Scientific Research Publishing

Journal of Intelligent Learning Systems and Applications, 2018, 10, 46-79
http://www.scirp.org/journal/jilsa
ISSN Online: 2150-8410
ISSN Print: 2150-8402

Prediction Distortion in Monte Carlo Tree Search and an Improved Algorithm

William Li

Delbarton School, Morristown, NJ, USA
Email: liwilliam2021@gmail.com

How to cite this paper: Li, W. (2018) Prediction Distortion in Monte Carlo Tree Search and an Improved Algorithm. *Journal of Intelligent Learning Systems and Applications*, **10**, 46-79.
https://doi.org/10.4236/jilsa.2018.102004

Received: March 11, 2018
Accepted: May 28, 2018
Published: May 31, 2018

Abstract

Teaching computer programs to play games through machine learning has been an important way to achieve better artificial intelligence (AI) in a variety of real-world applications. Monte Carlo Tree Search (MCTS) is one of the key AI techniques developed recently that enabled AlphaGo to defeat a legendary professional Go player. What makes MCTS particularly attractive is that it only understands the basic rules of the game and does not rely on expert-level knowledge. Researchers thus expect that MCTS can be applied to other complex AI problems where domain-specific expert-level knowledge is not yet available. So far there are very few analytic studies in the literature. In this paper, our goal is to develop analytic studies of MCTS to build a more fundamental understanding of the algorithms and their applicability in complex AI problems. We start with a simple version of MCTS, called random playout search (RPS), to play Tic-Tac-Toe, and find that RPS may fail to discover the correct moves even in a very simple game position of Tic-Tac-Toe. Both the probability analysis and simulation have confirmed our discovery. We continue our studies with the full version of MCTS to play Gomoku and find that while MCTS has shown great success in playing more sophisticated games like Go, it is not effective to address the problem of sudden death/win. The main reason that MCTS often fails to detect sudden death/win lies in the random playout search nature of MCTS, which leads to prediction distortion. Therefore, although MCTS in theory converges to the optimal minimax search, with real world computational resource constraints, MCTS has to rely on RPS as an important step in its search process, therefore suffering from the same fundamental prediction distortion problem as RPS does. By examining the detailed statistics of the scores in MCTS, we investigate a variety of scenarios where MCTS fails to detect sudden death/win. Finally, we propose an improved MCTS algorithm by incorporating minimax search to overcome prediction distortion. Our simulation has confirmed the effectiveness of the proposed algorithm. We provide an estimate of the additional computational

costs of this new algorithm to detect sudden death/win and discuss heuristic strategies to further reduce the search complexity.

Keywords

Monte Carlo Tree Search, Minimax Search, Board Games, Artificial Intelligence

1. Introduction

Teaching computer programs to play games through learning has been an important way to achieve better artificial intelligence (AI) in a variety of real-world applications [1]. In 1997 a computer program called IBM Deep Blue defeated Garry Kasparov, the reigning world chess champion. The principle idea of Deep Blue is to generate a search tree and to evaluate the game positions by applying expert knowledge on chess. A search tree consists of nodes representing game positions and directed links each connecting a parent and a child node where the game position changes from the parent node to the child node after a move is taken. The search tree lists computer's moves, opponent's responses, and computer's next-round responses and so on. Superior computational power allows the computer to accumulate vast expert knowledge and build deep search trees, thereby greatly exceeding the calculation ability of any human being. However, the same principle cannot be easily extended to play Go, an ancient game still very popular in East Asia, because the complexity of Go is much higher: the board size is much larger, games usually last longer, and more importantly, it is harder to evaluate game positions. In fact the research progress had been so slow that even an early 2016 Wikipedia [2] noted that "Thus, it is very unlikely that it will be possible to program a reasonably fast algorithm for playing the Go endgame flawlessly, let alone the whole Go game". Still, to many people's surprise, in March 2016, AlphaGo, a computer program created by Google's DeepMind, won a five game competition by 4-1 over Lee Sedol, a legendary professional Go player. An updated version of AlphaGo named "Master" defeated many of the world's top players with an astonishing record of 60 wins 0 losses, over seven days, in December 2016. AlphaGo accomplished this milestone because it used a very different set of AI techniques from what Deep Blue had used [3]. Monte Carlo Tree Search (MCTS) is one of the two key techniques (the other is deep learning) [4].

MCTS generates a search tree by running a large number of simulated playouts. Each playout starts at the current game state and simulates a random sequence of legal moves until the game ends. By analyzing the statistics, MCTS estimates which move is more likely to win and takes the statistically best move. MCTS was first proposed in 2006 [5] [6] [7] and has since received considerable interest due to its success in Go. What makes MCTS particularly attractive is

that it only understands the basic rules of Go, and unlike the Deep Blue approach, does not rely on expert-level knowledge in Go. Researchers thus expect that MCTS can be applied to other complex AI problems, such as autonomous driving where domain-specific expert-level knowledge is not yet available.

Most studies on MCTS in the literature are based on simulation [8]. There are very few analytic studies, which would contribute to a more fundamental understanding of the algorithms and their applicability in complex AI problems. A key difficulty of rigorous analysis comes from the iterative, stochastic nature of the search algorithm and sophisticated, strategic nature of games such as Go. In this paper we first study Random Playout Search (RPS), a simplified version of MCTS that is more tractable mathematically. The simplification is that unlike general MCTS, RPS does not expand the search tree, reflecting an extreme constraint of very little computational resource. We use a probabilistic approach to analyze RPS moves in a recursive manner and apply the analysis in the simple Tic-Tac-Toe game to see whether RPS is a winning strategy. We then extend our observations to the full version of MCTS by controlling the number of simulation iterations and play the more complex Gomoku game. We find that, with the computational resource constraint, MCTS is not necessarily a winning strategy because of prediction distortion in the presence of sudden death/win. Finally we propose an improved MCTS algorithm to overcome prediction distortion. Our analysis is validated through simulation study.

The remaining of this paper is organized as followed. Section 2 investigates RPS playing Tic-Tac-Toe with probability analysis and simulation and highlights the drawback of RPS. Section 3 discusses a simple solution to address the drawback of RPS and extend the discussion to introduce MCTS. Section 4 investigates the problem of sudden death/win, presents two propositions on the capability of detecting sudden death/win with a minimax search and MCTS, and examines the problem of prediction distortion in the context of MCTS playing Gomoku with detailed simulation results. Section 5 proposes an improved MCTS algorithm to enhance the capability to detect sudden death/win, analyzes the additional complexity and presents heuristic strategies to further reduce the search complexity. **Appendix A** describes the Java implementation of the analysis and simulation codes used in this study.

2. Random Playout Search in Tic-Tac-Toe Game

The Tic-Tac-Toe game is probably the simplest board game. We propose a simplified version of MCTS, Random Playout Search (RPS), to play the Tic-Tac-Toe game in order to build a mathematical model for analysis, and hope that the insight from our analysis is applicable to MCTS playing other more complex games.

2.1. Random Playout Search

To describe RPS, consider the current game position shown in **Figure 1**. It is the

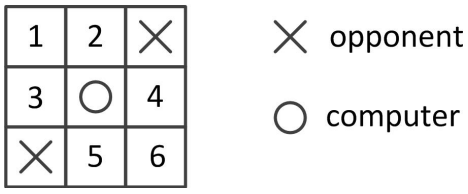


Figure 1. RPS in Tic-Tac-Toe game.

computer's turn to move. The computer can move to one of 6 possible positions, labeled 1 - 6. If the computer takes 1, then the opponent can take 6, which blocks the diagonal and opens up a vertical and a horizontal to win.

However, RPS does not have this ability to strategically look beyond a few steps. Instead, RPS employs a large number of simulated playouts $t=1,2,\cdots$ and estimates the expected score of move *i*, for $i=1\text{-}6$. **Figure 2(a)** describes the flow chart of RPS and (b) provides the details of function block "simulate from current game position to endgame".

Denote $M_i(t)$ the number of times that move *i* has been picked up to playout *t* and $N_i(t)$ the number of wins minus the number of losses. Initially, $M_i(0)=N_i(0)=0$ for all *i*. Each playout starts with the current game position. RPS randomly picks next move *i* and increments $M_i(t)$ by 1. RPS then picks a sequence of random moves for the computer and the opponent until the game ends. If the computer wins, then $N_i(t)$ increments by 1; if the computer loses, then $N_i(t)$ decrements by 1; otherwise, $N_i(t)$ is unchanged. The score is calculated as follows,

$$S_i(t)=\frac{N_i(t)}{M_i(t)}. \tag{1}$$

When the preset number of playouts T has been reached, RPS selects the move i^* with the highest score,

$$i^*=\arg\max_i S_i(T). \tag{2}$$

2.2. Probability Analysis

To assess the performance of RPS, we calculate $s_{\mathcal{P}_0,i}$, the expected score of move *i* from the current game position $\mathcal{P}_0$. To simplify the notation, $s_{\mathcal{P}_0,i}$ is sometimes reduced to s_i. Denote $\mathcal{P}_1$ the game position after move *i* from $\mathcal{P}_0$, where subscript 1 indicates that $\mathcal{P}_1$ is on level-1 from the starting position $\mathcal{P}_0$ on level-0. There are multiple game positions on level-1 and $\mathcal{P}_1$ is one of them. Denote $K_{\mathcal{P}_1}$ the number of legal moves from position $\mathcal{P}_1$. Because in a simulated playout RPS uniformly randomly selects the next move of the opponent, we have the following recursive equation

$$s_{\mathcal{P}_0,i}=s_{\mathcal{P}_1}=\frac{1}{K_{\mathcal{P}_1}}\sum_{j=1}^{K_{\mathcal{P}_1}}s_{\mathcal{P}_1,j}, \tag{3}$$

where $s_{\mathcal{P}_1,j}$ is the expected score of move *j* from $\mathcal{P}_1$. Equation (3) can be further expanded from every move *j* of $\mathcal{P}_1$ to the next levels until terminal

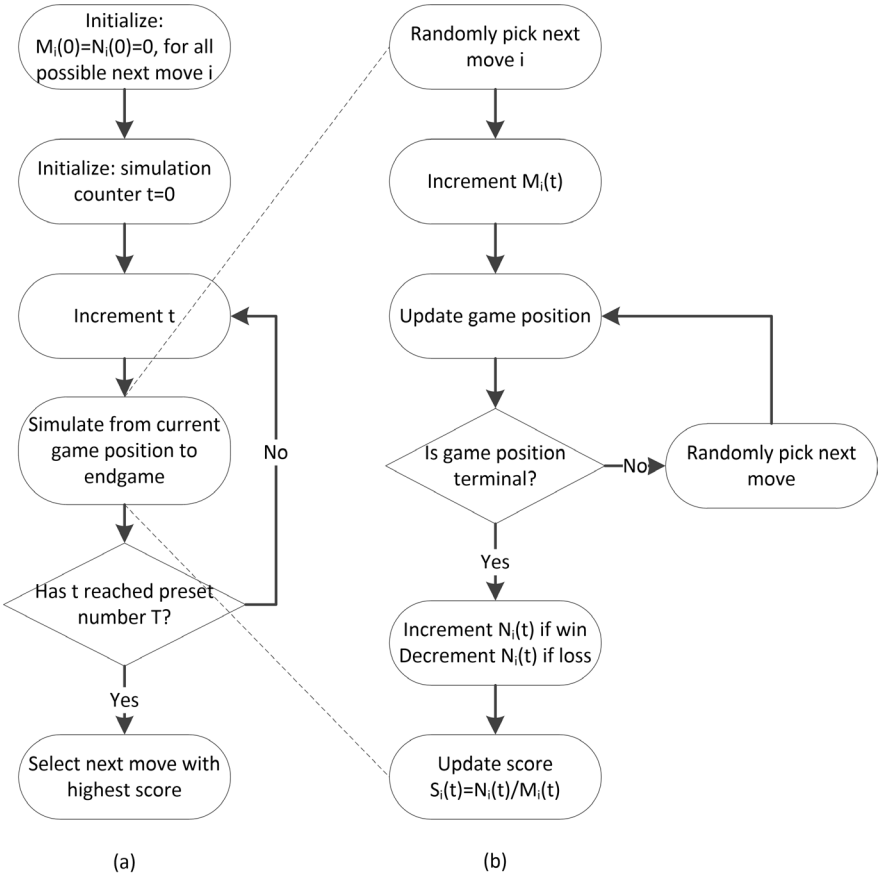


Figure 2. Flow chart of RPS.

positions are reached. A position is terminal if the game ends at the position, and the game result (loss, draw, win) determines the expected score of the terminal position to be $-1,0,1$, respectively.

While Equation (3) starts from the current position $\mathcal{P}_0$, it is solved by walking backwards from the terminal positions. An example of solving Equation (3) is illustrated in **Figure 3**, where (a) depicts the current game position $\mathcal{P}_0$ and 4 possible moves and (b) shows the calculation of $s_{\mathcal{P}_0,1}$.

