Advances in Intelligent Systems Research, volume 172
Sriwijaya International Conference on Information Technology and Its Applications (SICONIAN 2019)

Nature of Probability-Based Proof Number Search

Anggina PRIMANITA[1,2*], Hiroyuki IIDA[1]

[1]*School of Information Science, Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Nomi, 923-1211, Ishikawa, Japan*
[2]*Informatics Department, Universitas Sriwijaya, Indonesia*
**Corresponding author: S1820431@jaist.ac.id*

Abstract
Probability-based Proof Number Search (PPN-Search) is a best-first search algorithm that possesses a unique nature. It combines two kinds of information from a tree structure, namely, information from visited nodes and yet to be visited nodes. Information coming from visited nodes is determined based on winning status. On the other hand, information from yet to be visited (unexplored) nodes is determined by employing play-out technique in leaf nodes. All of the information is combined into a value called probability-based proof number (*PPN*). In this paper, PPN-Search is employed to solve randomly generated Connect Four positions. Its results are compared to two other well-known best-first search algorithms, namely Proof Number Search and Monte-Carlo Proof Number Search. The limitation of PPN-Search related to the use of real numbers is identified based on the experiment. To increase the performance of PPN-Search while preserving its strength, an improvement technique using precision rate is introduced. Analysis from further experiments shows that the addition of the precision rate value accentuates the nature of PPN-Search, especially in its ability to combine information into *PPN*, which leads to increased performance. It is marked by reduced number of nodes needed to be explored up to 57% compared to implementation without precision rate.

Keywords: *best first search, Probability-based Proof Number Search, Connect Four*

Background

Proof number search (PNS) [1] is a best-first search algorithm that was initially introduced as a mean to find the game-theoretic value. It is implemented as part of the attempt to solve a game. PNS uses two values, proof number and disproof number (*pn* and *dn* for short, respectively) as indicators. The proof number of a node represents the smallest number of leaf nodes that have to be proven in order to prove that it is a win, while the disproof number represents the smallest number of leaf nodes that have to be disproved in order to prove that it is a loss. Essentially, *pn* and *dn* are numbers that represent the difficulty to prove a node. These numbers hold importance in choosing the most proving node (*MPN*), which is the node that will be expanded for subsequent search.

Employing PNS and its variants to find a game-theoretic value has shown to be fruitful [2] . However, there were problems which arose from its implementation. The problems observed include: (1) a large amount of memory space usage, (2) difficulty in application to a non-tree state space, and (3) overly long solutions caused by its depth-first behavior [3]. Moreover, PNS also encounters the see-saw effect [4]. An improvement of PNS that aims to reduce the usage of computation resource if the Depth-first Proof Number (df-*pn*) Search [5]. It turns PNS, a best-first search algorithm, into depth-first search by introducing iterative deepening into the algorithm. Through the usage of iterative deepening, df-*pn* expands less interior node, as well as reduce the amount of proof number and disproof number that has to be recomputed.

Another prevalent algorithm that was employed to attempt solving games is the Monte-Carlo Tree Search (MCTS). Its usage has become well known especially in the field of Go. It is a part of the search algorithm that has led to AlphaGo's wins against top grandmasters [6]. Monte-Carlo Search is proposed by Coulom in [7], then Kocsis and Szepesvári introduced the concept of upper confidence bounds applied to trees (UCT) in [8]. Thereafter, Chaslot in [9] developed MCTS. It aims to overcome the difficulty found in building heuristic knowledge for a non-terminal game state by employing stochastic simulations [10]. Multiple simulations have shown to be an effective method to determine game-playing strategy. The MCTS framework consists of four major steps, namely selection, expansion, simulation, and backpropagation. The algorithm starts with selecting the next action based on a stored value (selection), and then, when it encounters a state that cannot be found in the tree, it expands the node. The node expansion is based on multiple randomly simulated games. The value is then stored and backpropagated to the root of the tree, where the algorithm continues to repeat the steps until the desired outcome is reached.

