges completely. The remaining moves were to follow the tree in Fig. 2*b*. Furthermore, it was unclear exactly how each of the steps in the decision tree was carried out as that would have a significant effect on the actual runtime of the algorithm. Runtime comparison against vanilla MM was done solely on the basis of asymptotic complexity and not clock time complexity. Asymptotic complexity does not give a true representation of the time taken by an algorithm to run, which is needed for benchmarking. All these shortcomings are addressed in this research work.

Thus, it is observed that initially most of the game-playing algorithms were inspired by MM and speed-ups involving pruning the game tree by various methods. Gradually approaches based on genetic and evolutionary algorithms were proposed, and these can generalise well to larger boards, but they take time to train and needs a careful tweaking of the hyperparameters. It is also difficult to estimate how many of these no-loss strategies can actually win against a sub-optimal player, which is a necessary trait to make the algorithm more human-like, for example, the ‘forgetting problem’ described in [41] prevents ‘expert level’ players from winning or even drawing against a medium level or random players.

People have through the ages, tried to intuitively come up with rules to win the game. However, surprisingly not much research has been done to either try to design a randomised rule-based solution until now that tries to maximise win-to-draw ratio for tic-tac-toe and compare such an algorithm's clock time complexity to MM and similar tree search approaches. Although most of the techniques to play tic-tac-toe are centuries old, the authors in [51, 52] reported some rules, which could be incorporated into an expert system such as T3DT. However, this book [51] steers clear from starting in an edge, which was necessary for the randomisation and hence incorporated into T3DT. In [52], Ryan Aycock went to great lengths to explain the motivation for some of the rules for effective gameplay.

Hence, this research work proposes a novel no-loss expert system to play tic-tac-toe, called T3DT, and compares the performance both theoretically as well as from actual runtime to existing MM-based procedures, proving that it is faster and a practical approach to play tic-tac-toe. The rest of the paper is organised as follows. In Section 3, the optimality of some existing MM-based tree-search algorithms, which have been used for benchmarking, are briefly discussed. In Section 4, the proposed algorithm is explained in detail. In Section 5, the experimental methodology is described and some newly devised metrics for the benchmarking process are elucidated. In Section 6, the algorithm is compared to the existing algorithms and the results are compared and analysed. Finally, Section 7 gives the conclusion and discusses some avenues for further work.



Get the first player input
If the first player places 'X' in the center position
Yes
Make a move to Top-Left corner
If the first player places 'X' in any other position (corner or edge)
Yes
Make a move in the center
a
A
Get the move from the First player
If self-doublet is open and found
Yes
Form triplet and say 'I Won'
No
If intersection found
If the opponent's doublet is open and found
Yes
Close the Doublet
A
If it's a 'Single' intersection
No
If it's a 'Double' intersection
Close the Doublet
Make a move which never closes the intersection and tries forming a self-doublet
A
A
b

Fig. 2 *Decision trees for the algorithm suggested in [50]*
(a) Decision tree for the first move, ***(b)*** Decision tree for the remaining moves

3 Optimality of some existing tree-search algorithms

This section gives a brief overview and discusses the optimality of three commonly used MM-based no-loss strategies, namely

(a) MM
(b) MM with alpha–beta pruning (ABP)
(c) MM with advanced alpha–beta (ABA)

Optimality refers to the algorithm arriving at the goal in the least number of moves. Here the vanilla MM and MM with ABP have their usual implementations. MM with ABA has a modified score function, which subtracts the number of moves left till victory from the returned score. The vanilla MM algorithm and ABP never lose but they may occasionally make a move that results in a slower victory. For example, the opponent starts the game and after both players alternatingly making moves O_1, X_1, O_2, X_2, and O_3, the bot's X_3 moves chosen by each of the algorithms are shown in Fig. 3. Here, marking X_3 in cell (3,1) would result in a victory on the diagonal instantly. MM and ABP do not choose this move, they still win eventually, but take a longer path. Including the depth into

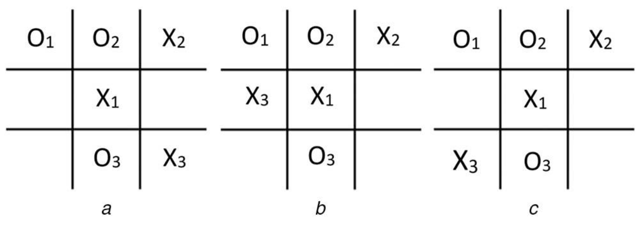


Fig. 3 *Different moves for X_3 made by the algorithms*
(a) MM, ***(b)*** ABP, ***(c)*** ABA on the same board state. T3DT makes the same winning move as ABA in this state

the evaluation function allows ABA to pick the optimal winning move (3,1) such as T3DT leading to the fastest victory.

4 Proposed method

In this section, the no-loss system called T3DT is detailed, which is much faster than MM-based procedures, because it logically partitions the entire game into small subparts that are easier to analyse. Henceforth the computer or artificial intelligence player will be referred to as the 'bot', and it is assumed for simplicity that the bot always plays with X and the opponent plays with O. The algorithm can be divided into two strategies

(a) When the bot starts the game (Sections 4.1 and 4.2) and
(b) when the opponent starts the game (Sections 4.3 and 4.4).

4.1 Algorithm when the bot starts the game

Start

Step 1: the bot makes the first move, with absolute randomness in a corner edge or centre.
Step 2: the opponent marks any of the remaining cells.
Step 3: depending on the opponent's first move in Step 2, the bot makes its second move as shown in Fig. 4, and explained in Section 4.2,
Step 4: then the opponent makes its second move.
Step 5: disregarding the opponent's second move, the bot always tries to first make a winning triplet or block the opponent, if no such possibilities exist, it makes use of the strategies shown in Fig. 4, and explained in Section 4.2.
Step 6: the opponent makes its move.
Step 7: the bot makes its consequent moves, based on the strategies mentioned in Section 4.5, which comprise winning or blocking or making a random move.
Step 8: repeat steps 6 and 7 until the game ends.
Step 9: declare the result.