Specifically, the game position $\mathcal{P}_1$ after move 1 from $\mathcal{P}_0$ is labeled as A, which is expanded into $K_A = 3$ next level positions B,C,D. B is a terminal position (loss), so $s_B = -1$. C is further expanded into $K_C = 2$ next level positions E, F. E leads to a terminal position G (draw) where $s_E = s_G = 0$. F leads to a terminal position H (loss), so $s_F = s_H = -1$. Thus, from Equation (3),

$$s_C = \frac{1}{K_C}\left(s_E + s_F\right) = -\frac{1}{2}.$$

Similarly, D is further expanded into $K_D = 2$ next level positions I, J. I leads to a terminal position K (draw) where $s_I = s_K = 0$. J leads to a terminal position L (loss), so $s_J = s_L = -1$. Thus,

$$s_D = \frac{1}{K_D}\left(s_I + s_J\right) = -\frac{1}{2}.$$

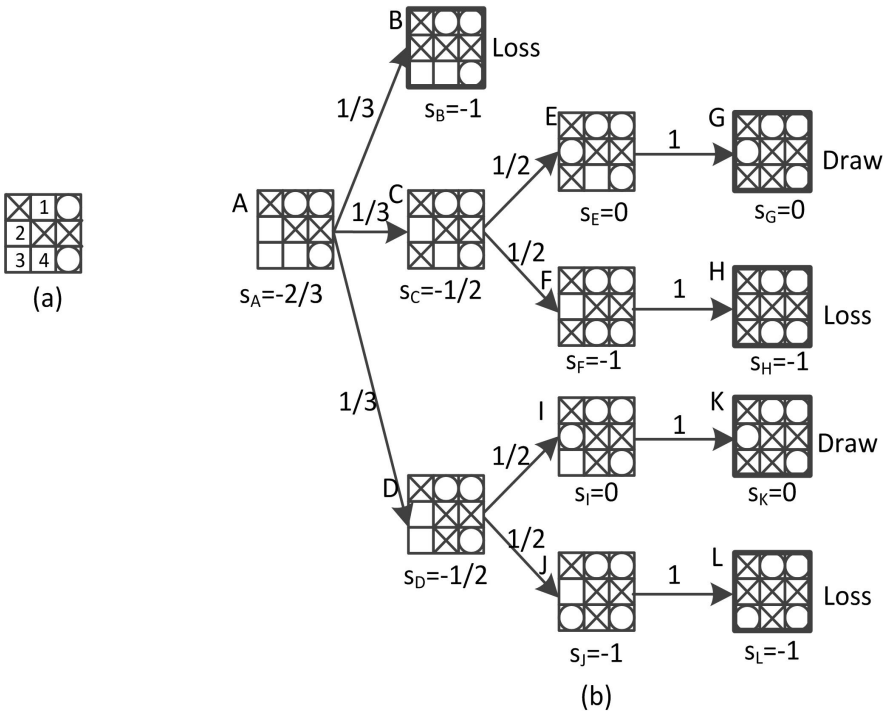


Figure 3. Example of calculating expected score of a move with recursive Equation (3). A terminal position is marked by a thick square.

Finally, from Equation (3),

$$s_{\mathcal{P}_0,1} = s_A = \frac{1}{K_A}\left(s_B + s_C + s_D\right) = -\frac{2}{3}. \tag{4}$$

Using the same recursive approach, we can obtain

$$s_{\mathcal{P}_0,2} = -\frac{1}{3}, s_{\mathcal{P}_0,3} = -\frac{2}{3}, s_{\mathcal{P}_0,4} = -\frac{1}{3}. \tag{5}$$

Because $s_{\mathcal{P}_0,2} = s_{\mathcal{P}_0,4} > s_{\mathcal{P}_0,1} = s_{\mathcal{P}_0,3}$, according to Equation (2) RPS selects either move 2 or 4 with equal probability as the next move from $\mathcal{P}_0$.

2.3. Simulation Results

We expect that

$$s_i = \lim_{t\to\infty} S_i(t). \tag{6}$$

Equation (6) states that $S_i(t)$ given in (1) converges to s_i, the expected score calculated with the recursive analysis model (3) as the number of playouts increases. We next validate (6) with simulation.

We have simulated the RPS algorithm in Java to play Tic-Tac-Toe according to **Figure 2** and implemented the recursive score calculation (3). **Figure 4** shows the simulation results of the estimated scores $S_i(t)$ of 4 possible moves from the game position in **Figure 3(a)**. From **Figure 4**, we observe that $S_i(t)$ does converge to s_i; however, a large number of simulation playouts are needed for convergence even for a simple game position.

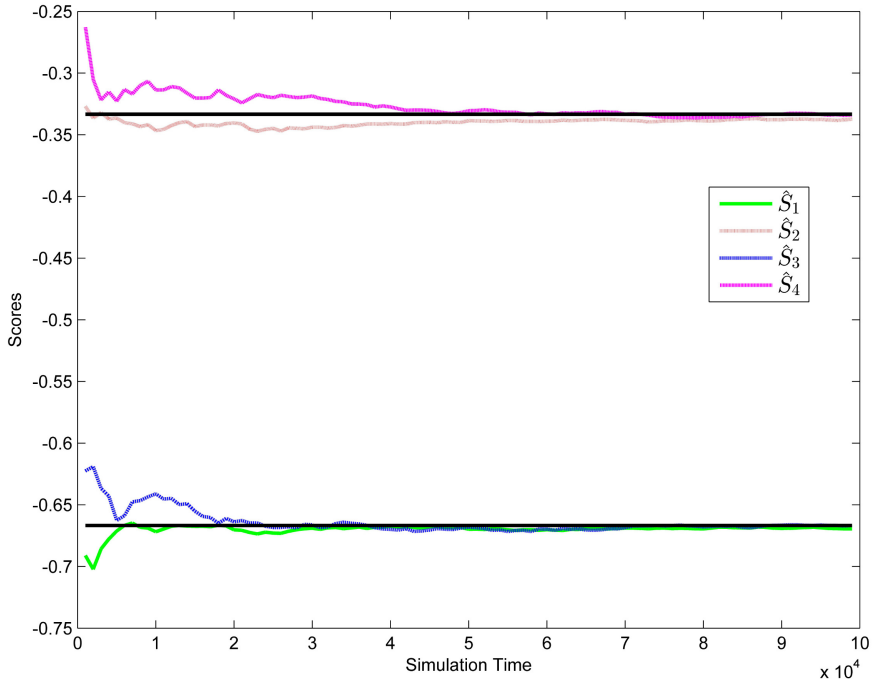


Figure 4. Comparison of simulation results and analysis model. The black straight lines represent s_1 to s_4, the expected scores calculated with the analysis model (4) (5).

3. From Random Playout Search to Monte Carlo Tree Search

3.1. Expansion of Search Depth in RPS

The above analysis and simulation show that for the simple game position in **Figure 3(a)**, RPS selects either move 2 or 4 as the next move, even though only 2 is the right move. We have examined other game positions and found that RPS is often *not* a winning strategy. **Figure 5** lists additional examples of game positions for which RPS fails to find out the right move.

The reason of RPS failure in those game positions is that in the simulated playouts RPS assumes that the opponent randomly picks its moves. However, in the actual game an intelligent opponent does not move randomly. As a result, the prediction ability of RPS is distorted. In the example of **Figure 3(b)**, when calculating s_A, it is assumed that the opponent moves to *B*, *C*, *D* with equal probability. However, it is obvious that an intelligent opponent will move to *B* to win the game and never bother with *C*, *D*. The same thing happens in the RPS simulated playouts.

One way to overcome the prediction distortion is to expand the search depth of RPS. Take **Figure 3(a)** for example. So far RPS only keeps the scores of 4 possible level-1 moves from the current game position. Playouts are simulated from these 4 moves, but RPS does not keep track of the scores below level-1. In this case, the search depth is 1, as illustrated in **Figure 6(a)**, where the winner moves are marked with the thick lines and the dashed lines represent simulated playouts from one position to a terminal position shown as a thick square.

In **Figure 6(b)**, we expand the search depth to 2 by keeping the scores of all

Figure 5. Additional examples where RPS fails to find out the right move. In each example, move 1 is the right move, but RPS selects move 2 because move 2 has a higher score than move 1.

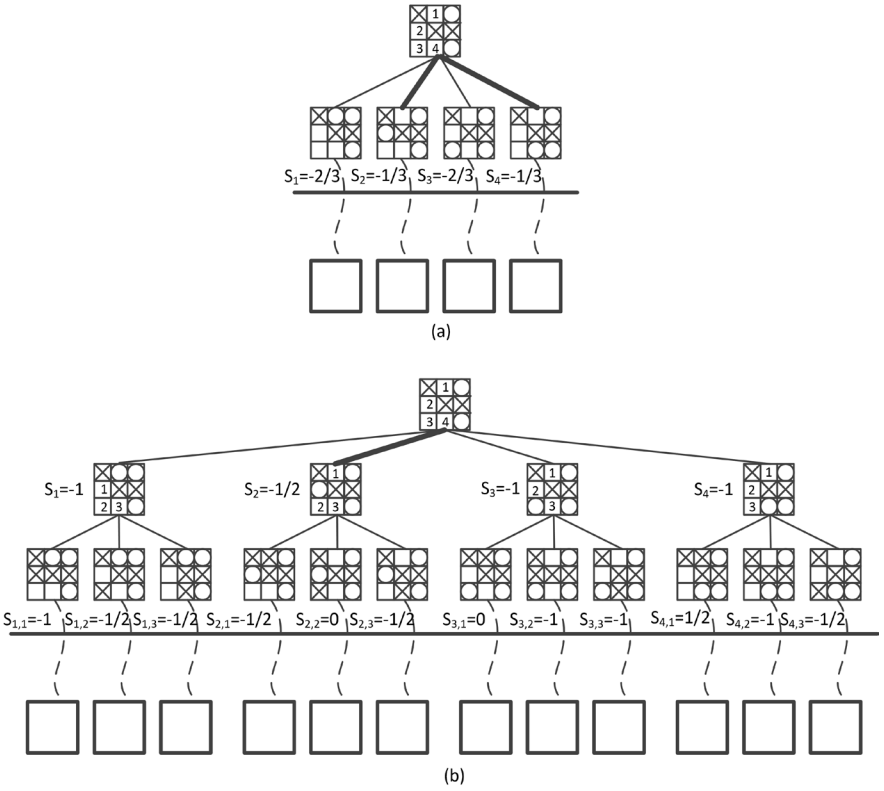


Figure 6. Expansion of search depth of RPS: search depth = 1 in (a) and 2 in (b). In (a), S_1 is what is labeled as S_A in **Figure 3**.

the level-2 moves, which are obtained from the simulated playouts. The scores of the level-1 moves are calculated from those of the level-2 moves with the minimax principle. That is, the opponent selects the best move, *i.e.*, with minimum score, instead of a random move on level-2,

$$S_i(t) = \min_{j=1,2,3} S_{i,j}(t), \tag{7}$$

for $i = 1, \cdots, 4$. In comparison, when the search depth is 1, although RPS does not keep track of $S_{i,j}(t)$, in effect $S_i(t)$ is estimated to be

$$S_i(t) = \frac{1}{3}\sum_{j=1,2,3} S_{i,j}(t). \tag{8}$$

Comparison of Equations (7) and (8) shows how the prediction distortion is corrected for the opponent's move on level-2 by increasing the search depth to 2.

Figure 6(b) shows that now RPS selects the right move (2), as opposed to either move 2 or 4 in Figure 6(a). Through simulation and analysis, we confirm that RPS is able to select the right move for the other game positions of Figure 5.

3.2. Brief Summary of MCTS

The idea of expanding the search depth can be generalized to mean that a good algorithm should search beyond the level-1 moves. MCTS takes this idea even further. Starting from the current game position, called the root node, the MCTS algorithm gradually builds a search tree through a number of iterations and stops when some computational budget is met. Similar to the one shown in Figure 6, a search tree consists of nodes representing game positions and directed links each connecting a parent and a child nodes where the game position changes from the parent node to the child node after a move is taken. Every node j in the search tree keeps the total score it gets X_j and the number of times it has been visited n_j.

Each iteration in MCTS consists of four steps as illustrated in Figure 7.

1) Selection: A child node is selected recursively, according to a tree policy, from the root node to a leaf node, *i.e.*, a node on the boundary of the search tree.

2) Expansion: A new node is added to the search tree as a child node of the leaf node.

3) Simulation: A simulated playout runs from the new node until the game ends, according to a default policy. The default policy is simply RPS, *i.e.*, randomly selecting moves between the computer and the opponent. The simulation result determines a simulation score: +1 for win, −1 for loss, and 0 for draw.

4) Backpropagation: The simulation score (+1, −1, or 0) is backpropagated from the new node to the root node through the search tree to update their scores. A simple algorithm is that the score is added to X_j of every node j along the search tree.

Recall that the nodes from the new node to the terminal node are outside the boundary of the search tree. Any node outside the boundary of the search tree does not keep track of any score.