With both strengths, i.e., combining PNS and MCTS, we can increase the expected quality of a game solver. One early example of this combination is the Monte-Carlo Proof Number Search (MCPNS) [11]. The algorithm is proposed as an improvement to PNS as it offers a higher degree of flexibility than PNS, while it still retains its reliability. Its main framework is similar to that of MCTS, but it uses the Monte-Carlo evaluation to guide PNS in expanding nodes, resulting in a more efficient order. In its implementation, however, MCPNS employed the MIN/SUM rules in its backpropagation step. The MIN/SUM rules have the drawback to compute results from integer numbers, instead of real numbers produced from statistical results in its simulation.

To conform to the requirement of processing real numbers derived from statistical probability, a new variant of PNS has been proposed. It is called probability-based proof number search (PPN-Search) [12]. It applies the idea of

"searching with probabilities" first suggested by [13] to draw better results from real numbers produced by Monte-Carlo simulation in the play-out step. PPN-Search is independently developed but uses similar principles with Product Propagation (PP) [14]. The main difference between PPN-Search and PP is that PPN-Search indicator is derived from both the explored and unexplored area of the tree to obtain information about the current state of the game. PP, on the other hand, utilizes the information only from the explored area of the tree [15]. In its introduction, PPN-Search was implemented to solve a simulated balanced game-tree structure, called P-Game Tree. The experiment shows a significant and meaningful result, but, its nature and characteristics are still unexplored.
This paper employs PPN-Search to solve positions of a real game, Connect Four. Experiments conducted in this article consist of the application of PPN-Search to solve real game positions and compare them to other major algorithms. Its advantages, limitations, and possible improvement are mentioned.

Probability-based Proof Number Search

Probability-based proof number search (PPN-Search) is a game solver algorithm that aims to improve the idea of PNS. It uses an indicator called "Probability-based Proof Number" (PPN) to indicate the probability of proving a node [12]. In a leaf node, its PPN is derived from the Monte-Carlo evaluation. This value is then being backpropagated to an internal node using AND/OR probability rules. Details of PPN-Search are specified in the following two subsections.

Probability-based Proof Number. A probability-based proof number is a number that specifies the probability of a node to be proven in an AND/OR tree. There are three types of nodes in such a tree, viz. terminal node, leaf node, and internal node. All of these nodes have its own *PPN* (n.*PPN*) that is calculated based on the following formula:

If a node n is a terminal leaf node:
If (*n*) is a winning node,

$$n.ppn = 1 \quad (1)$$

If (*n*) is not a winning node,

$$n.ppn = 0 \quad (2)$$

If a node n is a leaf node, then let R be the winning rate as a result of game play-out, and θ is a small positive number close to 0:

If $R \in (0,1)$,

$$n.ppn = R \quad (3)$$

If $R = 1$,

$$n.ppn = R - \theta \quad (4)$$

If $R = 0$,

$$n.ppn = R + \theta \quad (5)$$

If a node n is an internal node, then:
If (*n*) is an OR node,

$$n.ppn = 1 - \prod_{n_c \in children\ of\ n} 1 - n_c.ppn \quad (6)$$

If (*n*) is an AND node,

$$n.ppn = \prod_{n_c \in children\ of\ n} n_c.ppn \quad (6)$$

as seen on the formula, *PPN* that is assigned to an OR and AND internal node is the product of its child values. Fig. 1 illustrates these calculations.
PPN of a node contains two different information derived from the current game-tree. The first information is acquired from the current known tree structure (Eq. 1). The second information is derived from the currently unknown tree structure. Eq. 2 is employed to calculate the probability of winning or losing from the current position, thus, providing more information from part of the tree that is yet to be expanded.
While obtaining *PPN* of a node, every statistical value produced by a simulated game is accounted (Fig. 1). This is different than the MIN/SUM rules used by MCPNS (Fig. 2). Usage of MIN/SUM rules may disregard some of these values due to its rule.
To avoid confusion, in this paper the first player is always an OR node. At any given node, *PPN* signifies the possibility of the node to be proved; a higher value means that it is more likely to be proved, which means that from the root position, it is more likely for the first player to win the game. *PPN* of the left side AND node in Fig. 1 is the result of applying Eq. (7). Its *PPN* is the product of all its children. *PPN* of the OR node at the top of the subtree is the result of applying Eq. (6). As seen in the figure, children of all nodes affect the way *PPN* is calculated. The distinction of the probability rule application can be observed as the game-tree of MCPNS is displayed in Fig. 2. MCPNS applies the AND/OR rule in updating its *pmc* value. The left side AND node's *pmc* is the sum of all its children, and the top OR node's *pmc* is the minimum *pmc* of its children. In this case, the *pmc* that has been previously calculated does not affect the value of the top OR node.