End

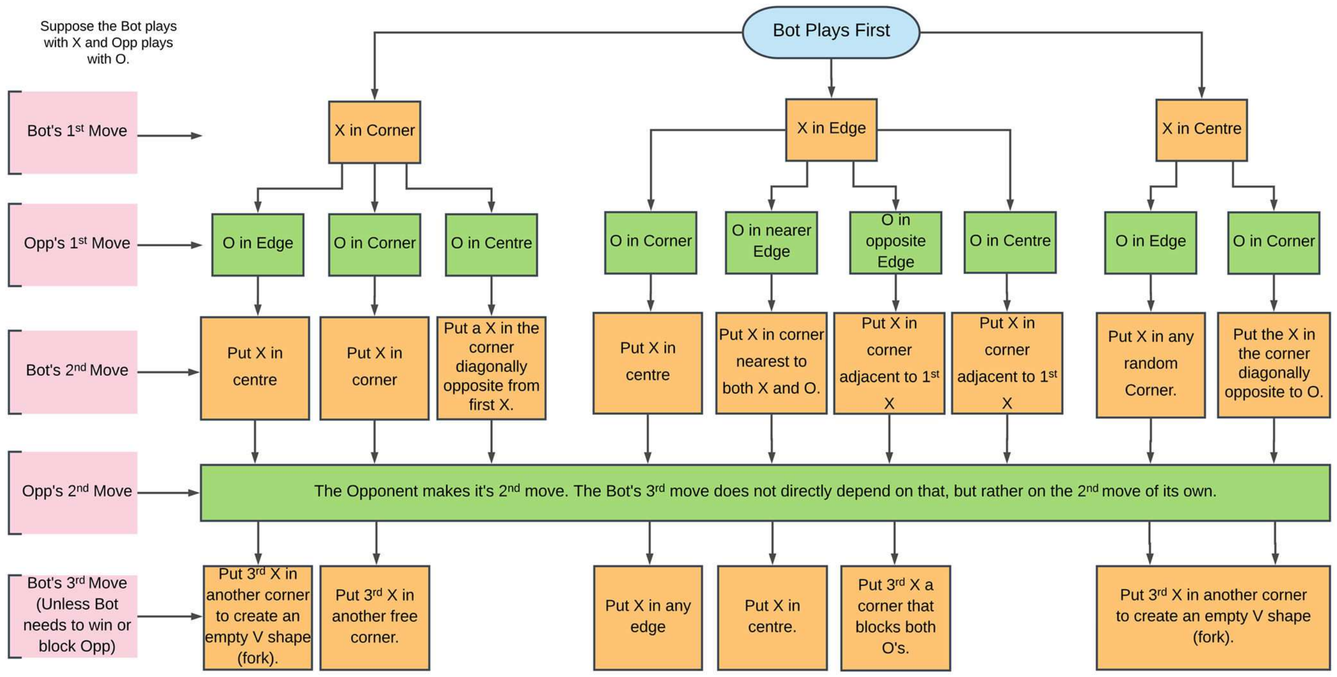
4.2 First few moves when the bot starts the game

When the bot starts the game, to make the game truly randomised, it chooses a corner, edge, or centre with equal probability. These three cases again have their own sub-parts, which are all described in Fig. 4.

In the following Sections 4.2.1–4.2.3, the motivation for each of the choices made in this decision tree is elaborated.

4.2.1 Bot starts in a corner: The opponent chooses one of the following positions, based on which the bot makes the next few moves:

(a) *Edge*: if the opponent chooses an edge, then the bot chooses the centre as its second move. Now since the bot has made two moves along a diagonal, the opponent is forced to mark the other corner of the diagonal to block the bot from winning. Now unless it needs to win or block the opponent, the bot marks a corner in its third move, such that it creates an empty V-shaped fork, this guarantees a win.



Suppose the Bot plays with X and Opp plays with O.
Bot Plays First
Bot's 1st Move
Opp's 1st Move
Bot's 2nd Move
Opp's 2nd Move
Bot's 3rd Move (Unless Bot needs to win or block Opp)
X in Corner
X in Edge
X in Centre
O in Edge
O in Corner
O in Centre
O in Corner
O in nearer Edge
O in opposite Edge
O in Centre
O in Edge
O in Corner
Put X in centre
Put X in corner
Put a X in the corner diagonally opposite from first X.
Put X in centre
Put X in corner nearest to both X and O.
Put X in corner adjacent to 1st X
Put X in corner adjacent to 1st X
Put X in any random Corner.
Put the X in the corner diagonally opposite to O.
The Opponent makes it's 2nd move. The Bot's 3rd move does not directly depend on that, but rather on the 2nd move of its own.
Put 3rd X in another corner to create an empty V shape (fork).
Put 3rd X in another free corner.
Put X in any edge
Put X in centre.
Put 3rd X a corner that blocks both O's.
Put 3rd X in another corner to create an empty V shape (fork).

Fig. 4 *Decision tree when the bot starts the game*

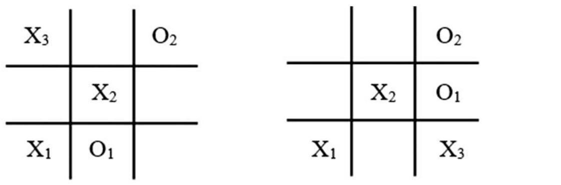


Fig. 5 *Bot starts in the corner and the opponent's first move is in an edge*

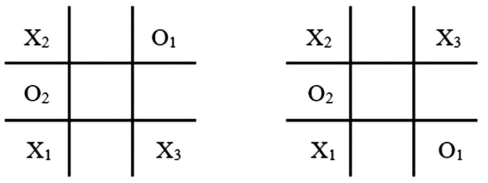


Fig. 6 *Bot starts in the corner and the opponent's first move is in a corner*

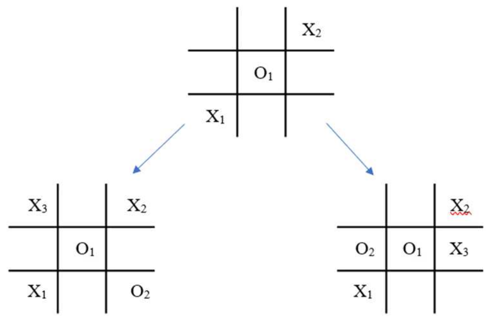


Fig. 7 *Bot starts in the corner and the opponent's first move is in the centre*

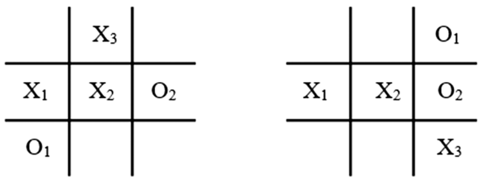


Fig. 8 *Bot starts in edge and the opponent's first move is in a corner*

In both of the games in Fig. 5, irrespective of the position of O_3, the bot always wins.

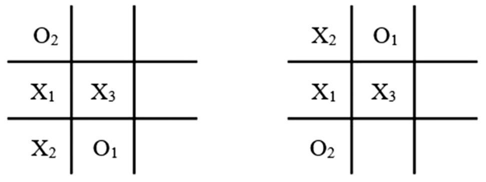


Fig. 9 *Bot starts in edge and the opponent's first move is in the nearer edge*

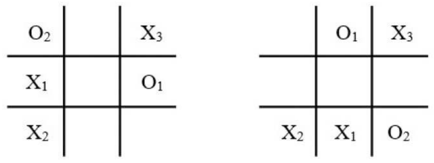


Fig. 10 *Bot starts in edge and the opponent's first move is in the opposite edge*

(b) *Corner*: if the opponent chooses another corner, then the bot marks any of the remaining free corners. The opponent is now forced to block the bot's win and the bot marks the last free corner, thus creating a fork for itself. Once again this guarantees a win for the bot, such as in the games in Fig. 6.

(c) *Centre*: suppose the opponent marks the centre, then the bot marks the diagonally opposite corner to the first X. Now if the opponent places O_2 another corner, the bot's win is guaranteed (see resulting board on left in Fig. 7); else the game will end in a draw (see the resulting board on right in Fig. 7) for further optimal play.

4.2.2 Bot starts in an edge: This is the most complicated and interesting part of the algorithm when the bot chooses an edge. It would be much simpler to just prevent the bot from starting in the edge, as suggested by multiple sources [51, 52], but to make the game randomised, this starting move has to be included. The opponent may choose any of the following cases, based on which the bot's subsequent plays are explored:

(a) *Corner*: if the opponent chooses a corner, the bot moves in the centre and in the third move, it marks any of the remaining empty edges, unless a winning or blocking opportunity arises. This move-sequence almost always results in a draw, such as the two games in Fig. 8.