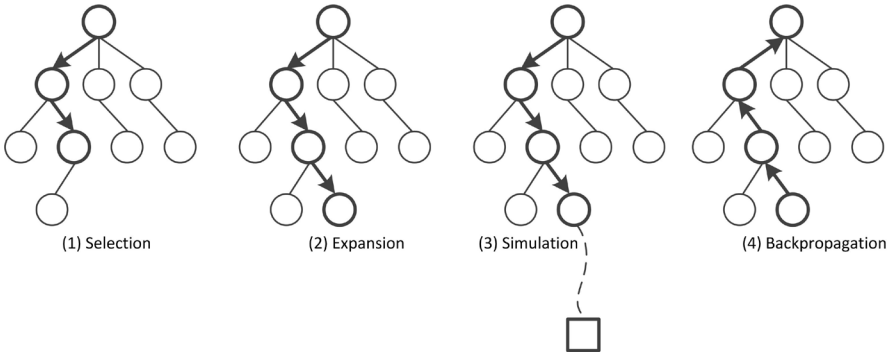


Figure 7. Illustration of one iteration in MCTS. A circle represents a game position in the search tree. A square represents a terminal position.

The most popular tree policy in MCTS is based on the Upper Confidence Bounds applied to Trees (UCT) algorithm [9] [10] whose idea is to focus the search on more promising moves and meanwhile explore unvisited game positions. To describe the UCT algorithm, consider (1) of **Figure 7**. The root node has three child nodes, j = 1, 2, 3, and selects the one that maximizes the UCB score

$$UCB1 = \frac{X_j}{n_j} + C\sqrt{\frac{2\ln n}{n_j}}, \tag{9}$$

where X_j is the total score of child node j, n_j the number of times that child node j has been tried, and $n = \sum_j n_j$ the total number of times that all child nodes have been tried. The first term average score

$$S_j = \frac{X_j}{n_j} \tag{10}$$

aims to exploit higher-reward moves (exploitation), while the second term $\sqrt{\frac{2\ln n}{n_j}}$ encourages examining less-visited moves (exploration), and parameter C balances exploitation and exploration.

After the algorithm stops, the child node of the root node with the highest average score S_j is selected and the corresponding move from the root node to the selected child node is taken in the actual game.

4. Monte Carlo Tree Search in Gomoku Game

An important theoretical result [8] is that given enough run time and memory, MCTS converges to the optimal tree and therefore results in the optimal move. As shown in **Figure 7**, the search tree adds one node in the expansion step of every iteration. Within the search tree, UCT selects nodes on the basis of their scores and can thus identify moves of great promise. Outside the boundary of the search tree, MCTS uses RPS in the simulation step. The reason that MCTS converges to the optimal move is that with an infinite amount of search all possible game positions are added to the search tree and that UCT in effect behaves like minimax, which is optimal.

In practice, however, MCTS is limited by computational resource and the search tree is a small subset of the entire game position space. Therefore, MCTS has to rely on RPS to explore a large game position space that is outside the search tree. From the study of Tic-Tac-Toe we learn that a major drawback of RPS is that it assumes the opponent is playing randomly when in fact it is possible that the opponent knows exactly what it should do strategically.

To study MCTS, we need more sophisticated games than Tic-Tac-Toe. In this section, we play Gomoku and see how the insight from the study of Tic-Tac-Toe applies here.

4.1. Sudden Death/Win and Prediction Distortion

Figure 8 shows two game positions in Gomoku where it is currently the

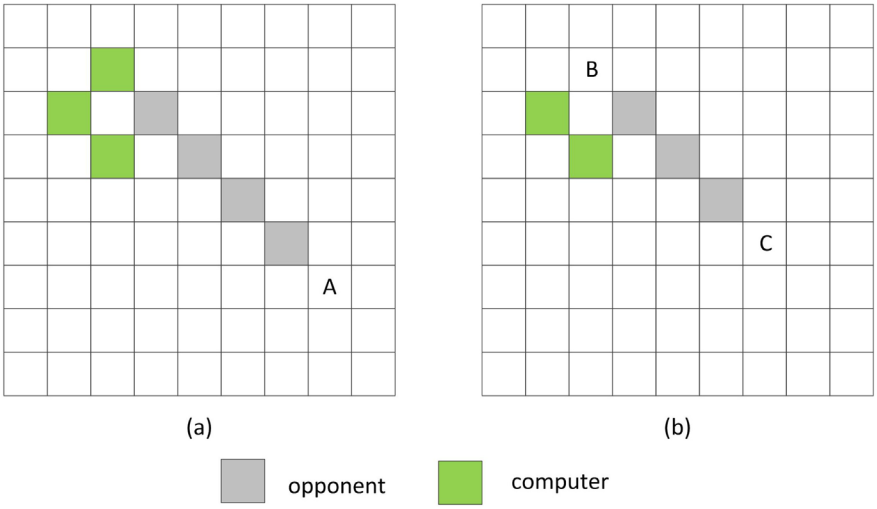


Figure 8. Illustration of sudden death in Gomoku: (a) level-2 and (b) level-4.

computer's turn to move. In (a), without taking move A, the computer loses in two steps. In (b), without taking move B or C, the computer loses in four steps. On the other hand, if the computer is black and the opponent is green in Figure 8, then the computer wins in one step in (a) by taking move A and wins in three steps in (b) by taking move B or C.

Definition 1. ***Level-k sudden death or sudden win***

A level-k sudden death is a game position where the computer will lose the game in k steps if the computer does not take one of some subset of moves. A level-k sudden win is a game position where the computer will win the game in k steps if the computer takes one of some subset of moves.

The subset of moves is called *decisive moves*. The level k is the number of steps that results in a terminal position from the current game position. k is even for sudden death and odd for sudden win. k is a small number for sudden death or win; otherwise, death or win is not "sudden". More importantly, because of computational resource constraints, it is impractical to detect definite death or win for a large k. In Figure 8, the game position is a level-2 sudden death in (a) and a level-4 sudden death in (b).

One way to detect a level-k sudden death or win is to employ a level-k minimax search, a slightly modified version of minimax search. Specifically, starting from the current game position (root node at level-0), repeatedly expand to the next level by adding the child nodes that represent all the possible next moves from the parent node. Node expansion ends at level-k. A node of terminal position is assigned a score of −1, 0, 1 as in Section 2.2. If a node at level-k is not a terminal position, it is assigned a score of 0, because its state is undetermined. Backpropagate the scores to the root node according to the minimax principle.

Figure 9 illustrates an example of level-5 minimax search. The root node at level-0 has two child nodes, A_1, A_2, in total, A_1 has two child nodes B_1, B_2, and so on. At level-5, E_1, E_4, E_9, E_{12} are terminal positions and assigned a score of 1, and all other nodes are not terminal and assigned a score of 0. The minimax

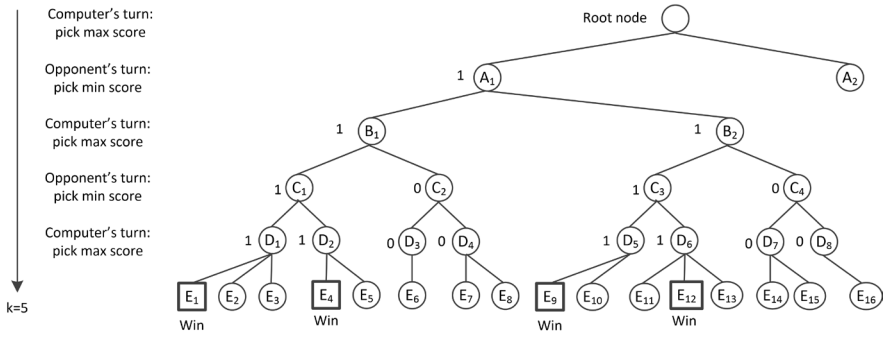


Figure 9. Illustration of level-*k* minimax search.

principle is that a operation of selecting the minimum or maximum score is applied at every node in alternate levels backwards from level-*k*. The score of a node at level-4 is the maximum of the scores of its child nodes at level-5 because it is the computer's turn to move. The score of a node at level-3 is the minimum of the scores of its child nodes at level-4 because it is the opponent's turn to move, and so on. In **Figure 9** A_1 is a decision move that leads to a level-5 sudden win.

Proposition 2. Any level-*k* sudden death or win is detectable with the level-*k* minimax search.

Proof. We focus on sudden win in the proof. The case of sudden death can be proved similarly.

For a level-*k* sudden win to exist, one of the child nodes of the root node (e.g., A_1 in **Figure 9**), say node $N_{1,1}$ at level-1, must be a decisive move that leads to a definite win in level-*k*. Because it is the opponent's turn to move at level-1, *all* of the child nodes of $N_{1,1}$ at level-2 (e.g., B_1, B_2 in **Figure 9**), say node $N_{2,1}, N_{2,2}, \cdots$, must each lead to a definite win. For this to happen, *at least one of* the child nodes of $N_{2,1}$ at level-3 (e.g., C_1 in **Figure 9**), say node $N_{3,1}$, must lead to a definite win so that the opponent has no valid move to avoid the computer's win, and *at least one of* the child nodes of $N_{2,2}$ at level-3 (e.g., C_3 in **Figure 9**), say node $N_{3,2}$, must lead to a definite win, and so on. The above logic applies to each of $N_{3,1}, N_{3,2}, \cdots$. For example, all of the child nodes of $N_{3,1}$ (e.g., D_1, D_2 in **Figure 9**) must each lead to a definite win, and at least one child node of each of these nodes (e.g., E_1, E_4 in **Figure 9**) must lead to a definite win. The process repeats until level-*k* is reached.

At level-*k*, the score of a node is $-1, 0, 1$ depending on the state of the node (loss, draw/undetermined, or win). From the process described above, for $N_{1,1}$ to be a decisive move, a set of level-$(k-1)$ nodes under $N_{1,1}$ must each have at least one child node at level-*k* being a win, and thus obtain a score of 1 themselves according to the minimax principle, which in turn makes their parent nodes at level-$(k-2)$ to have a score of 1. Their scores are further backpropagated upwards so that eventually the scores of $N_{3,1}, N_{3,2}, \cdots$ are all 1 and those of $N_{2,1}, N_{2,2}, \cdots$ are 1 as well, which results in a score of 1 for $N_{1,1}$ indicating that a level-*k* sudden win exists and $N_{1,1}$ is a decisive move leading to the sud-

den win. □

Recall from Section 3.2 that MCTS selects the move on the basis of the average scores of the child nodes of the root node (10). In general, the average scores of the child nodes of the root node are rarely equal to the scores of those nodes in a level-k minimax search precisely. However, prediction distortion is not severe if the discrepancy between the two scores does not make MCTS to select a different child node from what the level-k minimax search would select. It should be pointed out that the level-k minimax search is not necessarily superior to MCTS in all game positions. However, in the presence of sudden death or win, if MCTS selects a move different from the choice of the level-k minimax search, then by definition MCTS fails to make a decisive move. Hence we have the following proposition.

Proposition 3. MCTS fails to detect a level-k sudden death or win if the average scores (10) of the child nodes of the root node are sufficiently deviated from their scores in the level-k minimax search that a decisive move is not selected in MCTS.

Because of computational resource constraints, the search tree in MCTS is a subset of the entire state space. Random moves are applied in the simulation outside the boundary of the search tree, therefore potentially causing prediction distortion as learned in the RPS analysis. Figure 10 illustrates two examples. In (a), it is the opponent's turn to move in nodes A and B. Simulations through node A backpropagate higher scores than through node B, making the path through A deceptively more promising, although in reality A is a loss position and B is a draw. If the opponent randomly selects its move, then on average A is better than B. However, if the opponent is intelligent, it selects the move that results in the computer's immediate loss. In this example, A is a trap that results in computer's sudden death. In (b), it is the computer's turn to move in nodes A and B. Simulations through node A backpropagate lower scores than through node B, making the path through A deceptively less favorable, although in reality A is a win position and B is a draw. If the computer randomly selects its move, then on average A is worse than B. However, as the game progresses toward A,

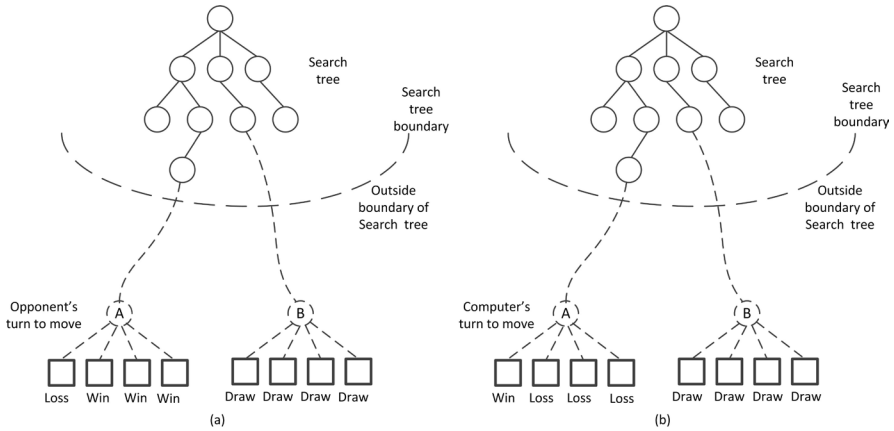


Figure 10. Illustration of sudden death (a) and sudden win (b).

the computer will eventually select the move that results in the computer's immediate win. This is an example of sudden win for the computer. Furthermore, even for the nodes within the search tree boundary, their scores may not converge to the minimax scores if not enough simulation iterations have been taken.