Algorithm of Probability-based Proof Number. The PPN-Search consists of four following steps:
Selection: For all nodes, select the child with maximum *PPN* at OR nodes, and child with minimum *PPN* at AND nodes. Regard these nodes as the most proving node (*MPN*) for expansion.
Expansion: Expand the most proving node. In this phase, all available next moves from the *MPN* position is regarded as child.
Play-out: For positions that are not already in the tree. Simulate the move in a random self-play mode until the end of the game. After several play-outs, the PPNs of expanded nodes are derived from Monte-Carlo evaluations. In this step, if the result of simulations (R) is equal to 1 or 0, it is either reduced or added with a small value (θ) to differentiate it from Leaf node.
Backpropagation: Update the PPNs from extended nodes back to the root, while following the AND/OR probability rules given in subsection 2.1.
all of the steps are repeated until PPN of the root node reaches 1 or 0 (see Algorithm 1). If the PPN is equal to 1, the game is defined as solved.

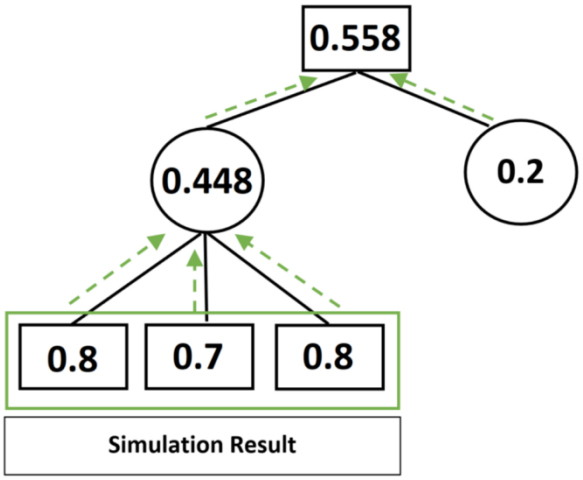

Figure 1. Illustration of *PPN* calculation in PPN-Search. OR nodes are displayed as square, while AND nodes are displayed as circle.

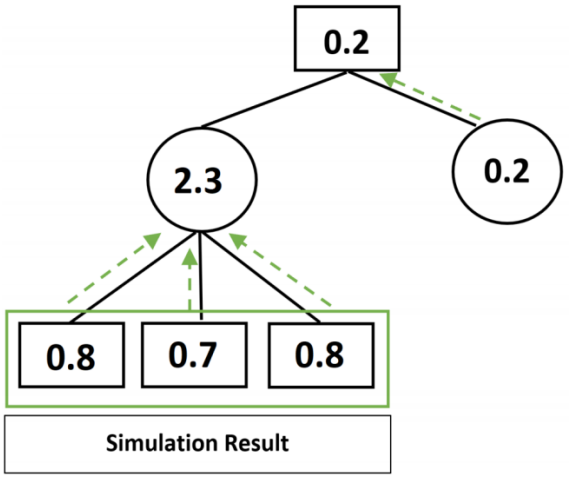

Figure 2. Illustration of *pmc* calculation in MCPNS. OR nodes are displayed as square, while AND nodes are displayed as circle.