(b) *Edge*: it is again of two types:

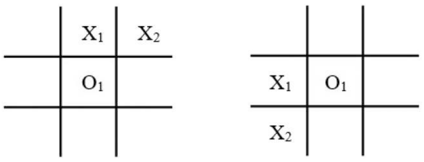


Fig. 11 *Bot starts in edge and the opponent's first move is in centre*

i. *Edge nearer to the bot's first move*: here we will attempt to create an L-shaped fork making sure the bot wins. If the opponent chooses any of the two edges nearest to the bot's chosen edge, the bot puts a X_2 in the square adjacent to both the bot's X_1 and opponent's O_1. Now the opponent will be forced to block the bot and the bot will now make its third move I the centre. This sequence guarantees a win for the bot, because of the L-shaped fork that is formed, as seen in Fig. 9.

ii. *Edge opposite to the bot's first move*: if the opponent chooses the opposite edge to the bot's first move X_1, the bot places a cross in any of the corners adjacent to X_1. The opponent will block the bot with O_2 in the corner. Now the bot places X_3 in the corner that is closest to the opponent's moves O_1 and O_2. This sequence results in a draw, as in Fig. 10.

(c) *Centre*: if the opponent chooses a centre, the bot marks the corner adjacent to its first move. This always ends in a draw for optimal play, as in Fig. 11.

4.2.3 Bot starts in the centre: Then the opponent chooses either of the following positions, based on which the bot makes its subsequent moves:

(a) *Edge*: if the opponent chooses an edge, the bot chooses any of the four corners randomly. Now the opponent needs to block the bot by its move O_2. The bot takes advantage of this and creates a fork which is an empty V-shape, by placing X_3 in the required corner. This sequence guarantees victory for the bot, as in both the games in Fig. 12.

(b) *Corner*: If the opponent chooses a corner, the bot marks the diagonally opposite corner. Now unless the opponent moves in a corner, the bot will always win, by forming an empty V-shaped fork, as in Fig. 13.

4.3 Algorithm when the opponent starts the game

```
Start

Step 1: bot waits for the opponent to make its first move.
Step 2: based on Step 1, the bot makes its first move as shown in Fig. 14, and explained in Section 4.4,
Step 3: then the opponent makes its second move.
Step 4: the bot always tries to first make a winning triplet or block the opponent, if no such possibilities exist, then it makes its move depending on the strategies shown in Fig. 14, and explained in Section 4.4.
Step 5: the opponent makes its move.
Step 6: the bot makes its consequent moves, based on the strategies mentioned in Section 4.5, which comprise winning or blocking or making a random move.
Step 7: repeat steps 6 and 7 until the game ends.
Step 8: declare the result.

End
```

4.4 First few moves when the opponent starts the game

The opponent's first move may be any position on the board – a corner, an edge, or the centre. This leads to three strategies for the bot, which are all described in Fig. 14.

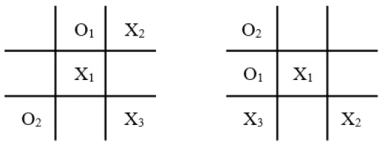


Fig. 12 *Bot starts in the centre and the opponent's first move is in an edge*

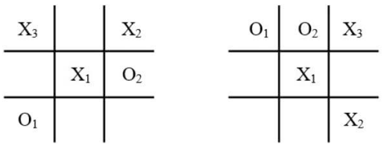


Fig. 13 *Bot starts in the centre and the opponent's first move is in a corner*

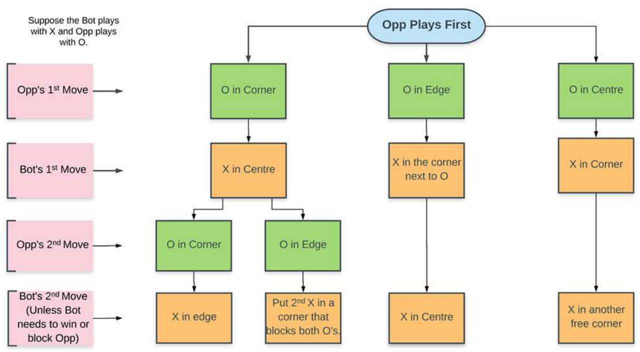


Fig. 14 *Decision tree when the opponent starts the game*

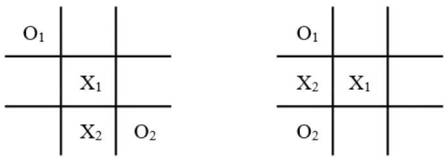


Fig. 15 *Opponent starts in the corner and the opponent's second move is in another corner*

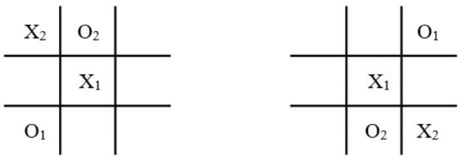


Fig. 16 *Opponent starts in the corner and the opponent's second move is in an edge*

4.4.1 Opponent starts in a corner: If the opponent starts in the corner, the bot moves in centre as its first move. Now the opponent's second move may be either a corner or an edge, based on which the bot makes its subsequent moves.

(a) *Corner*: if opponent's second move is a corner, bot's second X is in any of the four empty edges, unless there is a need to block the opponent. In other words, if the opponent marks the corner diagonally opposite to the first move, then the bot marks a random edge, as seen in the left board in Fig. 15, else the bot will need to block the opponent, such as in the right board in Fig. 15. These move sequences usually result in a draw.

(b) *Edge*: if opponent's second move is in an edge unless the bot needs to block the opponent, its second X_2 is in the corner that blocks both the O's. In other words, X_2 is in the corner which is at a minimum distance from O_1 and O_2. These move sequences usually result in a draw, as in Fig. 16.

4.4.2 Opponent starts in an edge: When the opponent starts in an edge, the bot marks any of the two corners next to the opponent's first move O_1, and unless there is a need to block the opponent, which arises only if the opponent has marked O_2 in the

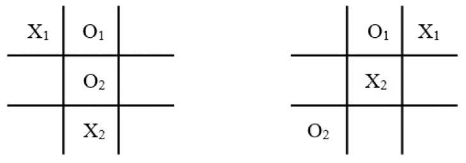


Fig. 17 *Opponent starts in an edge*

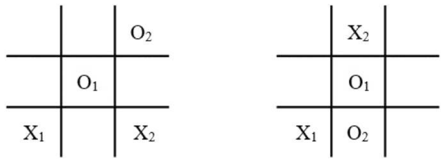


Fig. 18 *Opponent starts in the centre*

Table 1 Specifications of setups used for benchmarking

Specifications	Setup 1	Setup 2	Setup 3
processor	Intel Core™ i5–7200U	Intel Core™ i5-7200U	AMD A4-3330MX APU
cores/threads	2/4	2/4	2/2
base frequency	2.5 GHz	2.5 GHz	2.2 GHz
operating system	Windows 10	Ubuntu 18.04 LTS	Windows 7
RAM	12 GB DDR4	12 GB DDR4	2 GB DDR3

centre, the bot puts second move X_2 in the centre. These move sequences usually result in a draw, such as in Fig. 17.

4.4.3 Opponent starts in the centre: If the opponent starts in the centre, the bot's first move is in a corner, and if there's no need to block (that is if the opponent marks the cell diagonally opposite to the bot's first move), the second bot's move is placed randomly in any of the two remaining corners. This sequence of moves usually leads to a draw, line in the boards in Fig. 18.