Clearly as the level increases, it becomes more difficult to discover sudden death/win. In our implementation of MCTS playing Gomoku, we find that MCTS frequently discovers level-1 and level-3 sudden win and level-2 sudden death, but often unable to discovery level-4 sudden death. We will analyze the simulation performance of MCTS next.

4.2. Simulation Results

We have simulated the MCTS algorithm in Java to play Gomoku on an $K \times K$ board. The implementation details are described in **Appendix A**. We report the simulation results in this section.

4.2.1. Methodology

We do not model the computational resource constraint in terms of precise memory or runtime. Instead we vary the number of iterations T. We find that the run time increases roughly linearly with T and that the Java program runs into heap memory errors when T exceeds some threshold. For example, for $K = 9$, the run time is between 7 to 10 seconds for every move at $T = 30000$, and a further increase to $T = 40000$ causes the out of memory error. In contrast, for $K = 11$, the Java program can only run $T = 10000$ before hitting the memory error. Therefore, we use parameter T to represent the consumed computational resource, and K to represent the complexity of the game.

We are interested in the performance of MCTS in dealing with sudden death/win. To report the win rate of an entire game, it would depend on the strength of the opponent, a quantity difficult to measure consistently. Therefore, we let a human opponent to play the game long enough to encounter various sudden death/win scenarios and record the fraction of times that the program detects sudden death/win. Detecting sudden death means that MCTS anticipates the threat of an upcoming sudden death and takes a move to avoid it. Detecting sudden win means that MCTS anticipates the opportunity of an upcoming sudden win and takes a move to exploit it and win the game.

4.2.2. Result Summary

The simulation results are summarized in Table 1[1]. We make the following observations.

- Detecting level-4 sudden death is more difficult than detecting level-2 sudden death. The detection rate drops drastically as the game complexity, represented by K, increases. On the other hand, the detection rate drops drastically as the computational resource, represented by T, decrease.

[1]For $K = 11$, $T = 10000$ instead of 30,000.

Table 1. Detection rate of sudden death/win of MCTS.

	Sudden Death			
	Level-2		Level-4	
	$T = 30{,}000$	$T = 3000$	$T = 30{,}000$	$T = 3000$
$K = 7$	100%	100%	80%	60%
$K = 9$	100%	40%	50%	10%
$K = 11$	60%	<15%	15%	<5%
	Sudden Win			
	Level-1		Level-3	
	$T = 30{,}000$	$T = 3000$	$T = 30{,}000$	$T = 3000$
$K = 7$	100%	100%	100%	100%
$K = 9$	100%	100%	100%	90%
$K = 11$	100%	100%	80%	50%

- There seems to be a tipping point beyond which the detection rate of sudden death is so low that MCTS is useless. For example, for $T = 30{,}000$, the tipping point is around $K = 7$ to 9.
- The detection rate is not symmetric between sudden death and sudden win. Indeed, it seems much easier to detect sudden win than to detect sudden death. In particular, MCTS is always able to detect level-1 sudden win. Moreover, at a first glance, one may expect the detection rate of level-2 sudden death to be higher than that of level-3 sudden win, because level-3 sudden win involves more steps to search through. However, it turns out that detecting level-3 sudden win is more robust than detecting level-2 sudden death in a variety of K, T scenarios. We will further investigate this phenomenon by looking deeper into the scores in MCTS iterations.
- Not shown in Table 1, we observe that when T is small, e.g., $T = 3000$, and K is large, e.g., $K = 9$, sometimes MCTS takes seemingly random positions when there are no sudden death/win moves. Furthermore, when $K = 11$, MCTS often miss the opportunity of seizing level-3 win positions; however, conditional on it already getting into a level-3 win position, MCTS has high probability to exploit the level-3 win opportunity. Therefore, given the computational resource constraint, it is challenging for our Java implementation of MCTS to handle the complexity of a 11×11 or larger board.

4.2.3. Level-4 Sudden Death

To investigate the reason that MCTS fails to detect level-4 sudden death, we next examine a specific scenario shown in Figure 11 with $K = 9$. Shown on the left side of the figure is the current game position, a level-4 sudden death situation. It is now the computer's turn to move. The correct move should be either B or C. However, after examining all available positions on the board with $T = 30{,}000$, MCTS decides to take position A.

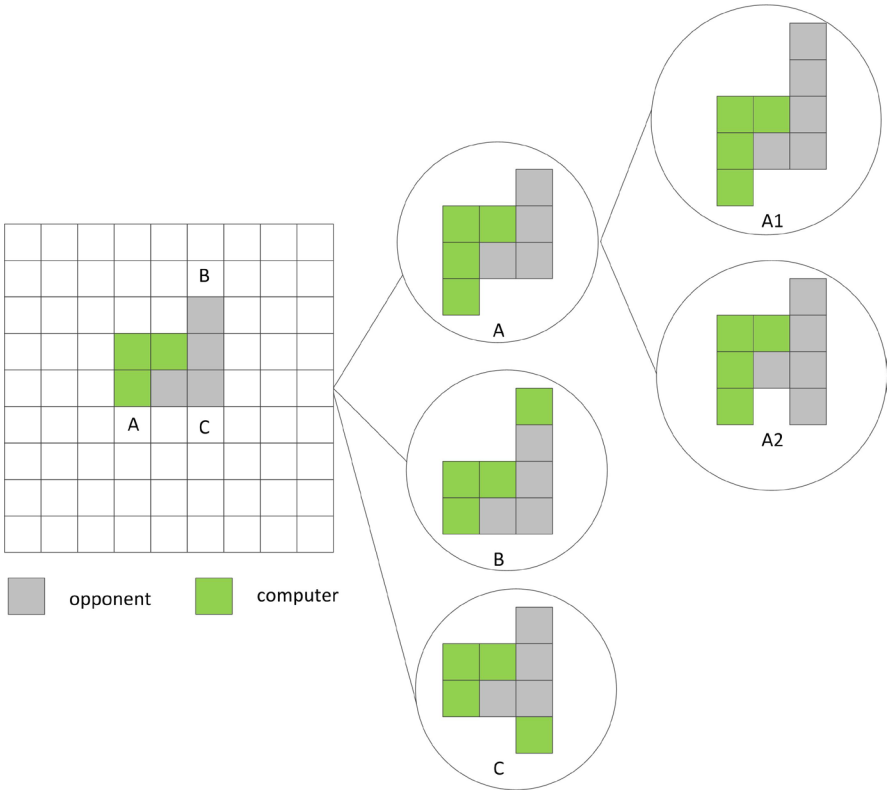


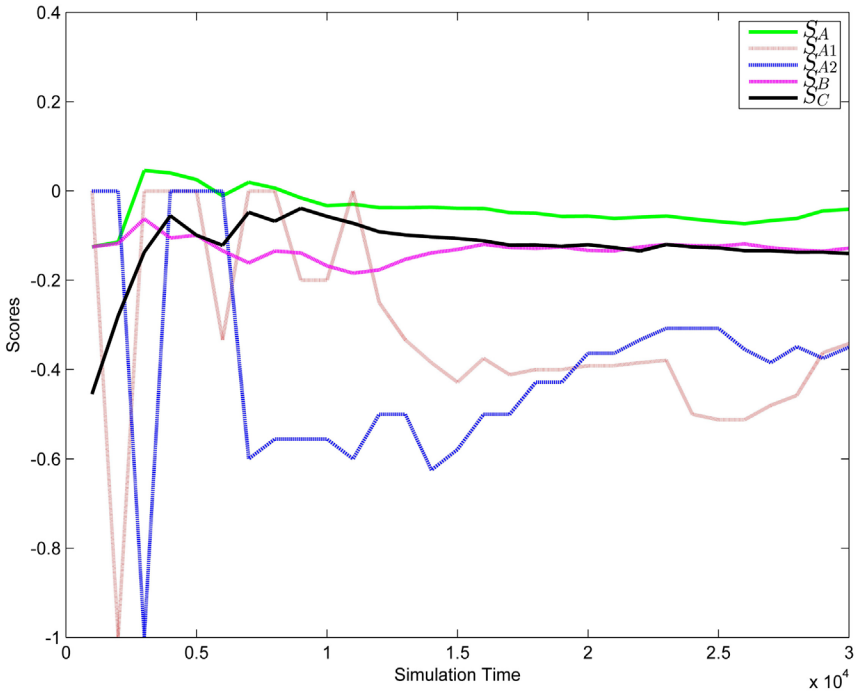
Figure 11. An example of level-4 sudden death in Gomoku.

The moves A, B, C correspond to three child nodes of the root node, represented by three circles labeled as A, B, C in **Figure 11**. To understand why MCTS chooses node A over node B or C, we plot the MCTS statistics of the three nodes in **Figure 12**. We observe that node A achieves a higher average score than node B or C does and is visited much more frequently.

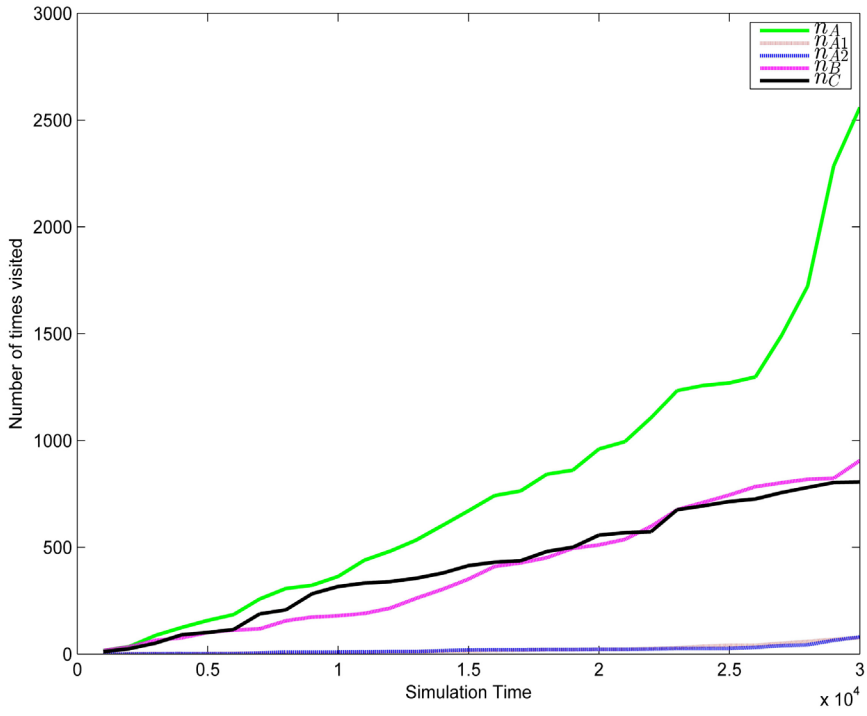
This observation may be puzzling, because position A leads to level-3 sudden win, after the computer takes A. So, wouldn't the opponent take B or C therefore resulting in computer's loss? These two possible moves are represented by two child nodes of node A, labeled as $A1$, $A2$ in **Figure 11**. The statistics of nodes $A1$, $A2$ are also plotted in **Figure 12**. We note that the average scores of nodes $A1$, $A2$ are indeed quite negative, indicating that $A1$, $A2$ likely result in computer's loss as expected.

To understand why the low, negative scores of $A1$, $A2$ fail to bring down the score of node A, we plot in **Figure 13** the MCTS statistics of all the child nodes of node A when the simulation stops at $T = 30{,}000$. Among all the child nodes, node $A1$ is of index 15 and node $A2$ of index 43. We note that MCTS indeed visits $A1$, $A2$ much more frequently than other child nodes of node A and that the total scores of $A1$ and $A2$ are quite low, both being expected from the exploitation principle of the UCT algorithm (9).