Algorithm 1 Probability-based Proof Number Search

```
1: function PPNSearch(root)
2:     Create root node
3:     while root.ppn ≠ 0 and root.ppn ≠ 1 do
4:         MPN = Selection(root)
5:         Expansion(MPN)
6:         BackPropagation(root)
7:     if root.ppn = 1 then
8:         return true                          ▷ Root is solved
9:     else if root.ppn = 0 then
10:        return false                         ▷ Root is unsolved
```

Probability-based Proof Number Search on Connect Four

Connect Four is a two-player perfect information board game. The objective of the game is to line up four chips in an either horizontal, vertical, or diagonal manner. The game can be won as quick as 13-ply or until the board is filled in 42-ply, making the depth of the tree to be highly varied. There are also possibilities for sudden death moves to occur during the game which made its game-tree structure unbalanced. In terms of complexity, Connect Four has a fairly good game-tree. This is due to its move limitation as well as the size of the board. All of these traits of Connect Four made it suitable to be used as test-bed to observe the nature and characteristics of PPN-Search as it is vastly different than the previous experiment.

Experimental Setup. The experiment is configured to check algorithms' performance in limited time and memory space. In the experiment, 200 Connect Four positions were generated. Each position contains 12-ply randomly generated moves. Three different algorithms, PNS, MCPNS, and PPN-Search were given the task to solve each position independently. All algorithms are stopped once it expands into 35,000,000 nodes or the time elapsed reaches 420 seconds. For the experiment, parameters for each algorithm are as follows; For MCPNS and PPN-Search, the number of simulated moves is 60, and for PPN-Search, $\theta = 0.01$. The number of iterations performed, of nodes visited, and time used to solve the position were measured.
The experiments were performed by a computer with Intel i5-8400 processor running at 2.81 GHz using 8 GB of RAM, running Windows 10, on a 64-bit machine. To ensure the correctness and independent measurement, the experiments are performed sequentially. It means that only one position is solved by one algorithm at a time.

Experimental Result. The result of the experiment is shown in Fig. 3. All of the positions produced the same conclusion unless it reached either the set limit. In the limited configuration, PNS performs the best, with the most amount of positions being solved and unsolved, totaling in 122 positions. The second-best performing algorithm is PPN-Search. It solved a total of 102 positions. The algorithm with the least performance is MCPNS as it solves a total of 80 out of 200 positions.
One notable thing from this experiment result is that PNS is bounded by the amount of memory available, while PPN-Search and MCPNS are bounded by time. All of the PNS that stopped midway of the search is because it exceeds the number of nodes visited, while all of PPN-Search and MCPNS stopped midway because it exceeds the time limit set for the experiment.

Prolonged Search in PPN-Search. Based on its nature, PPN-Search combines two kinds of information, the information from explored part of the tree (visited nodes) and information from the unexplored part of the tree (yet to be visited nodes). PNS, on the other hand, only uses information gained from explored part of the tree. With more sources of information, PPN-Search should be able to utilize it to reach to the conclusion. To further inspect the utilization of this information, change of *PPN* of the root of the Connect Four position that has the highest time difference between PNS and PPN-Search is inspected. The highest time difference between PNS and PPN-Search is 322.345 second. For such position, PPN-Search visited the total of 2,295,856 nodes, while PNS and MCPNS visited 1,921,730 and 1,114,283 nodes respectively. This case indicates that PPN-Search was not able to fully combine the information from both parts of the tree, which leads to sub-optimal performance. To explain why such occurrences

exist, *PPN* value of the root for every 10,000 iterations of the chosen position is displayed in Fig. 4.

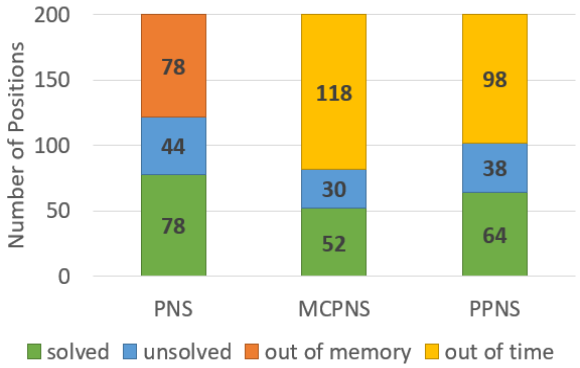

Figure 3. Number of Connect Four positions solved, unsolved, and out of bounds by PNS, MCPNS, and PPN-Search.

