ECAI 2025
I. Lynce et al. (Eds.)

doi:10.3233/FAIA251406

Generalized Proof-Number Monte-Carlo Tree Search

Jakub Kowalski[a,*], Dennis J. N. J. Soemers[b], Szymon Kosakowski[a] and Mark H. M. Winands[b]

[a] Faculty of Mathematics and Computer Science, University of Wrocław, Poland
[b]Department of Advanced Computing Sciences, Maastricht University, The Netherlands
ORCID (Jakub Kowalski): https://orcid.org/0000-0003-1932-4278, ORCID (Dennis J. N. J. Soemers): https://orcid.org/0000-0003-3241-8957, ORCID (Szymon Kosakowski): https://orcid.org/0009-0006-7992-0310, ORCID (Mark H. M. Winands): https://orcid.org/0000-0002-0125-0824

Abstract. This paper presents Generalized Proof-Number Monte-Carlo Tree Search: a generalization of recently proposed combinations of Proof-Number Search (PNS) with Monte-Carlo Tree Search (MCTS), which use (dis)proof numbers to bias UCB1-based Selection strategies towards parts of the search that are expected to be easily (dis)proven. We propose three core modifications of prior combinations of PNS with MCTS. First, we track proof numbers per player. This reduces code complexity in the sense that we no longer need disproof numbers, and generalizes the technique to be applicable to games with more than two players. Second, we propose and extensively evaluate different methods of using proof numbers to bias the selection strategy, achieving strong performance with strategies that are simpler to implement and compute. Third, we merge our technique with Score Bounded MCTS, enabling the algorithm to prove and leverage upper and lower bounds on scores—as opposed to only proving wins or not-wins. Experiments demonstrate substantial performance increases, reaching the range of 80% for 8 out of the 11 tested board games.

1 Introduction

Monte-Carlo Tree Search (MCTS) [10, 13] is a best-first search method guided by the results of Monte-Carlo simulations, well established in game AI [7, 29]. Using the results of previous simulations, the method gradually builds up a game tree in memory and increasingly becomes better at accurately estimating the values of the most promising moves. MCTS has substantially advanced the state of the art in several deterministic game domains, in particular Go [23], but also other board games including Amazons [16], Hex [2], Lines of Action [28], and General Game Playing (GGP) [5].

In tactical games, where the main line towards the winning position is typically narrow with many non-progressing alternatives, MCTS may often lead to an erroneous outcome because the nodes' values in the tree do not converge fast enough to their game-theoretic value. To mitigate this effect, MCTS variants have been proposed that integrate minimax search [27, 25, 15, 4]. Recently, Proof-Number Search (PNS) [1] has been integrated in MCTS [11, 20]. PNS has the advantage proving endgames faster than traditional minimax in many domains. The variant PN-MCTS [14] has been shown to improve over default MCTS in domains such as Lines of Action, MiniShogi, Knightthrough, and Awari.

* Corresponding Author. Email: jakub.kowalski@cs.uni.wroc.pl

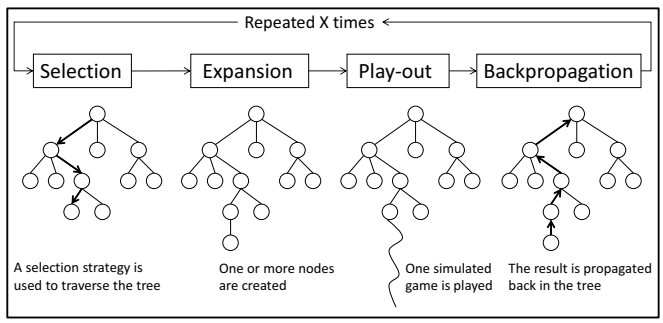

Figure 1: Outline of Monte-Carlo Tree Search [9].

In this paper, we propose a generalization of PN-MCTS, called Generalized Proof-Number Monte-Carlo Tree Search (GPN-MCTS). First, this extension of PN-MCTS tracks proof numbers per player, which reduces code complexity in the sense that we no longer need disproof numbers, and generalizes the technique to be applicable to games with more than two players. Second, GPN-MCTS contains different enhancements of using proof numbers to bias the UCT selection strategy [13], achieving strong performance with strategies that are simpler to implement and compute. Third, PN-MCTS is integrated with Score Bounded MCTS [8], enabling the technique to prove and leverage upper and lower bounds on scores—as opposed to only proving wins or non-wins.

For the purpose of the experiments, GPN-MCTS was implemented in the Ludii general game playing system [19]. Although it is not guaranteed to improve performance for every game, in many cases when it works, its improvements are significant, reaching 80% win rate on 8 out of the 11 tested board games, and over 60% on the remaining three. Moreover, we show that it pairs well with the Score-bounded MCTS enhancement.