From the backpropagation step in Section 3.2, we know that a node's total score is the sum of the total scores of its child nodes. Therefore, we predict that if more computational resource were available (e.g., with greater numbers of



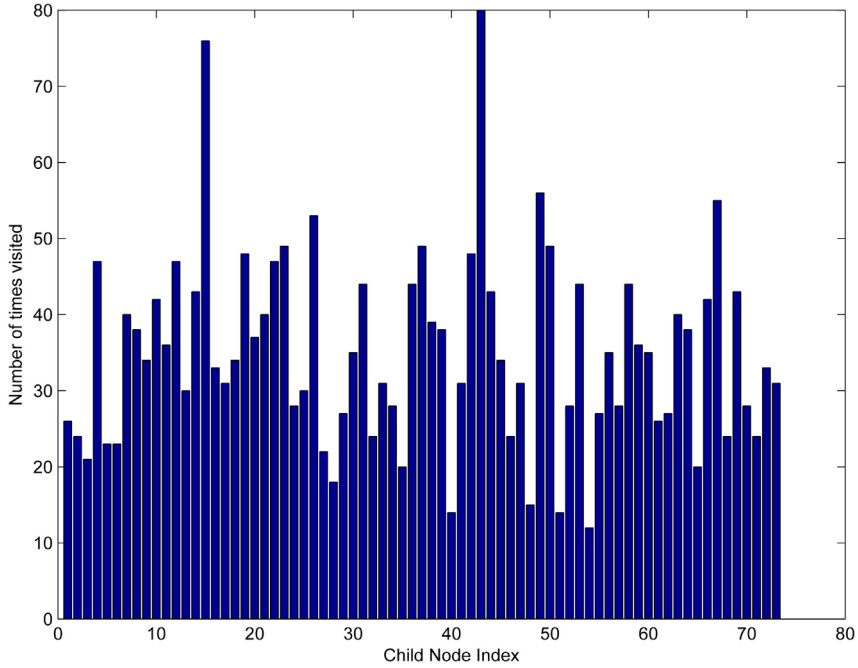
(a)



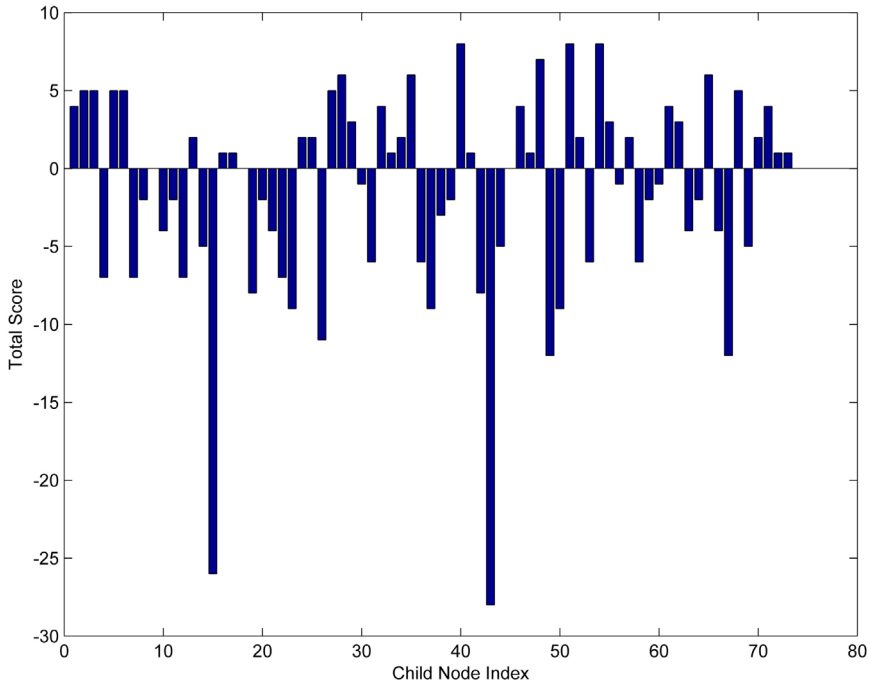
(b)

Figure 12. Average scores and numbers of visits of nodes *A*, *B*, *C*, *A*1, *A*2 of **Figure 11** as the MCTS simulation progresses.

MCTS simulation), the scores of *A*1 and *A*2 would eventually dominate the score of their parent node *A* so that it would be lower than the score of node *B* or *C* at which point MCTS would correctly select position *B* or *C* and avoid



(a)



(b)

Figure 13. Total scores and numbers of visits of all the child nodes of node *A* of **Figure 11** at the end of simulation.

sudden death. In other words, the practical limitation of MCTS in dealing with sudden death is caused by the computational resource. In a way, this is somewhat expected when we note in Section 2.3 that convergence is quite slow even for a simple game position (Tic-Tac-Toe) and a simple algorithm (RPS).

4.2.4. Level-2 Sudden Death versus Level-3 Sudden Win

The example scenario in Figure 14 with $K = 11$ shows how MCTS fails to detect level-2 sudden death while pursuing level-3 sudden win. Shown on the left side of the figure is the current game position, a level-2 sudden death situation. It is now the computer's turn to move. The correct move should be B to avoid level-2 sudden death. However, after examining all available positions on the board with $T = 10{,}000$, MCTS decides to take position A to pursue level-3 sudden win.

The moves A, B correspond to two child nodes of the root node, represented by two circles labeled as A, B in Figure 14. Figure 15 plots the MCTS statistics of the two nodes. We observe that node A achieves a higher average score than node B does and is visited much more frequently.

Figure 16 plots the MCTS statistics of all the child nodes of node A when the simulation stops at $T = 10{,}000$. Among all the child nodes, node $A1$ is of index 56. We note that MCTS indeed visits $A1$ more frequently than any other child node and the total score of $A1$ is the lowest, similar to what we have observed in Figure 13. However, the difference between the child nodes is not as significant. The reason is probably that K is larger and T is smaller in this example and as a result MCTS is still biased towards exploration and exploitation has not been in effect.

The difference of the statistics of nodes A and B in Figure 15, however, is quite minor. In fact, we note that towards the end of the simulation, the average score of B increases while that of A decreases and MCTS visits node B increasingly more often, indicating that if more computational resource were available to allow T to go further beyond 10,000, then node B would probably overtake node A as the winner and MCTS would find the correct move and overcome level-2 sudden death.

5. An Improved MCTS Algorithm to Address Sudden Death/Win

The issue of sudden death/win is a known problem for MCTS [11] [12]. In [13], the authors propose that if there is a move that leads to an immediate win or

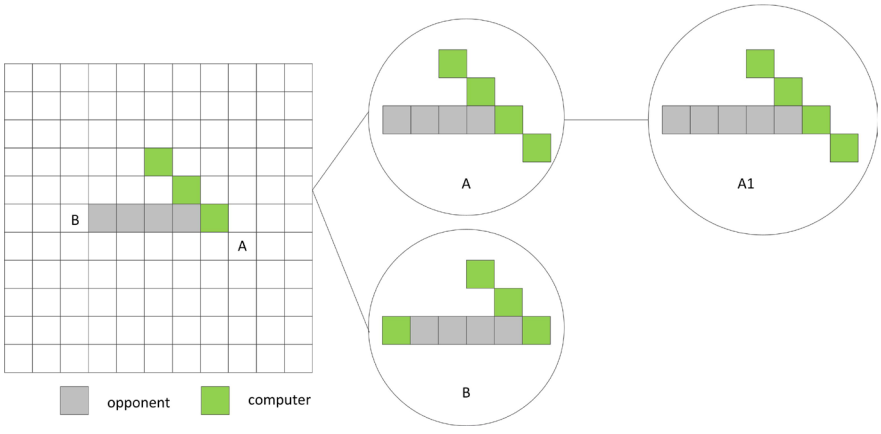
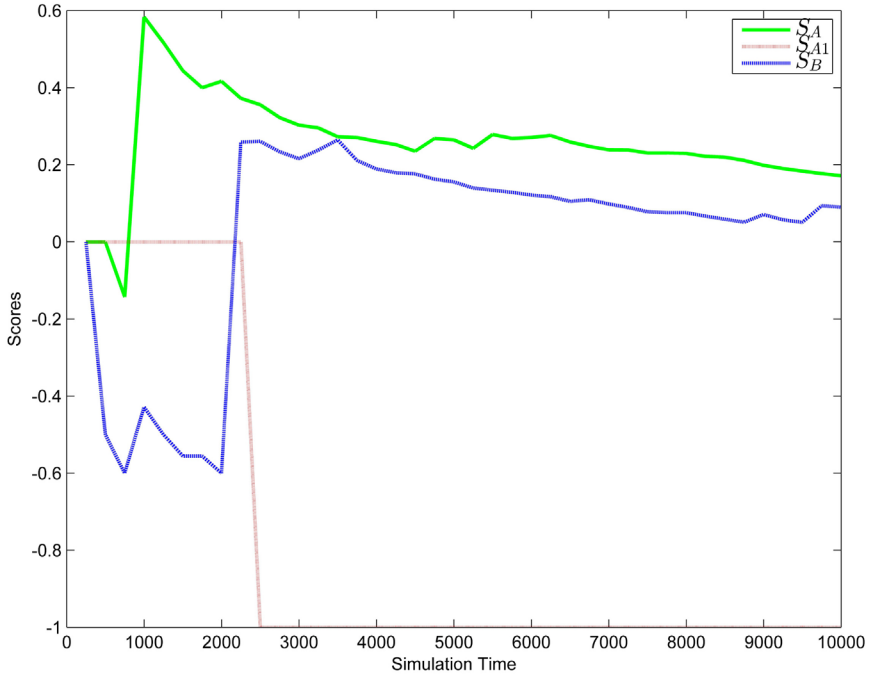
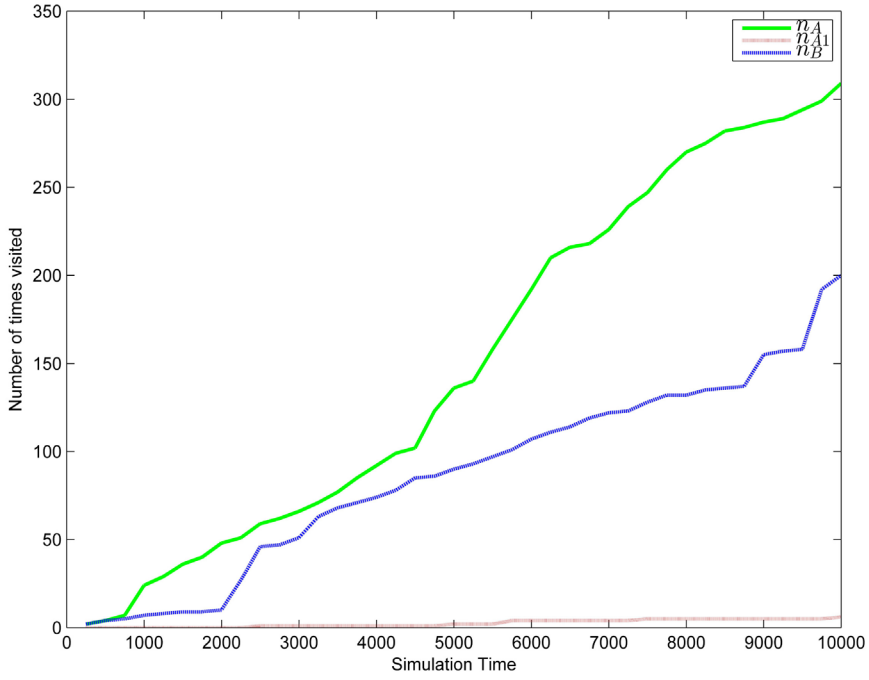


Figure 14. An example of level-2 sudden death in Gomoku.