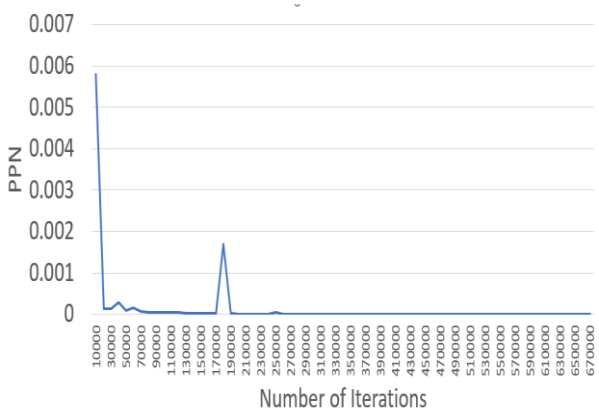

Figure 4. *PPN* value of the root of the position with the highest time difference between PNS and PPN-Search.

The *PPN* value of the root at the first 10,000 iterations is 0.005794, which is also the peak value for the entire solving process. The value is then kept decreasing and stabilized between the 70,000th iteration until the 170,000th iteration. A sudden increase appears at the 180,000th iteration, in which the value of *PPN* of the root is equal to 0.001699. Following the sudden increase, the *PPN* value stayed steadily under 0.001 for the rest of the search process. Based on the output value, *PPN* value of the root already hits 0.000000 by the 340,000th iteration, however, the algorithm keeps iterating until it reaches the total of 677,765 iterations, which is almost twice number of iterations. This also affects the number of nodes needed to solve the position. By the 340,000th iteration, total number of nodes explored is 1,205,010 nodes, while at the 677,765th iteration, total number of nodes explored is also almost twice, that is 2,274,949 nodes.

To observe the problem from different perspective, a solved position with a large time difference is observed. In this position, the elapsed time difference between PNS and PPN-Search is 77.802 second. For this position, PPN-Search visited a total of 661,783 nodes, while PNS and MCPNS visited 11,055 and 1,567 nodes respectively. In this case, the difference between total nodes visited by PNS and MCPNS is very big. Observation of *PPN* value of the root of the second chosen position (Fig. 5 shows similar trend to that of the first chosen position. At the first 10,000 iterations, the *PPN* value of the root is 0.99995. The *PPN* value keeps changing between the start until the 100,000th iteration, in which it outputs *PPN* value of 1.00000. However, in the same fashion of the previous position, the algorithm keeps iterating until 214,110 iterations.

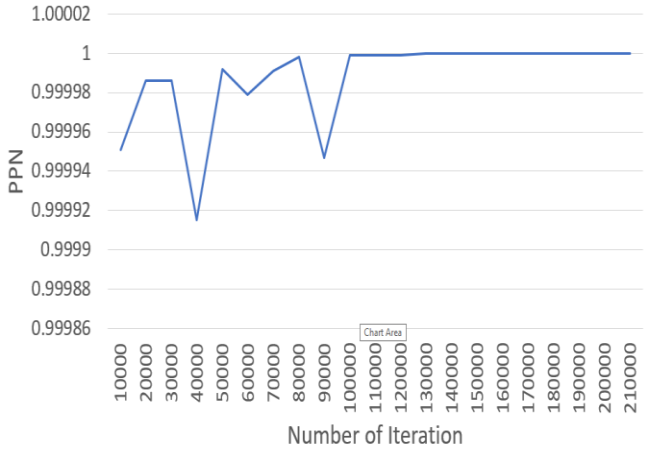

Figure 5. *PPN* value of the root of a solved position with substantial time difference between PNS and PPN-Search.