4.5 Last few moves

In all of the above, irrespective of whether the bot or the opponent starts the game, after the decision tree has played its part, the rest of the bot's moves comprise the three tasks in the following priority order:

(a) Make a triplet and win.
(b) Block the opponent's triplet.
(c) Randomly select any one of the empty squares; because now there is no longer a way to win (or lose) the game, and the moves are kept randomised to add some variety.

5 Experimental methodology and proposed comparison metrics

This section discusses the implementation details, experimental setup, and the newly formulated notations for this research work. The algorithm is implemented in Java 8. All times are measured in nanoseconds, using the internal nanosecond counter called by the method nanoTime() of system class [53], which has a high degree of precision. Runtimes were calculated for the four algorithms MM, ABA, ABP, and T3DT (please refer to Sections 3 and 4 for a description of these algorithms).

For these comparisons, each algorithm was made to play against itself thousands of times, on Java HotSpot(TM) 64-Bit Server VM, with the JIT compiler both enabled and disabled, on multiple computers having dissimilar specifications to simulate all the possibilities and these statistics have been tabulated. JIT compiler was disabled to prevent the runtime optimisation for this benchmarking. Always the runtimes or the first 50 games were dropped, as they might have some noise due to the cold start of the compiler. Three of these setups whose findings would be presented have their specifications listed in Table 1.

For each particular setup and JIT state (mixed/interpreted), matrix $\boldsymbol{M}$ was created having a number of rows equal to the number of games played and nine columns. Here the matrix notation $\boldsymbol{M}_{ij}$ indicates the time taken by the algorithm to make the jth move in the ith game; j belongs to the finite set $\{x | x \in N, 1 \le x \le 9\}$, while i belongs to the finite set $\{x | x \in N, 1 \le x \le N_{\rm g}\}$, where $N_{\rm g}$ is the number of games played

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{M}_{11} & \boldsymbol{M}_{12} & \cdots & \cdots & \boldsymbol{M}_{19} \\ \boldsymbol{M}_{21} & \ddots & & & \boldsymbol{M}_{29} \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \boldsymbol{M}_{N_{\rm g}1} & \boldsymbol{M}_{N_{\rm g}2} & \cdots & \cdots & \boldsymbol{M}_{N_{\rm g}9} \end{pmatrix} \tag{2}$$

For comparing clock-time analyses across the different setups, this paper formulates the following new notations:

(a) *Time per move (TPM)*: it refers to the response time of each move made by the bot. TPM is defined as the average of all moves of a certain move index j, for a particular setup, and state of the JIT compiler. TPM gives one an idea of how the algorithm is performing at each move. TPM for the jth move is represented by TPM_j and is given by

$$\text{TPM}_j = \frac{\sum_{i=1}^{N_{\rm g}} M_{ij}}{N_{\rm g}} \tag{3}$$

(b) *Time per game (TPG)*: TPG is another metric, which compares each algorithm on the basis of how long all the nine moves take to execute per game, it refers to the total time taken by each algorithm, on average, to play a game from start to finish. TPG for the ith game is represented by TPG_i, and is given by

$$\text{TPG}_i = \sum_{j=1}^{9} M_{ij} \tag{4}$$

TPG, the mean of all TPG_i values, is a very good estimate of the speed of an algorithm, and on a certain setup, it is given by

$$\text{TPG} = \frac{\sum_{i=1}^{N_{\rm g}} \text{TPG}_i}{N_{\rm g}} = \frac{\sum_{i=1}^{N_{\rm g}} \sum_{j=1}^{9} M_{ij}}{N_{\rm g}} \tag{5}$$

Runtimes are calculated for each move on the three setups with JIT compiler enabled and disabled, using the four algorithms being compared. These 24 matrices are then used to calculate TPM_j and TPG, which are used to analyse the performance of the algorithms.

TPG values are tabulated and the effective speedup of T3DT over the competing algorithms is calculated. The speedup is chosen as the benchmarking criterion here because, keeping all other variables (such as setup and JIT state) constant, it gives a true representation of each algorithm's performance. Speedup of T3DT over a certain algorithm X, on a particular setup and JIT state, is given by

$$\text{Speedup of T3DT}(X) = \frac{\text{TPG }(X)}{\text{TPG (T3DT)}} \tag{6}$$

Table 2 Theoretical comparison of the four algorithms

Property	MM	ABP	ABA	T3DT
complete search?	yes	yes	yes	yes
optimal for optimal opponent?	yes	yes	yes	yes
optimal for sub-optimal opponent?	no	no	yes	yes
time complexity	$O(b^m)$	$O(b^{m/2})$	$O(b^{m/2})$	$O(1)$
space complexity	$O(b^m)$ for depth-first search			$O(1)$

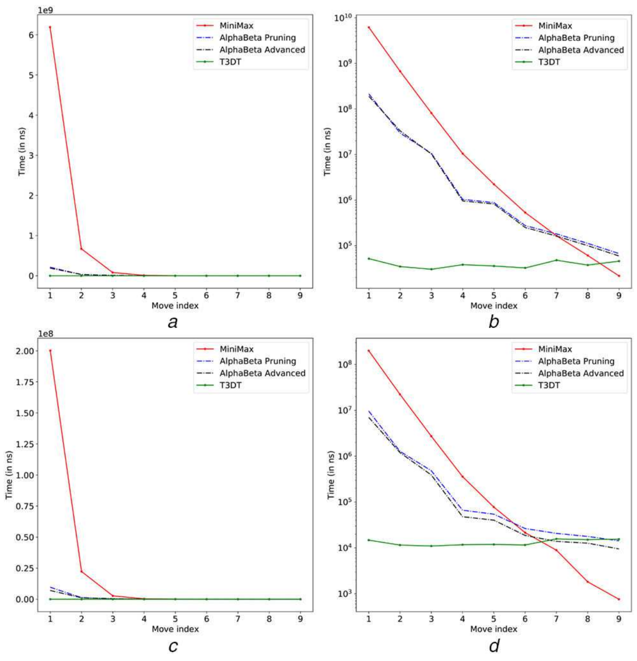


Fig. 19 *TPM$_j$ for setup 1*
(a) Linear scale, no JIT, ***(b)*** Semi-log scale, no JIT, ***(c)*** Linear scale with JIT, ***(d)*** Semi-log scale with JIT

6 Analysis and results

In this section, the claims of improved runtime are supported by irrefutable empirical evidence. Both theoretical and practical clock-time analyses have been done for the algorithms as described below.

6.1 Theoretical analysis

This section explores some of the competing approaches to solve this problem and compares and contrasts the merits and demerits of each. The simplest approach for maximising win-to-draw ratio while still maintaining $O(1)$ time complexity would have been to generate all the possible moves and store them in a tree or directed acyclic graph and map the current board state to the next optimal state(s). Randomisation could be achieved by mapping the current state to a list of optimal next states and selecting one out of them, but this method comes with a high space complexity as it has to store all the sub-trees for the board, maybe even copies of a similar board states, in different sub-trees. Hence this approach was discarded.