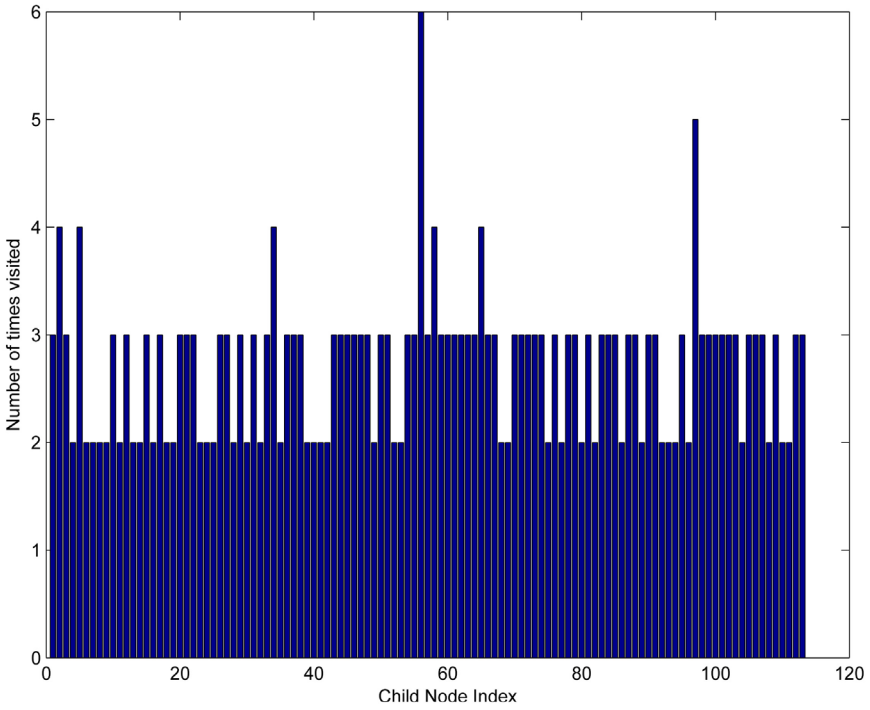
(a)



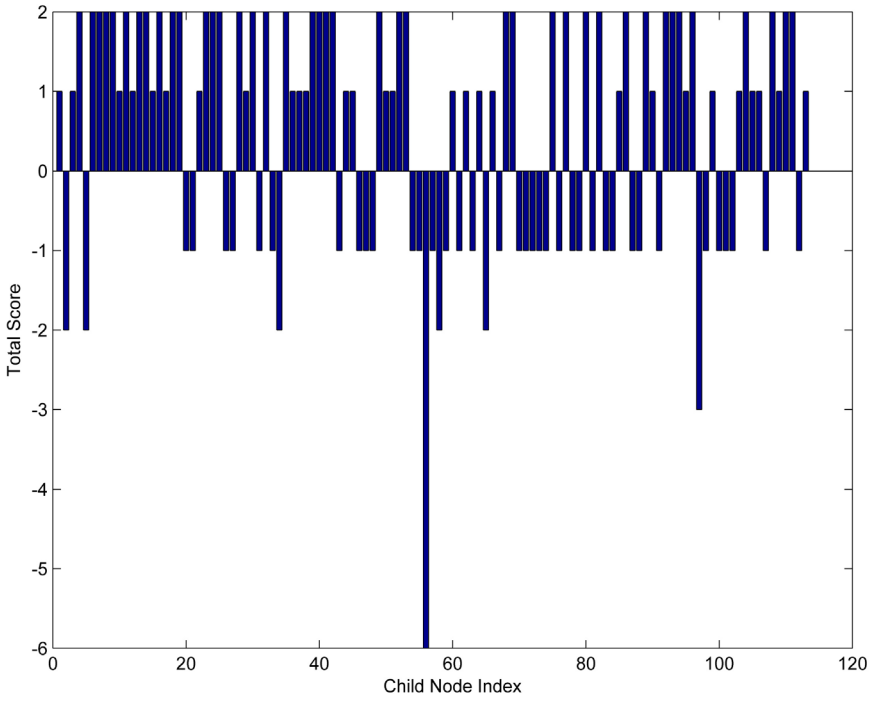
(b)

Figure 15. Average scores and numbers of visits of nodes *A*, *B*, *A*1 of **Figure 14** as the MCTS simulation progresses.

loss, then play this move; otherwise, play MCTS. Clearly this algorithm only addresses level-1 sudden win or level-2 sudden problems. As mentioned before, level-2 sudden death or level-1 win is not a major issue in MCTS at least for playing Gomoku with reasonable *K*, *T*. While it is possible that with sufficient



(a)



(b)

Figure 16. Total scores and numbers of visits of all the child nodes of node *A* of **Figure 14** at the end of simulation.

computational resource MCTS could eventually deal with sudden death/win, next we propose an improved MCTS algorithm that addresses this problem without increasing drastically required computational resource.

5.1. Basic Idea

To identify a level-k sudden death/win, we have to use a minimax search of depth k starting from the root node [11]. One brute-force algorithm, as shown in Figure 17(a), is that when the computer enters a new root node, it first runs the minimax search; if there is a sudden death/win within k steps, then follows the minimax strategy to take sudden win or avoid sudden death; otherwise, proceed with MCTS.

However, the computational costs of the minimax search would be significant even for a modest k. Carrying out the minimax search for every new root node reduces the computational resource left for MCTS. The key idea of our proposal is to not burden MCTS with the minimax search until it becomes necessary.

Figure 17(b) shows the flow chart of the improved MCTS algorithm. When the computer enters a new root node, it carries out MCTS. At the end of the simulation step of every MCTS iteration, if the number of steps from the root node to the terminal position exceeds k, then continue to the next MCTS

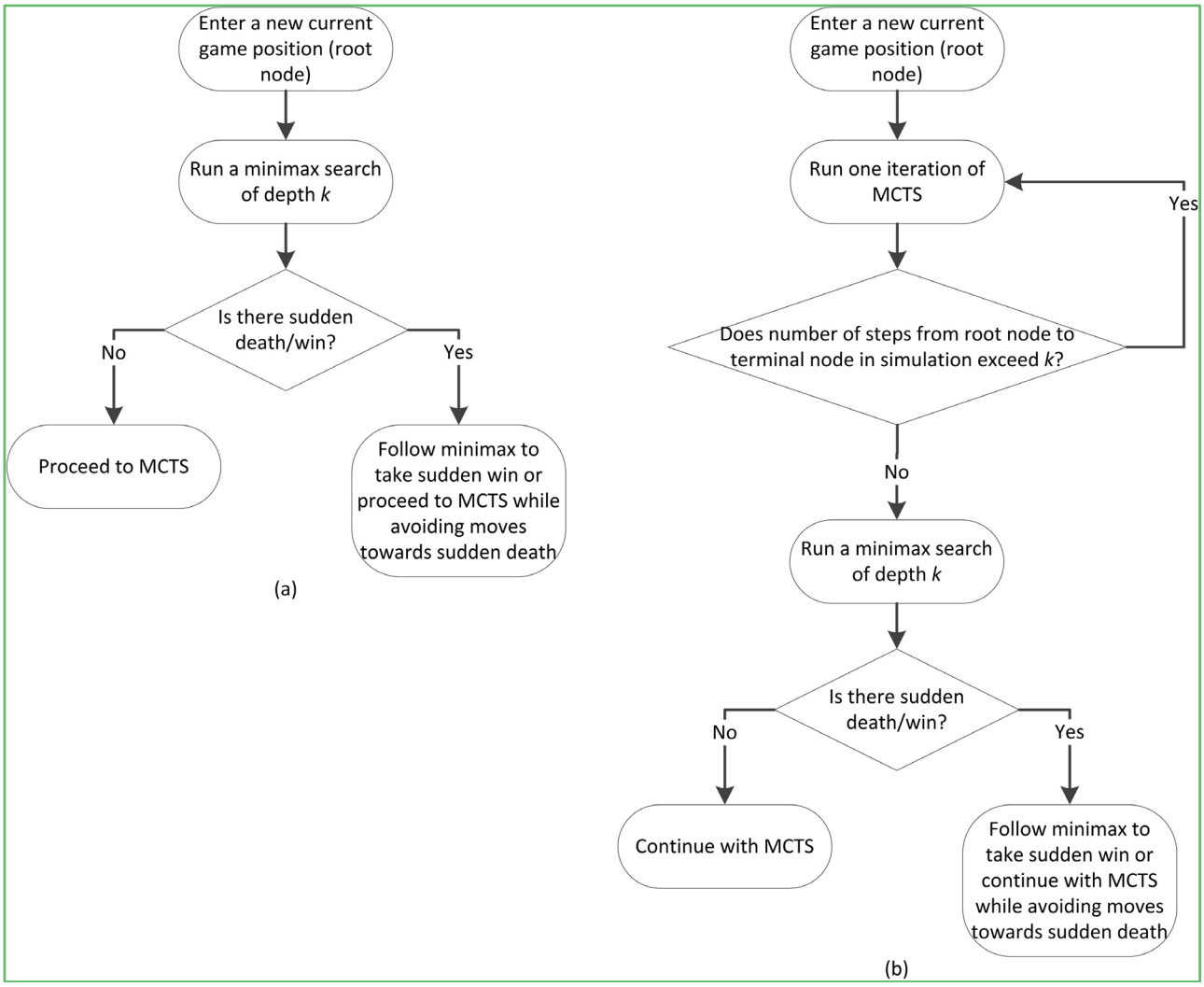


Figure 17. Flow chart of the improved MCTS algorithm: (a) Brute-force algorithm; and (b) Proposed algorithm.

iteration; otherwise, run the minimax search from the root node and follow the algorithm of Figure 17(a). In our implementation, we run a single minimax search of depth k to cover both level-k sudden death and level-$(k-1)$ sudden win for even k.

From Proposition 2 the brute-force algorithm is guaranteed to detect any level-k sudden death or win. However, the improved MCTS algorithm can only guarantee the detectability with probability.

Proposition 4. Any level-k sudden death or win is detectable, with probability, with the improved MCTS algorithm in Figure 17(b), where the probability depends on the number of MCTS iterations, the complexity of the game, and the value of k.

The reason of "guarantee *with probability*" in Proposition 4 is that the improved MCTS algorithm needs to encounter a terminal node within level-k in at least one of the many iterations before a minimax search is triggered. In Gomoku, we find that the probability is close to 1 when the number of iterations is large, the board size is modest, and depth k is a small number, e.g., $T = 30{,}000$, $K = 9$, $k = 4$.

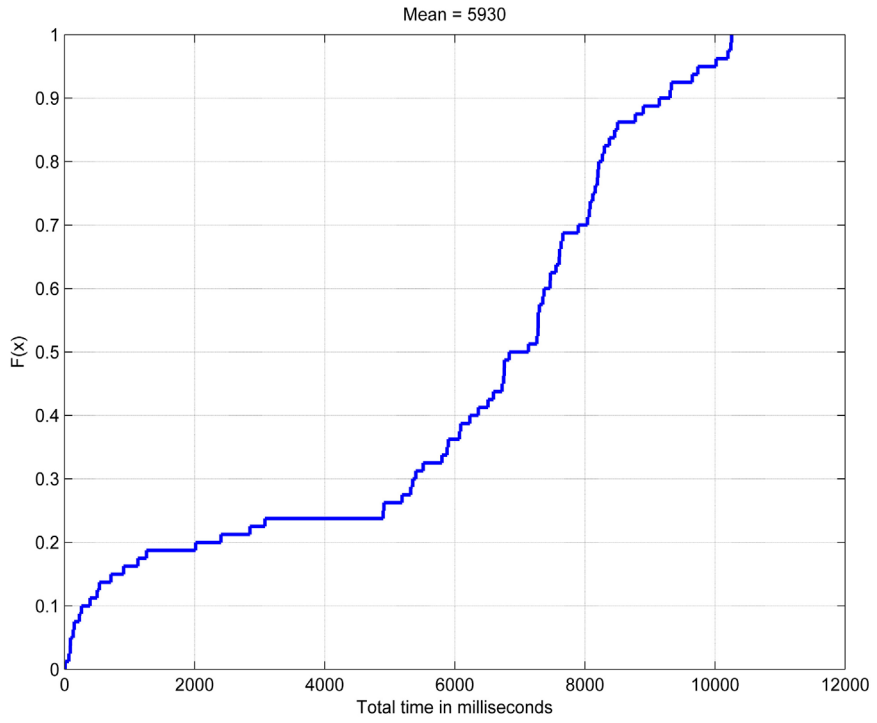
5.2. Simulation Results

We have implemented the improved MCTS algorithm in JAVA. The implementation includes a minimax search of depth 4, and we confirm that the program is able to always detect any level-2/4 sudden death and level-1/3 sudden win. To assess the computational resource used in the minimax search of depth 4 and the overall MCTS algorithm, we record the total elapsed time of every MCTS step, which includes T simulations and possibly a minimax search, as well as the elapsed time of every minimax search when it happens. Figure 18 shows that the cumulative distribution functions (CDF) of the two elapsed times for the case of $T = 30{,}000$, $K = 9$.

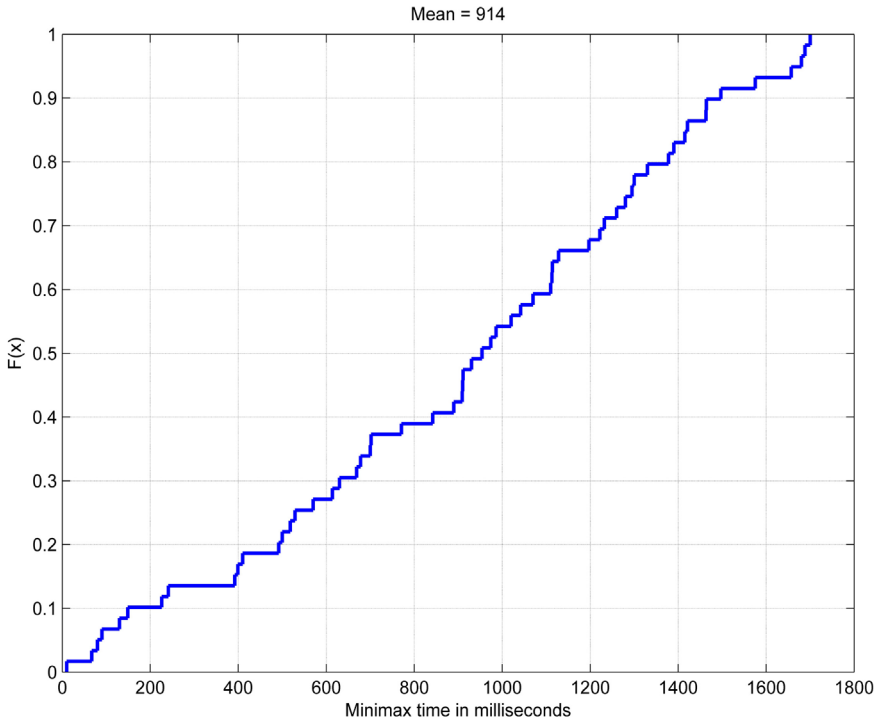
We observe that the elapsed time of a minimax search is pretty uniformly distributed from 0 to about 1.8 seconds and that the elapsed time of a MCTS step is either small (below 2 seconds) or quite large (above 6 seconds), where the small value corresponds to the scenarios where MCTS can find easy moves to win the game (level-1 or level-3 sudden win). On average, a single minimax search is about 1/6 of the entire MCTS 30,000 simulations. As the depth increases, the minimax search will be exponentially expensive. Therefore, the advantage of the proposed algorithm over the brute-force algorithm could be significant.