From the two observations, it can be concluded that one of the problems that occur in PPN-Search is the prolonged search problem. This problem occurs when the *PPN* value of the root hits either 1 or 0, but the algorithm keeps searching, resulting in longer search time. Prolonged search problem stems from the precision of floating-point representation in the computer system. Floating point is represented as fractions of the binary system in the computer memory, hence, a floating-point representation of a certain value, albeit that it has integer representation, might not be exact [16]. In this implementation of PPN-Search, although the *PPN* value of the root has reached the value of 0.000000, it still contains a very small number trailing behind, thus, not recognized as a conclusive value. The programming language of which the algorithms are implemented in C++, however, this also applies to other popular programming languages such as Python or Java, which made the problem unavoidable. The following section proposes an approach that can be used to mitigate the identified problem.

Precision of Probability-based Proof Number Search

PPN-Search suffers from prolonged search problem that is caused by the representation of floating-point. To tackle this problem, a solution is proposed. The solution is to apply a precision rate (*pr*). The application of this rate is aimed to reduce the rounding error in comparing the *PPN* root value. The *pr* value is applied in Algorithm 1 to become Algorithm 2. The rest of the algorithm is not affected by this change.

Algorithm 2 Probability-based Proof Number Search

```
1: function PPNSEARCH(root)
2:     Create root node
3:     while |root.ppn − 0| ≤ pr and |root.ppn − 1| ≤ pr do
4:         MPN = Selection(root)
5:         Expansion(MPN)
6:         BackPropagation(root)
7:     if root.ppn = 1 then
8:         return true                          ▷ Root is solved
9:     else if root.ppn = 0 then
10:        return false                         ▷ Root is unsolved
```

To show the efficacy of this enhancement, experiments with different values of θ and *pr* is performed.

Experimental Setup. To see the influence of *pr* value in PPN-Search performance, an experiment is done using the same hardware in Section 3.1. PPN-Search is employed to solve the same 200 positions generated in the previous experiment. Search is ended when the position is considered to be solved, unsolved, or it reaches the limit of 420 seconds or 35,000,000 nodes.
In the precision experiment, two different values of θ are used, viz. 0.01 and 0.001. Three different *pr* values are used with the chosen θ. The *pr* values are 0.001, 0.0001, and 0.00001. This setup results in six different configurations.

Experimental Result and Analysis. Six different configurations of θ and *pr* value is used to solve 200 Connect Four positions. Result of the configuration for θ = 0.01 is shown in Fig. 6 while the result for θ = 0.001 is shown in Fig. 7.

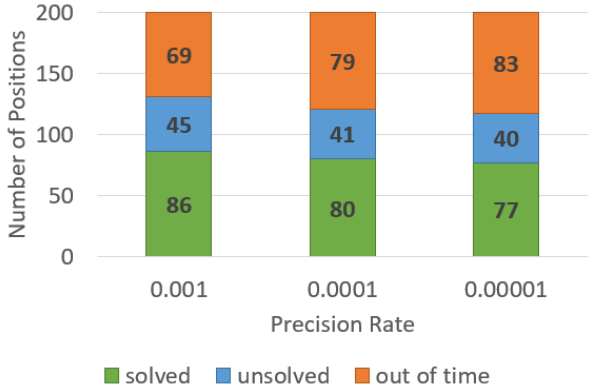

Figure 6. Number of positions solved, unsolved and out of bounds by PPN-Search with θ = 0.01.

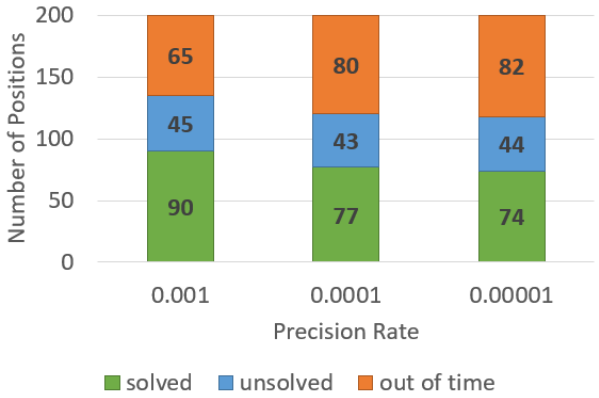

Figure 7. Number of positions solved, unsolved, and out of bounds by PPN-Search with θ = 0.001.