2 Background

2.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [10, 13] is a best-first search method that gradually builds up a search tree, balancing exploitation of parts that seem promising based on earlier iterations, with explorations of parts that were infrequently explored. It does this by iterating through four strategic steps [9], depicted in Figure 1, until a time or iteration budget expires.

Selection Step. The selection step traverses the tree, starting from the root node, until a node is reached for which there are still legal actions that have not yet been expanded into nodes of the search tree (or until a terminal node is reached). This step implements the trade-off between exploration and exploitation. One of the most common baseline implementations of MCTS—referred to as *Upper confidence Bounds applied to Trees* (UCT) [13]—uses the UCB1 strategy [3] to choose among the children of any given current node. It works as follows. Let I be the set of nodes immediately reachable from the current node p. The selection strategy selects the child b of node p that satisfies Formula (1):

$$b \in \operatorname{argmax}_{i \in I} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \right), \tag{1}$$

where v_i is the estimated value of the node i, n_i is the visit count of i, and n_p is the visit count of p. C is a hyperparameter which can be tuned experimentally. Here, ties are broken randomly.

Expansion Step. As previously stated, the selection step continues until a node is reached that has not yet expanded all of its children. Among the children that have not been stored in the tree, one is selected uniformly at random. This node L is then added as a new leaf node. If the selection step arrives at a terminal node, the expansion and subsequent play-out steps are skipped.

Play-out Step. From the new leaf node L onwards, the play-out step is performed. Moves are selected in self-play until the end of the game is reached. This step might consist of playing uniformly random moves or—often better—semi-random moves chosen according to a *simulation strategy*.

Backpropagation Step. In the final step, the result *R* of a play-out k is backpropagated from the leaf node L, through the previously traversed nodes, all the way up to the root. The result is scored positively ($R_k = +1$) if the game is won, and negatively ($R_k = -1$) if the game is lost. Draws lead to a result $R_k = 0$. A *backpropagation strategy* is applied to the *value* v_i of a node i. Here, it is computed by taking the average of the results of all simulated games made through this node [10], i.e., $v_i = (\sum_{k \in K} R_k)/n_i$, where K is the set of indices for all play-outs. Visit counts n_i for all nodes along the trajectory are also incremented.

When the search budget expires, the move that is ultimately selected to be played is the one from the root node that has the highest visit count (though other strategies are possible as well [9]).

2.2 Proof-Number Search

Proof-Number Search (PNS) is a best-first search method especially suited for finding the game-theoretic value in game trees [1]. Its aim is to prove a particular goal. In the context of this paper, the goal is to prove a forced win for the player to move. A tree can have three values: *true*, *false*, or *unknown*. In case of a forced win, the tree is *proven* and its value is true. In case of a forced loss or draw, the tree is *disproven* and its value is false. Otherwise, the value of the tree is unknown. As long as the value of the root is unknown, the most-promising node is expanded. Like MCTS, PNS does not need a domain-dependent heuristic evaluation function to determine the most-promising node [1]. In PNS, this node is usually called the *most-proving* node. PNS selects the most-proving node using two criteria: (1) the shape of the search tree (the branching factor of every internal node) and (2) the values of the leaves. These two criteria enable PNS to treat game trees with a non-uniform branching factor efficiently.

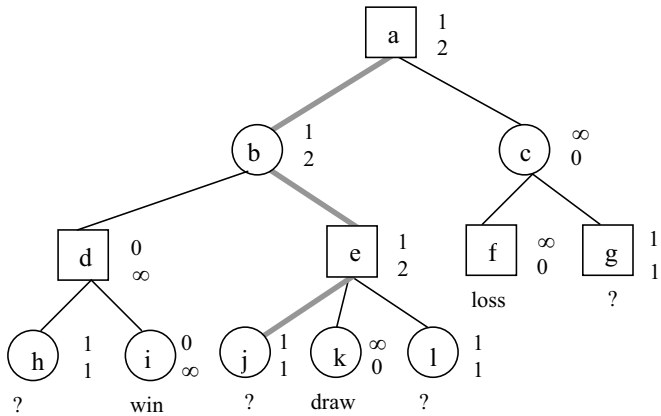

Figure 2: An AND/OR tree with proof and disproof numbers [26]. A square denotes an OR node, and a circle denotes an AND node. The numbers to the right of a node denote the proof number (upper) and disproof number (lower).