5.3. Heuristic Strategies to Enhance Performance

In the improved algorithm of Figure 17(b), the minimax search of depth k runs from the root node to detect the presence of sudden death or win. We next propose to employ a heuristic strategy to reduce the minimax search complexity and to enhance the MCTS effectiveness. The basic idea is that it is not necessary



(a)



(b)

Figure 18. CDF of the total elapsed time in every MCTS step and the elapsed time of a minimax search.

to search over *all* the child nodes of the root node. Instead, we only need to run the minimax search from a subset of *promising* child nodes.

Suppose that in one iteration of MCTS a terminal node of loss (or win) is encountered within depth k from the root node, indicating the potential presence of sudden death (or win). Mark the child node of the root node selected in that iteration as a potential sudden death (or win) move.

Consider the first heuristic strategy. Instead of triggering a minimax search immediately as in Figure 17(b), we continue to run MCTS for some number of iterations and then apply the following two rules.

1) For the subset of child nodes of the root node whose average scores are in the top x_1 percentile and that have been marked as a potential sudden death move, run a minimax search of depth $k-1$ from each of them. If the minimax score of a child node in the subset is equal to −1, then mark it as a sudden death move and refrain from selecting it in subsequent MCTS iterations. Continue to run MCTS.

2) For the subset of child nodes of the root node whose average scores are in the top x_2 percentile and that have been marked as a potential sudden win move, run a minimax search of depth $k-1$ from each of them. If the minimax score of any of them is equal to 1, then mark the child node as a sudden win move, which is the best move to select, and stop MCTS. Otherwise, continue to run MCTS.

Rule 1 is to prevent a sudden death move from prevailing in MCTS node selection. For example, move A in Figure 11 will be detected and discarded, and therefore will not prevail over move B or C. Rule 2 is to discover a sudden win move and to end MCTS sooner. Here x_1, x_2 are heuristic numbers that are used to trade off the reduction in minimax search with the improvement of MCTS effectiveness.

Consider the second heuristic strategy. Instead of running MCTS for some number of iterations, applying the above two rules and then continuing MCTS as in the first strategy, we can run MCTS to the end and then apply the following two similar rules.

1) For the child node of the root node with the highest average score, if it has been marked as a potential sudden death move, run a minimax search of depth $k-1$. If its minimax score is equal to −1, then discard it, proceed to the child node with the second highest average score and repeat this rule. Otherwise, proceed to the next rule.

2) For the subset of child nodes of the root node whose average scores are in the top x_2 percentile and that have been marked as potential sudden win moves, run a minimax search of depth $k-1$ from each of them. If the minimax score of any of them is equal to 1, then select the move. Otherwise, select the move according to the MCTS rule.

The performance of the above two heuristic strategy can be assessed with the following proposition.

Proposition 5. Either the first or second heuristic strategy has the same capability to detect a sudden death as the improved MCTS algorithm in Figure

17(b). Its capability to detect a sudden win improves as x_2 increases, and is in general strictly weaker than the improved MCTS algorithm, except in the special case where $x_2 = 100$ in which case it has the same sudden win detection capability.

In Gomoku, if x_2 is a small number, the complexity reduction over the improved algorithm of Figure 17(b) is in the order of $O\left(K^2\right)$, because the size of the subset of the child nodes from which a minimax search actually runs is roughly $1/K^2$ of the size of the total set. Even for a modest board size K, e.g., $K = 9$, the reduction is significant.

6. Conclusions

In this paper we analyze the MCTS algorithm by playing two games, Tic-Tac-Toe and Gomoku. Our study starts with a simple version of MCTS, called RPS. We find that the random playout search is fundamentally different from the optimal minimax search and, as a result, RPS may fail to discover the correct moves even in a very simple game of Tic-Tac-Toe. Both the probability analysis and simulation have confirmed our discovery.

We continue our studies with the full version of MCTS to play Gomoku. We find that while MCTS has shown great success in playing more sophisticated games like Go, it is not effective to address the problem of sudden death/win, which ironically does not often appear in Go, but is quite common on simple games like Tic-Tac-Toe and Gomoku. The main reason that MCTS fails to detect sudden death/win lies in the random playout search nature of MCTS. Therefore, although MCTS in theory converges to the optimal minimax search, with computational resource constraints in reality, MCTS has to rely on RPS as an important step in its simulation search step, therefore suffering from the same fundamental problem as RPS and not necessarily always being a winning strategy.

Our simulation studies use the board size and number of simulations to represent the game complexity and computational resource respectively. By examining the detailed statistics of the scores in MCTS, we investigate a variety of scenarios where MCTS fails to detect level-2 and level-4 sudden death. Finally, we have proposed an improved MCTS algorithm by incorporating minimax search to address the problem of sudden death/win. Our simulation has confirmed the effectiveness of the proposed algorithm. We provide an estimate of the additional computational costs of this new algorithm to detect sudden death/win and present two heuristic strategies to reduce the minimax search complexity and to enhance the MCTS effectiveness.

The importance of sudden death/win is not limited to the board games. The real-world AI applications such as autonomous driving sometimes face similar situations where certain actions lead to drastic consequences. Therefore, we hope to extend the research of the MCTS algorithm to these applications in the future study.

References

[1] Schaeffer, J., Müller, M. and Kishimoto, A. (2014) Go-Bot, Go. *IEEE Spectrum*, **51**, 48-53. https://doi.org/10.1109/MSPEC.2014.6840803

[2] https://en.wikipedia.org/wiki/Computer_Go

[3] Nandy, A. and Biswas, M. (2018) Google's DeepMind and the Future of Reinforcement Learning. In: *Reinforcement Learning*, Apress, Berkeley, 155-163. https://doi.org/10.1007/978-1-4842-3285-9_6

[4] Silver, D., *et al.* (2017) Mastering the Game of Go without Human Knowledge. *Nature*, **550**, 354-359. https://doi.org/10.1038/nature24270

[5] Coulom, R. (2006) Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: van den Herik, H.J., Ciancarini, P. and Donkers, H.H.L.M., Eds., *Computers and Games. CG* 2006. *Lecture Notes in Computer Science*, vol 4630, Springer, Berlin, Heidelberg, 72-83.

[6] Kocsis, L. and Szepesvári, C. (2006) Bandit Based Monte-Carlo Planning. In: Fürnkranz, J., Scheffer, T. and Spiliopoulou, M., Eds., *Machine Learning: ECML* 2006. *ECML* 2006. *Lecture Notes in Computer Science*, vol 4212, Springer, Berlin, Heidelberg, 282-293. https://doi.org/10.1007/11871842_29

[7] Chaslot, G.M.J-B., Saito, J.-T., Bouzy, B., Uiterwijk, J.W.H.M. and van den Herik, H.J. (2006) Monte-Carlo Strategies for Computer Go. Proceedings of the 18*th BeNeLux Conference on Artificial Intelligence*, Belgium, 2006, 83-90.

[8] Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samthrakis, S. and Colton, S. (2012) A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, **4**, 1-49. https://doi.org/10.1109/TCIAIG.2012.2186810

[9] Auer, P., Cesa-Bianchi, N. and Fischer, P. (2002) Finite-Time Analysis of the Multi-Armed Bandit Problem. *Machine Learning*, **47**, 235-256. https://doi.org/10.1023/A:1013689704352

[10] Chang, H., Fu, M., Hu, J. and Marcus, S. (2005) An Adaptive Sampling Algorithm for Solving Markov Decision Processes. *Operations Research*, **53**, 126-139. https://doi.org/10.1287/opre.1040.0145

[11] Ramanujan, R., Sabharval, A. and Selman, B. (2010) On Adversarial Search Spaces and Sampling-Based Planning. Proceedings of the 20*th International Conference on Automated Planning and Scheduling*, Toronto, 12-16 May 2010, 242-245.

[12] Browne, C. (2011) The Dangers of Random Playouts. *ICGA Journal*, **34**, 25-26. https://doi.org/10.3233/ICG-2011-34105

[13] Teytaud, F. and Teytaud, O. (2010) On the Huge Benefit of Decisive Moves in Monte-Carlo Tree Search Algorithms. *Proceedings of IEEE Conference on Computational Intelligence and Games*, Copenhagen, 8-21 August 2010, 359-364. https://doi.org/10.1109/ITW.2010.5593334

Appendix A

A. Java Implementation

The following appendix sections describe the Java implementation of the analysis and simulation codes used in this study.

A1. Tic-Tac-Toe and Gomoku Game GUI

I downloaded a Java program from the Internet, which provides a graphic user interface (GUI) to play Tic-Tac-Toe. I modified the program to play Gomoku. Figure A1 shows a screenshot of the game GUI generated by the program.

The game GUI consists of two panels. The left panel shows two options to play the game: person-to-person or person-to-computer. In this study I am only interested in the latter option. The two colors, black (person) and green (computer), show whose turn it is to move. The right panel is a $K \times K$ grid to represent the game board.

The program uses an array currentTable[] of length K^2 to represent the current game position, where element currentTable[i*K+j] is the state of row i and column j of the game board. currentTable[i*K+j] can be in one of three states:

enum TILESTATUS {EMPTY, PERSON, COMPUTER}.

Initially, all elements are EMPTY. When it is the person's turn to move and the human player clicks on one of the empty grid element, the program changes the corresponding element of currentTable[] to PERSON. When it is the computer's turn to move, the program calls a function.

int choseTile()

which returns an integer index pointing to one of the elements of currentTable[] representing the computer's move. The program changes the corresponding

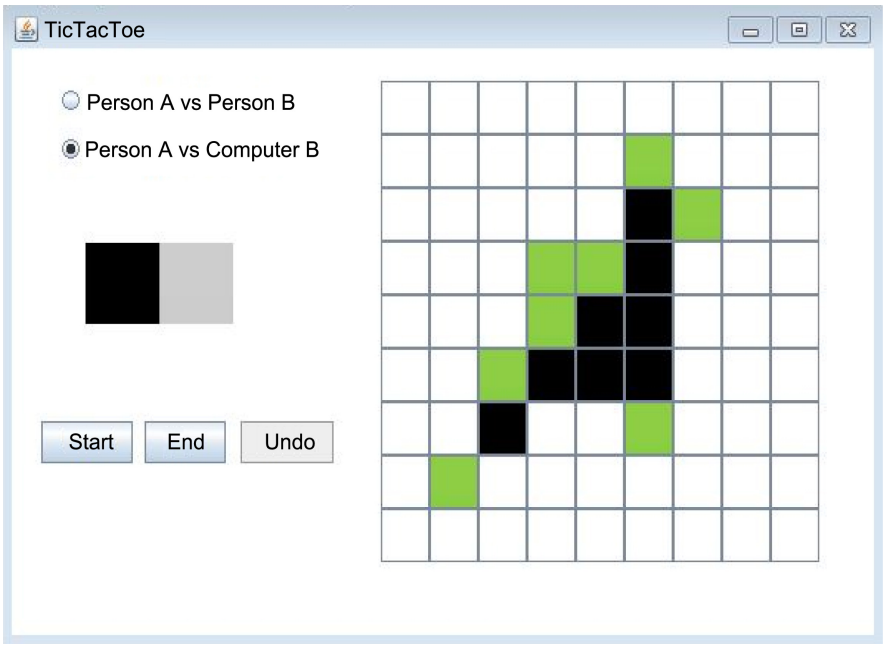


Figure A1. A screenshot of Gomoku game environment.

element of currentTable[] to COMPUTER. After every move, the program calls a function

boolean checkWinner(TILESTATUS[] currentTable, int select)

to check whether the game position currentTable has a winner where select is the latest move.

Function choseTile() is where any AI algorithm is implemented. In the Tic-Tac-Toe Java program downloaded from the Internet, the default implementation is to randomly return any index that corresponds to an EMPTY element. In this study, I have implemented the RPS and MCTS algorithms in choseTile(). I will describe our implementation of RPS and MCTS later.

A2. Implementation of Probability Analysis of RPS

I wrote a Java program to calculate the expected score of each move from the current game position of RPS. The Java program is used in the probability analysis in Section 2.2.