However, T3DT based on decision trees has both $O(1)$ space and time complexity, for each move on a fixed 3×3 board, as it stores almost nothing in memory, just the board and about dozen integer values. To make a move, MM would have to construct the entire game tree at each state, which is quite enormous at the beginning. In contrast, T3DT only has to take a few fixed decisions at every step. In some cases, it might not need to analyse the complete board even once to make a move. This leads to constant space and time complexity in practice.

Some more properties such as completeness and optimality of the game tree have been analysed for these algorithms. A search algorithm is said to be complete if an algorithm is guaranteed to reach the goal, provided it exists. All four algorithms are complete by this definition. Optimality refers to the algorithm reaching the goal in the least number of moves. Note that completeness is not a guarantee for optimality. Optimality for these algorithms has been discussed in detail in Section 3 of this paper.

The asymptotic complexities and some other properties such as complete search and optimality for each of the four algorithms being compared MM, ABP, ABA, and T3DT are tabulated in Table 2. Here for the MM-based algorithms, complexity is expressed in terms of b and m, where b is the number of legal moves at each point and m is the depth of the tree.

6.2 Practical clock-time analysis

This section delves into the actual runtime data collected from running each algorithm against itself. The methodology for the same is discussed in Section 5. The results are analysed at a granular level, elaborating why and in which moves T3DT outperforms MM and similar algorithms.

The time taken per move on the three setups is presented in the TPM$_j$ graphs in Figs. 19–21. They show how each algorithm compares against each other for every move. Both linear and semi-log graphs have been used because the linear graphs highlight the significance of the time taken to make the first few moves, while the semi-log scale gives a clearer view of which algorithm performs better in the last few moves when the magnitude of the difference is negligible. The exact time taken for each move from 1 to 9 is also presented in Tables 3–5 for reference and further analysis.

Quite as expected, MM takes the highest time at the first step as it has to go through the entire game tree, which comprises 255,168 unique games. It is followed by MM with ABP and ABA both of which have similar timings. Initially, T3DT used by the bot has the best response time by a large margin, compared to the other algorithms. Gradually this difference decreases, with a decrease in search space and move generation, and in the last few moves, MM-based approaches might have a faster response than the bot.

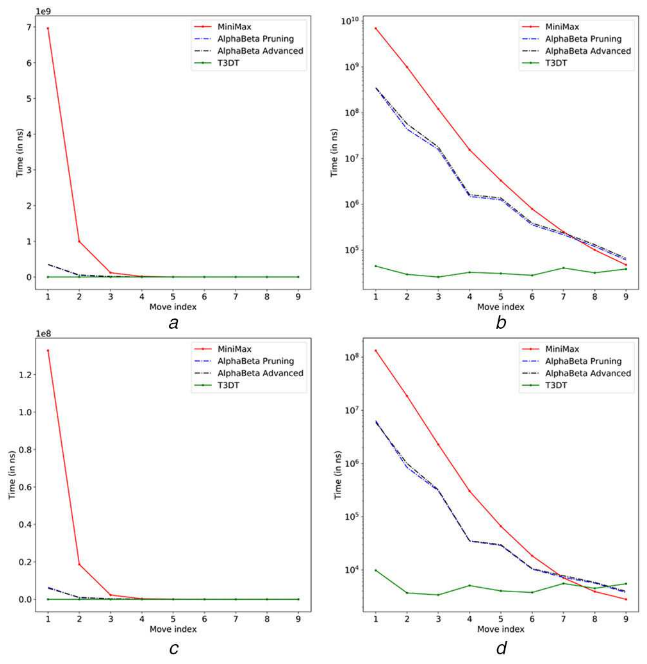


Fig. 20 TPM_j *for setup 2*

(a) Linear scale, no JIT, ***(b)*** Semi-log scale, no JIT, ***(c)*** Linear scale with JIT, ***(d)*** Semi-log scale with JIT

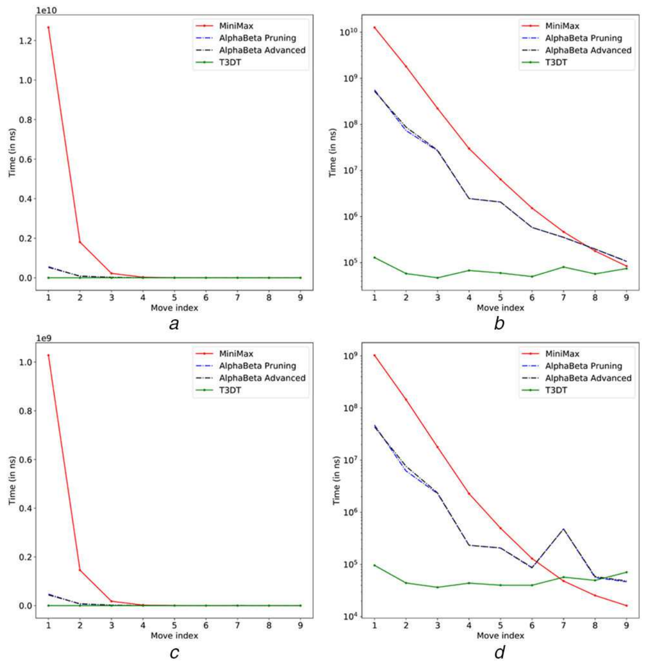


Fig. 21 TPM_j *for setup 3*

(a) Linear scale, no JIT, ***(b)*** Semi-log scale, no JIT, ***(c)*** Linear scale with JIT, ***(d)*** Semi-log scale with JIT

Table 3 TPM values averaged over 10,000 moves for each algorithm on setup 1

Algorithm	1	2	3	4	5	6	7	8	9
MM w/o JIT compiler	6.19×10^9	6.72×10^8	8.13×10^7	1.05×10^7	2.23×10^6	5.34×10^5	1.65×10^5	6.10×10^4	2.17×10^4
ABP w/o JIT compiler	2.14×10^8	2.97×10^7	1.06×10^7	1.04×10^6	8.83×10^5	2.76×10^5	1.82×10^5	1.12×10^5	6.71×10^4
ABA w/o JIT compiler	1.91×10^8	3.31×10^7	1.03×10^7	9.61×10^5	8.25×10^5	2.51×10^5	1.63×10^5	1.00×10^5	5.95×10^4
T3DT w/o JIT compiler	5.19×10^4	3.49×10^4	3.04×10^4	3.83×10^4	3.59×10^4	3.25×10^4	4.80×10^4	3.76×10^4	4.58×10^4
MM	2.00×10^8	2.23×10^7	2.73×10^6	3.59×10^5	7.78×10^4	2.16×10^4	8.95×10^3	1.83×10^3	7.61×10^2
ABP	9.66×10^6	1.28×10^6	4.83×10^5	6.65×10^4	5.45×10^4	2.65×10^4	2.08×10^4	1.78×10^4	1.43×10^4
ABA	7.05×10^6	1.20×10^6	3.90×10^5	4.80×10^4	4.04×10^4	1.87×10^4	1.39×10^4	1.27×10^4	9.53×10^3
T3DT	1.48×10^4	1.15×10^4	1.10×10^4	1.18×10^4	1.19×10^4	1.16×10^4	1.58×10^4	1.52×10^4	1.56×10^4

However, this is insignificant compared to the enormous time taken by these MM-based approaches in the beginning. For reference, time for the first move constitutes about 80–90% approximately of the total TPG while the time for the last move is ∼0% of TPG for