With the application of *pr* value, the total positions that reach a conclusion, whether it is solved or unsolved, is increased. The configuration that leads to the highest amount of positions is the configuration with θ = 0.001 and *pr* = 0.001, with the total of 135 positions. This is higher than that of PNS (122 positions) and MCPNS (82 positions) from the first experiment. Elapsed time and nodes explored are also affected by the application of *pr* value. The average elapsed time and nodes explored is displayed in Table 1. Change of θ value does not significantly affect the average time and node, but, the increase of precision rate yields longer elapsed time and higher number of nodes visited. The same with total number of solved positions, configuration with θ = 0.001 and *pr* = 0.001 has shown the best performance.
Based on the experiment result, a small number of *pr* value leads to increasing the total number of positions that reaches a conclusion. However, lowering *pr* value to a rate where it is less precise than that of θ would lead to false conclusions. In this case, the best *pr* value would be that of the same with precision, or one rate above it. Increase and decrease of the value beyond it would decrease the performance of PPN-Search.

Table 1. Average time and node for each PPN-Search configurations

θ	*pr*	Average Time (s)	Average Node
0.01	no precision	251.180925	2,004,052.895
0.01	0.001	183.256315	1,323,718.975
0.01	0.0001	214.63364	1,582,187.985
0.01	0.00001	226.707205	1,667,750.875
0.001	0.001	174.90736	1,269,587.96
0.001	0.0001	217.987025	1,614,710.08
0.001	0.00001	229.07937	1,638,641.2

DISCUSSION

Experimental results of implementing three best first algorithms to Connect Four show that there are positions in which PPN-Search performs sub-optimally. It visits more nodes than PNS before solving positions with the same result. This is against the idea of which PPN-Search is based on. Of which that it does not have to visit all of the nodes, but instead, combining information for the visited node, and the probability of unexplored nodes.

Upon further observation, one of the problems identified that cause PPN-Search to underperform is the prolonged search. This problem stems from the usage of real number in PPN-Search. The previous algorithms, PNS and MCPNS, uses integer-based backpropagation technique. Because of this technique, the precision of floating-point does not affect its performance. PPN-Search uses real number-based backpropagation technique, in which the *PPN* of a node went through product operations. With this change, a new risk related to precision arose.

To negate the risk related to precision, a new parameter called precision rate is introduced to PPN-Search algorithm. the results of the experiments with various configurations show that the addition of *pr* value increases the performance of PPN-Search without affecting its accuracy result. Observation upon the results shows that the closer the *pr* value to θ the better PPN-Search performance result will be. However, *pr* value cannot be lower than θ, as it will decrease the performance of PPN-Search

The results produced from the experiments demonstrate that PPN-Search with *pr* value reduces the number of explored nodes needed to solve a position up to 57%. This implies that even with a smaller number of explored nodes, it can exploit information from unexplored area and combines it to reach the desired conclusion. This identified nature leads to a new hypothesis, that PPN-Search is suitable for a game that requires a long look-ahead strategy. PPN-Search has been introduced to solve two different tree structures, balanced and unbalanced. In both implementations, PPN-Search demonstrates better performance than the other algorithms. However, it has not been tested with a game that requires long look-ahead strategy. The current state of best performance algorithm on different tree structures is shown in Table 2. For a game with a hard problem, the current best performing algorithm is depth-first proof number search (Df-pn). It is an expansion of PNS that is aimed to tackle larger problems. In the future, it is important to see the expansion of probability-based search idea into its depth-first version and test it on a game with bigger tree size and harder problems to test this hypothesis.