PN search represents the game as an AND/OR tree. OR nodes correspond to positions with the first player to play, while in AND nodes, the second player is to play. An example of such a tree is given in Figure 2. In PNS, the *proof number* (*pn*) represents the minimum number of leaf nodes, which have to be proven in order to prove the node. Analogously, a *disproof number* (*dpn*) represents the minimum number of leaf nodes that have to be disproven in order to disprove the node. Because the goal of the search is to prove a forced win, winning nodes are regarded as proven. Therefore, they have $pn = 0$ and $dpn = \infty$. Lost or drawn nodes are regarded disproven. They have $pn = \infty$ and $dpn = 0$. Unknown leaf nodes have $pn = 1$ and $dpn = 1$. The *pn* of an internal OR node is equal to the minimum of its children's proof numbers, because to prove an OR node it suffices to prove one child. The *dpn* of an internal OR node is equal to the sum of its children's disproof numbers, because to disprove an OR node all the children have to be disproven. The *pn* of an internal AND node is equal to the sum of its children's proof numbers, because to prove an AND node all the children have to be proven. The *dpn* of an AND node is equal to the minimum of its children's disproof numbers, because to disprove an AND node it suffices to disprove one child.

The procedure of selecting the most-proving node to expand next is as follows. The algorithm starts at the root. Then, at each OR node the child with the smallest *pn* is selected as successor, and at each AND node the child with the smallest *dpn* is selected as successor. Finally, when a leaf node is reached, it is expanded (which makes the leaf node an internal node) and the newborn children are evaluated.

2.3 Proof-Number Monte-Carlo Tree Search

Proof-Number Monte-Carlo Tree Search (PN-MCTS) was initially proposed in [11], introducing an enhancement of the UCB1 formula for two-player zero-sum games that includes a PNS-related term biasing selection towards the children preferred by Proof-Number Search. It was later extended by [14] to take advantage of the observation that already computed (dis)proof numbers can also serve to bias final move selection and skip solved subtrees, similarly to a Score-Bounded MCTS [8]. In this work, we aim to generalize on all the above accomplishments.

PN-MCTS tracks proof and disproof numbers from the point of

view of the root node player in all nodes of the MCTS tree. The OR/AND nodes are assigned accordingly, from the root player's perspective. The PNS-based part of the algorithm is a paranoid type [22] as opponent decisions, based on disproof numbers, make it more interested in preventing the root player's victory than maximizing its own result.

The proposed UCT-PN formula, shown in (2), extends the standard UCB1 (1), introducing a term that is greater for child nodes that are closer to being solved according to (dis)proof numbers, weighted by a C_{pn} constant.

$$b \in \operatorname{argmax}_{i \in \mathcal{I}} \left(v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + C_{pn} \times \mathrm{PNRank}(i, \mathcal{I}) \right) \tag{2}$$

Instead of directly using the (dis)proof numbers in the formula, the idea is to sort their values and use a ranking system. This process aims to reflect the observation that the magnitudes of differences amongst (dis)proof numbers technically do not have much meaning, and is encapsulated by the PNRank function (3).

$$\mathrm{PNRank}(i, \mathcal{I}) = 1 - \frac{\mathrm{rank}(i)}{\max_{j \in \mathcal{I}} \mathrm{rank}(j)} \tag{3}$$

The best node—the one that would be chosen by PNS—gets a rank of 1. This is the child with the lowest proof number in OR nodes, and the one with the lowest disproof number in AND nodes. The next one in order would get rank 2, and so on. Tied nodes are awarded the same rank. Then, the ranks are normalized into the range of $[0, 1]$, to allow easier scaling with the exploration and exploitation terms of the UCB1 formula.

Experiments performed in [14] showed that the introduction of UCT-PN was enough to achieve an overwhelming performance (around 87% versus the vanilla MCTS) in Lines of Action. Further extensions by final move selection and subtree solving were necessary to obtain winrates above 65% for MiniShogi and Knighttrough. To obtain a winrate above 50% for the last of the tested games, Awari, a special approach handling draws was introduced, involving the maintenance of a second set of (dis)proof numbers.

3 Generalized Proof-Number MCTS

In this section, we present the GPN-MCTS algorithm and discuss its main features.

3.1 Proof Numbers per Player

The goal of the PN-UCB formula is to bias exploration towards subtrees that can be proven quickly. Thus, a reasonable assumption is that each player is interested in their own win. As mentioned before, this is not the approach taken by the PN-MCTS version described in [14]. In that work, all (dis)proof numbers are computed for the perspective of the root player, with proof numbers being used to guide selection in OR nodes, and disproof numbers in AND nodes. This approach results in asymmetry in behavior in particular in games where draws can occur, as the root player attempts to prove their wins, whereas the opponent only attempts to disprove the root player's wins (i.e., attempts to prove any mix of drawing and winning for themselves).

We propose a different approach, which consists of tracking separate proof numbers *per player* in each node of the search tree. This reduces code complexity in the sense that we no longer need to encode separate logic for handling disproof numbers, and the search behaviour becomes more symmetric in that each player attempts to prove their own wins.