Table 4 TPM values averaged over 10,000 moves for each algorithm on setup 2

Algorithm	1	2	3	4	5	6	7	8	9
MM w/o JIT compiler	6.96×10^9	9.96×10^8	1.21×10^8	1.56×10^7	3.31×10^6	7.96×10^5	2.51×10^5	1.01×10^5	4.79×10^4
ABP w/o JIT compiler	3.49×10^8	4.40×10^7	1.58×10^7	1.49×10^6	1.26×10^6	3.57×10^5	2.18×10^5	1.21×10^5	6.12×10^4
ABA w/o JIT compiler	3.54×10^8	5.72×10^7	1.79×10^7	1.63×10^6	1.38×10^6	3.89×10^5	2.38×10^5	1.32×10^5	6.64×10^4
T3DT w/o JIT compiler	4.49×10^4	2.97×10^4	2.60×10^4	3.29×10^4	3.11×10^4	2.83×10^4	4.10×10^4	3.21×10^4	3.90×10^4
MM	1.33×10^8	1.86×10^7	2.29×10^6	3.04×10^5	6.64×10^4	1.84×10^4	7.09×10^3	3.92×10^3	2.80×10^3
ABP	6.39×10^6	8.44×10^5	3.08×10^5	3.48×10^4	2.89×10^4	1.04×10^4	7.24×10^3	5.67×10^3	3.75×10^3
ABA	5.96×10^6	1.01×10^6	3.21×10^5	3.53×10^4	2.98×10^4	1.06×10^4	7.78×10^3	5.85×10^3	3.95×10^3
T3DT	9.85×10^3	3.70×10^3	3.39×10^3	5.08×10^3	4.03×10^3	3.78×10^3	5.56×10^3	4.52×10^3	5.50×10^3

Table 5 TPM values averaged over 10,000 moves for each algorithm on setup 3

Algorithm	1	2	3	4	5	6	7	8	9
MM w/o JIT compiler	1.27×10^{10}	1.81×10^9	2.22×10^8	3.00×10^7	6.43×10^6	1.53×10^6	4.67×10^5	1.79×10^5	8.45×10^4
ABP w/o JIT compiler	5.62×10^8	7.35×10^7	2.68×10^7	2.47×10^6	2.08×10^6	5.81×10^5	3.54×10^5	1.97×10^5	1.07×10^5
ABA w/o JIT compiler	5.24×10^8	8.73×10^7	2.76×10^7	2.46×10^6	2.07×10^6	5.83×10^5	3.55×10^5	1.98×10^5	1.08×10^5
T3DT w/o JIT compiler	1.29×10^5	5.83×10^4	4.71×10^4	6.79×10^4	5.98×10^4	5.01×10^4	8.06×10^4	5.73×10^4	7.48×10^4
MM	1.03×10^9	1.46×10^8	1.79×10^7	2.29×10^6	4.98×10^5	1.30×10^5	4.82×10^4	2.55×10^4	1.63×10^4
ABP	4.73×10^7	6.22×10^6	2.30×10^6	2.34×10^5	2.07×10^5	8.61×10^4	4.83×10^5	5.62×10^4	4.67×10^4
ABA	4.36×10^7	7.65×10^6	2.37×10^6	2.34×10^5	2.08×10^5	8.79×10^4	4.81×10^5	5.86×10^4	4.84×10^4
T3DT	9.63×10^4	4.42×10^4	3.65×10^4	4.39×10^4	4.01×10^4	4.00×10^4	5.72×10^4	4.97×10^4	7.10×10^4

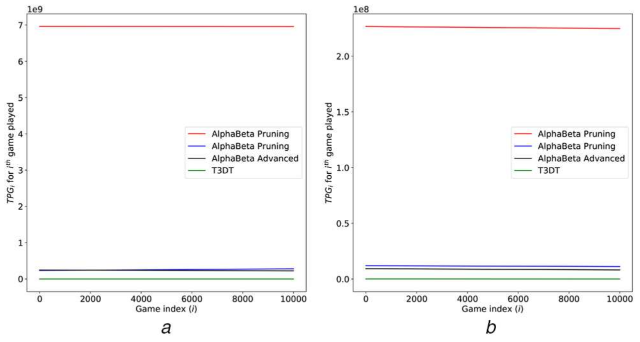


Fig. 22 *Best fit lines of TPG$_i$ on setup 1*
(a) JIT disabled, ***(b)*** JIT enabled

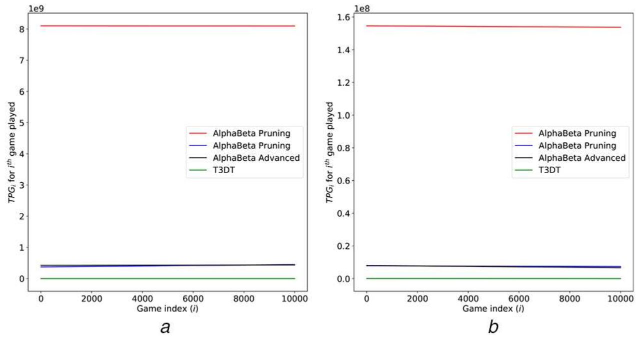


Fig. 23 *Best fit lines of TPG$_i$ on setup 2*
(a) JIT disabled, ***(b)*** JIT enabled

MM, ABP, and ABA. T3DT being a constant time algorithm takes about the same time to make every move.

Furthermore, all three MM-based approaches are not randomised, which leads to all the moves made at the same board state by each algorithm being always the same. The aim of this paper has been to create a randomised algorithm that also tries to maximise win-to-draw ratio. If a simple non-randomised no-loss decision tree algorithm had been implemented, it is highly probable that the decision tree approach would have been the most time-efficient at every move. This is because the randomisation in T3DT was observed to take a significant amount of time. Also, in board states where it is easy for the bot to just force a draw and then move in the first empty square till the end thus saving time, it still explores the possibility to win against a sub-optimal player, not only increasing the win-to draw ratio but also making the game more akin to a human and fun to play with. All these enhancements are novel but time-consuming; however, it only becomes noticeable in the last few moves, when MM itself takes comparably less time.

In Figs. 22–24, each figure representing a setup (please refer specifications in Table 1), the best fit lines are constructed to fit the scattered data of TPG$_i$ for each of the 10,000 games played with index i, for the four algorithms being compared. The order in which the games are played do not matter, hence there is not much to infer from the slope of the lines. However, the lines show the trend of TPG for each algorithm on a particular setup and optimisation state. Randomised T3DT is observed to perform better than even all the other non-randomised methods. The exact values of TPG in Table 6 and standard deviation in Table 7 can be used to infer the nature and spread of the TPG values. T3DT is three orders of magnitude faster than ABA when JIT is disabled, otherwise it is around one–two orders of magnitude faster. It is around four orders of magnitude faster than MM.