CONCLUSION

PPN-Search is a best first search algorithm that employs information from both inside and outside of a game-tree. In this paper, PPN-Search is employed to solve randomly generated Connect Four positions. Results from the application allow a problem related to the implementation of PPN-Search algorithm to be identified. The prolonged search problem arises because PPN-Search is highly dependent on real number-based operations. To alleviate the problem, a new parameter is introduced. Result of the improvement shows that PPN-Search is able to solve more positions in limited configurations than other best first search algorithms. Implementations of PPN-Search on different tree structures show that in the event of reduced explored information, PPN-Search is able to reach the desired conclusion by exploiting information from unexplored part of a game-tree.

Table 2. Best performance search algorithms in different tree structure.

Tree Structure	Complexity	Best Performance Algorithm
Balanced Tree	Small (easy)	PPN-Search on P-game tree [12]
Unbalanced Tree	Big (hard)	Df-pn in Go [17]
Unbalanced Tree	Small (easy)	PPN-Search on Connect Four

Further works in this direction include, but is not limited to (1) application of PPN-Search in other real game with a larger tree; (2) expansion into depth-first version of PPN-Search (df-PPNS); (3) comparison between depth-first proof number search and df-PPNS in (very) hard problem domains.

REFERENCES

[1] L. V. Allis, M. van der Meulen and H. J. van Den Herik, "Proof-number Search," Artificial Intelligence, Bd. 66, Nr. 1, pp. 91-124, (1994).

[2] H. J. van Den Herik and M. H. Winands, "Proof-number Search and Its Variants," in Oppositional Concepts in Computational Intelligence, Springer, (2008), pp. 99-118.

[3] A. Kishimoto, M. H. Winands, M. Muller and J. T. Saito, "Game-tree Search Using Proof Numbers: The First Twenty Years," ICGA Journal, Bd. 35, Nr. 3, pp. 131-156, (2012).

[4] T. Ishitobi, A. Plaat, H. IIda and H. J. van den Herik, "Reducing The Seesaw Effect With Deep Proof-Number Search," in Advances in Computer Games, Springer International Publishing, (2015), pp. 185-197.

[5] A. Nagai, "Df-pn Algorithm for Searching AND/OR Trees and Its Applications,“ PhD thesis, Department of Information Science, University of Tokyo, (2002).

[6] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam and M. Lanctot, "Mastering the Game of Go With Deep Neural Networks and Tree Search,“ Nature, Bd. 529, Nr. 7587, p. 484, *2016).

[7] R. Coulom, "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search,“ in International Conference on Computer and Games, (2006).

[8] L. Kocsis and C. Szepesvari, “Bandid Based Monte-Carlo Planning,“ in ECML-06, 2006.

[9] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. v. D. Herik, J. W. H. M. Uiterwijk and B. Bouzy, “Progressive Strategies for Monte-Carlo Tree Search,“ New Mathematics and Natural Computation, Bd. 4, Nr. 3, pp. 343-357, (2008).

[10] C. Guillaume, E. Bakkes, I. Szita and P. Sponck, “Monte-Carlo Tree Search: A new Framework for Game AI,“ in In Proceedings of AIIDEC-08, (2008).

[11] J.-T. Saito and G. Chaslot, “Monte-Carlo Proof Number Search for Computer Go,“ in International Conference on Computer and Games, (2006).

[12] Z. Song and H. v. D. H. H. J. IIda, “Probability based Proof Number Search,“ in Proceedings of the 11th International Conference on Agents and Artificial Intelligence, {ICAART} 2019, (2018).

[13] A. J. Palay, “Searching with Probabilities,“ Carnegie-Mellon Univ. Pittsburgh PA Dept of Computer Science, (1983).

[14] A. Saffidine and T. Cazenave, “Developments on Product Propagation,“ in International Conference on Computers and Games, (2013).

[15] D. Stern, R. Herbrich and T. Graepel, “Learning to Solve Game Trees,“ in Proceedings of the 24th international conference on Machine learning, (2007).

[16] W. Kahan, “IEEE standard 754 for binary floating-point arithmetic,“ Lecture Notes on the Status of IEEE, Bd. 754, p. 11, (1996).

[17] A. Kishimoto and M. Muller, “Df-pn in Go: An Application to The One-Eye Problem,“ Advances in computer games, pp. 125-141, (2004).