The calculation is based on the recursive Equation (3). The current game position is represented by the root node and all the legal next-step moves from the current game position by the computer form its immediately child nodes. The Java problem uses a recursive function

float scoreIteration(TURN turn, int select)

to calculate the expected score of the child node presented by currentTable. Denote K the number of EMPTY elements in the root node and index[k] the k-th EMPTY element. The main() function calls scoreIteration() for each of the K child nodes of the root node, k, with turn set to COMPUTER, select set to index[k], and currentTable[index[k]] set to COMPUTER. currentTable[index[k]] is reset to EMPTY after each function call.

The programming idea of scoreIteration() is that if the node is terminal, then return −1, 0, 1 depending on the game outcome; otherwise, expand to its child nodes and call scoreIteration() recursively. When a node expands to its next level child nodes, the number of EMPTY elements decrements by one. The pseudo of scoreIteration() is given as follows.

A3. MCTS and RPS Implementation

I downloaded a Java program from the Internet (https://code.google.com/archive/p/uct-for-games/source), which implements the MCTS algorithm. I modified the MCTS program to work with the GUI program, to output various statistics to study the simulation results in Sections 2.3 and 4.2, and to allow flexible alrogithm selection, which will be necessary to implement the improved MCTS algorithm of Section 5.

The MCTS Java program defines a class called MCTSNode, which includes a number of variables and methods to implement the algorithm described in Section 3.2. The most important variables are

int totalScore to store the total score of the node,

int timesVisited to store the total number of times the node is visited,

Algorithm 1 Probability Calculation of RPS

```
1: function float scoreIteration(turn, select)
2:     if checkwin(currentTable, select) == true then
3:         if turn is person's turn then                          ▷ Computer wins
4:             return 1
5:         else                                                   ▷ Person wins
6:             return -1
7:     else
8:         numEMPTY ← number of EMPTY elements
9:         nextTurn ← opposite of turn                            ▷ Toggle the turn
10:        if numEMPTY==0 then                                    ▷ No EMPTY elements left
11:            return 0                                           ▷ Draw
12:        else                                                   ▷ Expand to next level
13:            score ← 0
14:            for i ← 0 to numEMPTY do
15:                index ← i-th EMPTY element
16:                currentTable[index] ← turn                     ▷ Mark a tentative move
17:                score ← score+scoreIteration(nextTurn,index)/numEMPTY
                                                                  ▷ Recursion
18:                currentTable[index] ← EMPTY                    ▷ Restore currentTable
19:            return score
```

ArrayList<MCTSNode> nextMoves to store an array list pointing to the child nodes of the node,

int indexFromParentNode to store the move that its parent node takes to reach the node,

TILESTATUS[] nodeGameState to store the game position of the node,

TILESTATUS nodeTurn to store whose turn it is to move.

The average score of the node is simply totalScore/timesVisited. The most important methods are described below.

Function choseTile() calls bestMCTSMove(), which returns the move chosen by MCTS.

Algorithm 2 Main Loop

```
1: function int bestMCTSMove()
2:     Construct root node curNode from currentTable
3:     for i ← 1 to T do                          ▷ T is the number simulation iterations
4:         curNode.runTrial()
5:     return the child node of curNode with highest average score
```

bestMCTSMove() in turn calls runTrial() in each MCTS iteration, in which the game ends in one of three outcomes

enum TURN {PERSON, COMPUTER, DRAW}.

runTrial() calls simulateFrom() to simulate a random playout beyond the tree boundary.

The RPS algorithm is a simpler version of MCTS. One variable is added to the MCTSNode class:

int levelFromTopNode to store the number of levels from the root node to the node. For a child node of the root node, levelFromTopNode is set to 1.

Algorithm 3 Each MCTS iteration

```
1: function TURN runTrial()
2:     if checkwin(nodeGameState, indexFromParentNode) == true then
3:         rolloutResult ← game result: PERSON or COMPUTER
4:     else
5:         if Node is a leaf node then
6:             Construct and add all child nodes to nextMoves                ▷ Expansion
7:         if timesVisited == 0 OR node is a leaf node then
8:             rolloutResult ← simulateFrom(nodeGameState, nodeTurn, indexFromParentNode)  ▷ Simulation
9:         else
10:            nextNode ← the child node with highest UCB score given in (9)
                                  ▷ UCT search among child nodes in nextMoves
11:            rolloutResult ← nextNode.runTrial()                           ▷ Recursion
12:    timesVisited ← timesVisited+1                                         ▷ Backpropagation
13:    if rolloutResult == PERSON then
14:        totalScore ← totalScore-1                                         ▷ Backpropagation
15:    else if rolloutResult == COMPUTER then
16:        totalScore ← totalScore+1                                         ▷ Backpropagation
17:    return rolloutResult
```

Algorithm 4 Simulated Random Playout

```
1: function TURN simulateFrom(TILESTATUS[] state, TURN myTurn, int previousMove)
2:     if checkwin(state, previousMove) == true then
3:         return game result: PERSON or COMPUTER
4:     else
5:         if There is no EMPTY element then                                 ▷ Game board is full
6:             return DRAW
7:         else
8:             Randomly pick any EMPTY element index                         ▷ Random selection
9:             state[index] ← myTurn                                         ▷ Mark the simulated move
10:            nextTurn ← opposite of myTurn                                 ▷ Toggle the turn
11:            return simulateFrom(state, nextTurn, index)                   ▷ Recursion
```

The only change to implement RPS is that line 5 of function runTrial() is replaced by

if Node is a leaf node AND levelFromTopNode == 0 then

so that only the root node is expanded to add its child nodes, which are never expanded and are kept as the leaf nodes.

A4. Improved MCTS Implementation

I revised the MCTS Java program to implement the improved MCTS algorithm of Figure 17(b). In addition to levelFromTopNode, two variables are added to

the MCTSNode class:

int simulationDepth to store the number of steps from the root node to where the game ends in function simulateFrom()

ArrayList<MCTSNode> nextMinimaxMoves to store an array list pointing to the child nodes of the node that are the optimal moves found in the minimax search. nextMinimaxMoves is a subset of nextMoves.

Besides, a static variable is defined:

int MINIMAXSEARCH to indicate whether the minimax search should be taken and whether has been taken.

The changes to functions bestMCTSMove(), runTrial() and simulateFrom() are summarized as follows.

In bestMCTSMove(), lines 3 to 4 are replaced by the following.

Algorithm 5 Revision of function `bestMCTSMove()`

```
MINIMAXSEARCH = 0
for i ← 1 to T do                          ▷ T is the number simulation iterations
    if MINIMAXSEARCH == 1 then             ▷ Trigger minimax search
        minimax()
    else
        curNode.runTrial()
```

In runTrial(), lines 7 to 11 are replaced by the following. Here, constant NUMSIMULATIONDEPTH is the depth k in Figure 17(b)).

Algorithm 6 Revision of function `runTrial()`

```
if timesVisited == 0 OR node is a leaf node then
    simulationDepth ← levelFromTopNode
    rolloutResult ← simulateFrom(nodeGameState, nodeTurn, indexFromParentNode)   ▷ Simulation
    if simulationDepth ≤ NUMSIMULATIONDEPTH AND MINIMAXSEARCH==0 then
        MINIMAXSEARCH ← 1                  ▷ To trigger minimax search
else
    if MINIMAXSEARCH==2 then
        nextNode ← the child node with highest UCB score given in (9)
                                           ▷ UCT search among child nodes in nextMinimaxMoves
    else
        nextNode ← the child node with highest UCB score given in (9)
                                           ▷ UCT search among child nodes in nextMoves
    rolloutResult ← nextNode.runTrial()    ▷ Recursion
```

In simulateFrom(), the following is added immediately after line 8.

simulationDepth ← simulationDepth+1

Function minimax() does the minimax search and constructs the array lists nextMinimaxMoves of the root node and the child nodes up to level

NUMSIMULATIONDEPTH. I wrote two versions of minimax(). The first version uses recursion.

Algorithm 7 Minimax Version 1: Recursion

```
1: function int minimaxMove(TURN myTurn)
2:     if myTurn == COMPUTER then
3:         turn ← 1                                              ▷ Search for maximum
4:     else
5:         turn ← -1                                             ▷ Search for minimum
6:     if checkwin(nodeGameState, indexFromParentNode) == true then
7:         if Computer wins then
8:             totalScore ← 1                                    ▷ Win
9:         else
10:            totalScore ← -1                                   ▷ Loss
11:        return indexFromParentNode
12:    else
13:        if Node is a leaf node AND levelFromTopNode < NUMTREELEVEL then
                                  ▷ Parameter NUMTREELEVEL controls minimax search depth
14:            Construct and add all child nodes to nextMoves    ▷ Expansion
15:        if Node is a leaf node then          ▷ Node may be non-leaf after expansion
16:            totalScore ← 0                                    ▷ Draw
17:            return indexFromParentNode
18:        else
19:            nextTurn ← opposite of myTurn                     ▷ Toggle the turn
20:            for i ← 0 to nextMoves.size -1 do                 ▷ Check all child nodes
21:                nextNode ← i-th element of nextMoves
22:                nextNode.minimaxMove(nextTurn)                ▷ Recursion
23:                totalScore ← maximum of nextNode.totalScore * turn
24:                maxIndex ← corresponding index i
25:            return indexFromParentNode of maxIndex-th element of textttnextMoves
```

The recursive version of minimax() can successfully play Tic-Tac-Toe but runs into heap memory errors when playing Gomoku even with modest board sizes. Although the code is simple and clean, the recursive function calls consume a significant amount of memory. To overcome this problem, I wrote the second version using iteration. The second version does not run into any errors but is not general enough for any NUMSIMULATIONDEPTH. NUMSIMULATIONDEPTH has to be set to 4.

Algorithm 8 Minimax Version 2: Iteration

```
1: function int minimaxMove(TILESTATUS[] state)
2:     tmpValue ← 1
3:     minimaxValue1 ← -1
4:     for i1 ← 0 to K*K-1 do
5:         index1 ← a new EMPTY element
6:         state[index1] ← COMPUTER                              ▷ Mark a tentative move
```

```
7:      if checkwin(state, index1) == true then                          ▷ Computer wins
8:         state[index1] ← EMPTY                                         ▷ Restore state
9:         totalScore ← 1
10:        return index1                                                 ▷ No need to search further
11:     minimaxValue2 ← 1
12:     for i2 ← 0 to K*K-1 do
13:        index2 ← a new EMPTY element
14:        state[index2] ← PERSON                                        ▷ Mark a tentative move
15:        if checkwin(state, index2) == true then                       ▷ Person wins
16:           state[index2] ← EMPTY                                      ▷ Restore state
17:           minimaxValue2 ← -1
18:           break
19:        minimaxValue3 ← -1
20:        for i3 ← 0 to K*K-1 do
21:           index3 ← a new EMPTY element
22:           state[index3] ← COMPUTER                                   ▷ Mark a tentative move
23:           if checkwin(state, index3) == true then                    ▷ Computer wins
24:              state[index3] ← EMPTY                                   ▷ Restore state
25:              minimaxValue3 ← 1
26:              break
27:           minimaxValue4 ← 0
28:           for i4 ← 0 to K*K-1 do
29:              index4 ← a new EMPTY element
30:              state[index4] ← PERSON                                  ▷ Mark a tentative move
31:              if checkwin(state, index4) == true then                 ▷ Person wins
32:                 state[index4] ← EMPTY                                ▷ Restore state
33:                 minimaxValue4 ← -1
34:                 break
35:              state[index4] ← EMPTY                                   ▷ Restore state
36:           state[index3] ← EMPTY                                      ▷ Restore state
37:           if minimaxValue3 < minimaxValue4 then
38:              minimaxValue3 ← minimaxValue4
                                  ▷ minimaxValue3 is maximum of all child nodes' minimaxValue4
39:           state[index2] ← EMPTY                                      ▷ Restore state
40:           if minimaxValue2 > minimaxValue3 then
41:              minimaxValue2 ← minimaxValue3
                                  ▷ minimaxValue2 is minimum of all child nodes' minimaxValue3
42:     state[index1] ← EMPTY                                            ▷ Restore state
43:     if minimaxValue1 < minimaxValue2 then
44:        minimaxValue1 ← minimaxValue2
                                ▷ minimaxValue1 is maximum of all child nodes' minimaxValue2
45:        Initialize nextMinimaxMoves                                   ▷ if a new max is found, reset
46:        Add index1 to nextMinimaxMoves
47:     else if minimaxValue1 == minimaxValue2 then
48:        Add index1 to nextMinimaxMoves
49:     if tmpValue > minimaxValue2 then
50:        tmpValue = minimaxValue2                                      ▷ Store minimum of minimaxValue2
51:  if tmpValue > -1 then
52:     No sudden-death within 4 steps
53:  else if minimaxValue1 == -1 then
54:     All moves lead to sudden-death
55:  else
56:     MINIMAXSEARCH ← 2                                                ▷ Minimax search is done
57:  return index1
```