Table 8 shows the average speedup (see (6)) of T3DT over the other algorithms on different setups, with and without compiler optimisation. The results obtained are satisfactory and T3DT is about 23,000 times faster than MM without optimisation and about

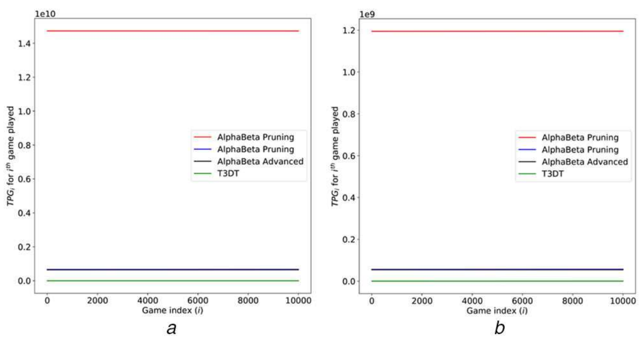


Fig. 24 *Best fit lines of TPG_i on setup 3*
(a) JIT disabled, ***(b)*** JIT enabled

Table 6 TPG for the algorithms on three setups

Algorithm	Setup 1		Setup 2		Setup 3	
	JIT disabled	JIT enabled	JIT disabled	JIT enabled	JIT disabled	JIT enabled
MM	6.96×10^9	2.26×10^8	8.10×10^9	1.54×10^8	1.47×10^{10}	1.19×10^9
ABP	2.57×10^8	1.16×10^7	4.13×10^8	7.64×10^6	6.68×10^8	5.69×10^7
ABA	2.36×10^8	8.78×10^6	4.33×10^8	7.38×10^6	6.45×10^8	5.47×10^7
T3DT	3.55×10^5	1.19×10^5	3.05×10^5	4.54×10^4	6.25×10^5	4.79×10^5

Table 7 Std. dev of TPG for the algorithms on different setups

Algorithm	Setup 1		Setup 2		Setup 3	
	JIT disabled	JIT enabled	JIT disabled	JIT enabled	JIT disabled	JIT enabled
MM	2.61×10^8	3.06×10^7	9.25×10^7	1.95×10^7	4.26×10^7	1.05×10^8
ABP	3.81×10^7	4.97×10^6	4.20×10^7	2.39×10^6	4.66×10^6	4.12×10^7
ABA	2.61×10^7	3.65×10^6	1.41×10^7	2.38×10^6	6.86×10^6	4.11×10^7
T3DT	1.55×10^5	2.39×10^5	6.58×10^4	2.34×10^5	1.07×10^6	1.91×10^6

Table 8 Speedup of T3DT: no. of times it is faster than other methods

Speedup with T3DT	Setup 1		Setup 2		Setup 3		Average	
	JIT disabled	JIT enabled	JIT disabled	JIT enabled	JIT disabled	JIT enabled	JIT disabled	JIT enabled
MM	1.96×10^4	1.89×10^3	2.66×10^4	3.39×10^3	2.36×10^4	2.49×10^3	**23,232**	**2594**
αβ pruning	7.22×10^2	9.75×10^1	1.35×10^3	1.68×10^2	1.07×10^3	1.19×10^2	**1048**	**128**
αβ advanced	6.65×10^2	7.37×10^1	1.42×10^3	1.63×10^2	1.03×10^3	1.14×10^2	**1039**	**117**

Bold value indicates the average across 3 setups and thus give a real-world estimate of the speed boost provided, forming the the crux of this paper.

2500 times faster with optimisation. ABA and ABP are about 1000 and 100 times slower than T3DT without and with JIT enabled.

7 Conclusion and further work

In this paper, a no-loss algorithm for tic-tac-toe is presented, its algorithm explained in detail, and runtime compared against existing methods based on various factors. It has been satisfactorily proved that this algorithm is superior to existing MM-based methods to play tic-tac-toe. It would be useful to try to generate a generalised rule-based approach such as this, creating rules through other methods and then utilising them for playing games of this class but of higher dimensionality. In its current form, the algorithm can be used to improve decision making in heuristic solutions and MM-based solutions to play tic-tac-toe. These rules can also be further analysed to create better fitness function, playing both as the first and second players. Also, it will be interesting to find the exact percentage of victories T3DT achieves out of all possible games of tic-tac-toe in which victory is achievable, including sub-optimal moves by the opponent.

8 Acknowledgments

This work benefitted from discussions with Dr P. Murali, who provided valuable feedback during the final drafting of the paper, his support is gratefully acknowledged.

9 References

[1] Neumann, J. v.: 'Zur theorie der gesellschaftsspiele', *Math. Ann.*, 1928, **100**, (1), pp. 295–320
[2] Kjeldsen, T.H.: 'John von Neumann's conception of the minimax theorem: a journey through different mathematical contexts', *Arch. Hist. Exact Sci.*, 2001, **56**, (1), pp. 39–68
[3] Copeland, A.H.: 'Book review: theory of games and economic behavior', *Bull. Am. Math. Soc.*, 2008, **37**, (1), pp. 103–104
[4] Shannon, C.E.: 'XXII. Programming a computer for playing chess', *London, Edinburgh, Dublin Philos. Mag. J. Sci.*, 1950, **41**, (314), pp. 256–275
[5] Plaat, A., Schaeffer, J., Pijls, W., *et al.*: 'Best-first fixed-depth minimax algorithms', *Artif. Intell.*, 1996, **87**, (1), pp. 255–293
[6] Korf, R.E., Chickering, D.M.: 'Best-first minimax search', *Artif. Intell.*, 1996, **84**, (1), pp. 299–337
[7] Knuth, D.E., Moore, R.W.: 'An analysis of alpha-beta pruning', *Artif. Intell.*, 1975, **6**, pp. 293–326
[8] Darwish, N.M.: 'A quantitative analysis of the alpha–beta pruning algorithm', *Artif. Intell.*, 1983, **21**, (4), pp. 405–433
[9] Baudet, G.M.: 'On the branching factor of the alpha–beta pruning algorithm', *Artif. Intell.*, 1978, **10**, (2), pp. 173–199
[10] Pearl, J.: 'Scout: a simple game-searching algorithm with proven optimal properties', in 'AAAI 1980', 1980
[11] Reinefeld, A.: 'An improvement to the scout tree search algorithm', *ICGA J.*, 1983, **6**, (4), pp. 4–14

[12] Stockman, G.C.: ‘A minimax algorithm better than alpha–beta?’, *Artif. Intell.*, 1979, **12**, (2), pp. 179–196
[13] Roizen, I., Pearl, J.: ‘A minimax algorithm better than alpha–beta? Yes and no’, *Artif. Intell.*, 1983, **21**, (1–2), pp. 199–220
[14] Plaat, A., Schaeffer, J., Pijls, W., *et al.*: ‘SSS* = alpha–beta + TT’, December 1994
[15] Ibaraki, T., Katoh, Y.: ‘Searching minimax game trees under memory space constraint’, *Ann. Math. Artif. Intell.*, 1990, **1**, (1), pp. 141–153
[16] Slagle, J.R., Dixon, J.E.: ‘Experiments with some programs that search game trees’, *J. ACM*, 1969, **16**, (2), pp. 189–207
[17] Shams, R., Kaindl, H., Horacek, H.: ‘Using aspiration windows for minimax algorithms’. 12th Int. Joint Conf. on Artificial Intelligence, **vol. 1**, Sydney, Australia, 1991, pp. 192–197
[18] Kaindl, H., Shams, R., Horacek, H.: ‘Minimax search algorithms with and without aspiration windows’, *IEEE Trans. Pattern Anal. Mach. Intell.*, 1991, **13**, (12), pp. 1225–1235
[19] Berliner, H.: ‘The B∗ tree search algorithm: a best-first proof procedure’, *Artif. Intell.*, 1979, **12**, (1), pp. 23–40
[20] Berliner, H.J., McConnell, C.: ‘B∗ probability based search’, *Artif. Intell.*, 1996, **86**, (1), pp. 97–156
[21] Rivest, R.L.: ‘Game tree searching by min/max approximation’, *Artif. Intell.*, 1987, **34**, (1), pp. 77–96
[22] McAllester, D.A.: ‘Conspiracy numbers for min-max search’, *Artif. Intell.*, 1988, **35**, (3), pp. 287–310
[23] Lister, L., Schaeffer, J.: ‘An analysis of the conspiracy numbers algorithm’, *Comput. Math. Appl.*, 1994, **27**, (1), pp. 41–64
[24] Russell, S., Wefald, E.: ‘On optimal game-tree search using rational meta-reasoning’. 11th Int. Joint Conf. on Artificial Intelligence, 1989, pp. 334–340
[25] Anantharaman, T., Campbell, M.S., Hsu, F.: ‘Singular extensions: adding selectivity to brute-force searching’, *Artif. Intell.*, 1990, **43**, (1), pp. 99–109
[26] Björnsson, Y., Marsland, T.A.: ‘Risk management in game-tree pruning’, *Inf. Sci.*, 2000, **122**, (1), pp. 23–41
[27] Hyatt, R.M., Suter, B.W., Nelson, H.L.: ‘A parallel alpha/beta tree searching algorithm’, *Parallel Comput.*, 1989, **10**, (3), pp. 299–308
[28] Borovska, P., Lazarova, M.: ‘Efficiency of parallel minimax algorithm for game tree search’. Proc. 2007 Int. Conf. on Computer Systems and Technologies, Rousse, Bulgaria, 2007, pp. 14:1–14:6
[29] Marsland, T.A., Campbell, M.: ‘Parallel search of strongly ordered game trees’, *ACM Comput. Surv.*, 2002, **14**, (4), pp. 533–551
[30] Li, L., Liu, H., Wang, H., *et al.*: ‘A parallel algorithm for game tree search using GPGPU’, *IEEE Trans. Parallel Distrib. Syst.*, 2015, **26**, (8), pp. 2114–2127
[31] Ahmed, A., Abdel, M., Gadallah, M., *et al.*: ‘A comparative study of game tree searching methods’, *Int. J. Adv. Comput. Sci. Appl.*, 2014, **5**, (5), pp. 68–77
[32] Campbell, M.S., Marsland, T.A.: ‘A comparison of minimax tree search algorithms’, *Artif. Intell.*, 1983, **20**, (4), pp. 347–367
[33] Diderich, C.G., Gengler, M.: ‘Minimax game tree searching minimax game tree searching’, in Floudas, C.A., Pardalos, P.M. (Eds.): ‘*Encyclopedia of optimization*’ (Springer, USA, 2009), pp. 2079–2087
[34] ‘Mathematical recreations, repository for mathematical diversions’. Available at http://www.mathrec.org/old/2002jan/solutions.html, accessed May 2019
[35] de Groot, A.D.: ‘*Thought and choice in chess*’ (Amsterdam University Press, Amsterdam, Netherlands, 2008)
[36] Fogel, D.B.: ‘Using evolutionary programming to create neural networks that are capable of playing tic-tac-toe’. IEEE Int. Conf. on Neural Networks – Conf. Proc., San Francisco, CA, USA., 1993, pp. 875–880
[37] Chellapilla, K., Fogel, D.B.: ‘Evolution, neural networks, games, and intelligence’, *Proc. IEEE*, 1999, **87**, (9), pp. 1471–1496
[38] Hochmuth, G.: ‘On the genetic evolution of a perfect tic-tac-toe strategy’, *Genet. Algorithms Genet. Program.*, 2003, pp. 75–82
[39] Soedarmadji, E.: ‘Decentralized decision making in the game of tic-tac-toe’. 2006 IEEE Symp. on Computational Intelligence Games, Reno, NV, USA., 2006, vol. 6, pp. 34–38
[40] Yau, Y.J., Teo, J.: ‘An empirical comparison of non-adaptive, adaptive and self-adaptive co-evolution for evolving artificial neural network game players’. 2006 IEEE Conf. on Cybernetics and Intelligent Systems, Bangkok, Thailand, 2006, pp. 1–6
[41] Yau, Y.J., Teo, J., Anthony, P.: ‘Pareto evolution and co-evolution in cognitive neural agents synthesis for tic-tac-toe’. Proc. 2007 IEEE Symp. on Computational Intelligence Games (CIG 2007), Honolulu, HI, USA., 2007, pp. 304–311
[42] Bhatt, A., Varshney, P., Deb, K.: ‘In search of no-loss strategies for the game of tic-tac-toe using a customized genetic algorithm’. Genetic and Evolutionary Computation Conf. (GECCO 2008), Atlanta, GA, USA., 2008, pp. 889–896
[43] Mohammadi, H., Browne, N.P.A., Venetsanopoulos, A.N., *et al.*: ‘Evolving tic-tac-toe playing algorithms using co-evolution, interactive fitness and genetic programming’, *Int. J. Comput. Theory Eng.*, 2013, **5**, (5), pp. 797–801
[44] Ling, S.S., Lam, H.K.: ‘Playing tic-tac-toe using genetic neural network with double transfer functions’, *J. Intell. Learn. Syst. Appl.*, 2011, **3**, pp. 37–44
[45] Rajani, N.F., Dar, G., Biswas, R., *et al.*: ‘Solution to the tic-tac-toe problem using hamming distance approach in a neural network’. Proc. 2011 2nd Int. Conf. on Intelligent System Modelling Simulation (ISMS 2011), Kuala Lumpur, Malaysia/Phnom Penh, Cambodia, 2011, no. 1, pp. 3–6
[46] Singh, A., Deep, K., Nagar, A.: ‘A ‘never-loose’ strategy to play the game of tic-tac-toe’. Proc. 2014 Int. Conf. on Soft Computing Machine Intelligence (ISCMI 2014), New Delhi, India, 2014, pp. 1–5
[47] Karamchandani, S., Gandhi, P., Pawar, O., *et al.*: ‘A simple algorithm for designing an artificial intelligence based tic tac toe game’. 2015 Int. Conf. on Pervasive Computing Advance Communication Technology Applied Society (ICPC 2015), Pune, India, 2015, vol. 00, no. c, pp. 1–4
[48] Garg, S., Songara, D.: ‘The winner decision model of tic tac toe game by using multi-tape turing machine’. 2016 Int. Conf. on Advances Computing Communications and Informatics (ICACCI 2016), Jaipur, India, 2016, pp. 573–579
[49] Garg, S., Songara, D., Maheshwari, S.: ‘The winning strategy of tic tac toe game model by using theoretical computer science’. 2017 Int. Conf. on Computer, Communications and Electronics (COMPTELIX 2017), Jaipur, India, 2017, pp. 89–95
[50] Sriram, S., Vijayarangan, R., Raghuraman, S., *et al.*: ‘Implementing a no-loss state in the game of tic-tac-toe using a customized decision tree algorithm’. 2009 IEEE Int. Conf. on Information Automation (ICIA 2009), Zhuhai/ Macau, China, 2009, pp. 1211–1216
[51] Puzzleland: ‘*Tic tac toe: 8 strategies to win every game*’ (Puzzleland, 2016)
[52] Ayock, R.: ‘How to win at tic-tac-toe’, 2002
[53] Oracle: ‘Java™ platform, standard edition 8 API specification’. Available at https://docs.oracle.com/javase/8/docs/api/index.html, accessed April 2020