This modification influences how the algorithm perceives AND/OR proof nodes. Let i be a game tree node, and p the player to move in that node. To prove a win, it is enough that any of the available paths leads to a win, so it is an OR node *for the player* p. From the perspective of any players other than p, the same node is treated as an AND node, in which all children must be proven for the node to be considered proven.

The advantage of the approach with a single proof number per player is that it naturally generalizes for games with more than two players. Although the proof number trees are no longer composed of alternating AND/OR layers, the way the algorithm behaves remains unchanged. For each player, when computing their proof tree, the first-player player's nodes are OR nodes, and all the remaining ones are AND nodes.

That way, the algorithm still assumes that every player is interested in proving their own win. However, proving a win for a given player still requires a paranoid assumption about the strategies of their opponents. Thus, our algorithm in this part behaves as Paranoid Proof-Number Search [22], a generalization of Proof-Number Search allowing to prove games with any number of players.

3.2 UCT-PN Formulas

The purpose of PN-term in the UCT-PN formula is to bias search towards nodes that are potentially easier to prove. However, the actual strength of this bias and how it depends on the actual proof numbers, is a critical point to discuss. Formula (3), using rank ordering of values, was proposed by [11, 14] with a justification that the magnitudes of differences amongst (dis)proof numbers are not meaningful.

We suspect that throwing away information about the actual (dis)proof values is a potential waste, and decided to search for possible alternatives. In fact, we believe that there are many functions that could serve this role, which may be simpler to implement and computationally cheaper than PNRank. This may be of special importance, as the problem of the PNRank formula is its inefficiency. The need to sort all children's values every time a (dis)proof number is updated to calculate proper rank is a significant computational effort. Note that this default behavior may be optimized: either by taking advantage of partial ordering and only fixing the position of the updated element (in $O(|\mathcal{I}|)$) or by using a method described in Section 3.2.5.

In this paper, we introduce two alternative formulas that can be used instead: PNMax and PNSum. We think both represent natural approaches when trying to take into account the actual proof values and making the spread proportionally rather than evenly.

3.2.1 Expected Properties

We begin by establishing the desired properties of a proof-number-based bias function, so it will nicely fit the framework, and its type could be treated as a parameter of the GPN-MCTS.

Thus, in line with the structure of (3), the proper formula should be a function taking a node i, and a set of nodes $\mathcal{I}$ (being i and all its siblings) and returning a number in $[0, 1]$, with larger values representing a greater bias towards selecting the node.

Additionally, we define a set of conditions that we argue a proper bias formula should meet. Let $\mathrm{pn}(i)$ be the proof number of a node i.

(a) Formula should return 0 if it is impossible to prove a node ($\mathrm{pn}(i)$ is ∞).
(b) If for any $j \in \mathcal{I}$, $\mathrm{pn}(j)$ is infinite, the values for all finite children should be strictly positive.
(c) If $\mathrm{pn}(i)$ equals $\mathrm{pn}(j)$, then the output for i and j, given the same $\mathcal{I}$ should also be equal.
(d) (optional) Formula should always return 1 for the lowest finite $\mathrm{pn}(i)$ value among $\mathcal{I}$.

The invariants above ensure that the actual behavior of the bias formula aligns with the natural expectations. For the PNRank formula defined as (3), the first three conditions hold.

3.2.2 PNMax

The first of the newly proposed approaches is to scale values with respect to the range of (finite) proof values among the children. This bias formula, named PNMax (4), requires calculating the minimal and maximal proof numbers among all children, which is significantly less computationally expensive than sorting.

The functions maxf and minf compute, respectively, the maximum and minimum proof values among the given set of nodes, taking into account only finite values. Thus, $\mathrm{maxf}(\mathcal{I}) = \max_{j \in \mathcal{I} \wedge \mathrm{pn}(j) \neq \infty} \mathrm{pn}(j)$.

$$\mathrm{PNMax}(i, \mathcal{I}) = \begin{cases} 0 & \text{if } \mathrm{pn}(i) \text{ is } \infty \\ 1 - \frac{\mathrm{pn}(i) - \mathrm{minf}(\mathcal{I})}{1 + \mathrm{maxf}(\mathcal{I}) - \mathrm{minf}(\mathcal{I})} & \text{otherwise} \end{cases} \tag{4}$$

For this formula, all four conditions are satisfied.

3.2.3 PNSum

As an alternative, we propose a bias formula that spreads values proportionally with respect to the sum of all finite proof values among the children. For this formula, PNSum (5), all conditions except (d) are met.

$$\mathrm{PNSum}(i, \mathcal{I}) = \begin{cases} 0 & \text{if } \mathrm{pn}(i) \text{ is } \infty \\ 1 - \frac{\mathrm{pn}(i)}{1 + \sum_{j \in \mathcal{I} \wedge \mathrm{pn}(j) \neq \infty} \mathrm{pn}(j)} & \text{otherwise} \end{cases} \tag{5}$$

The unique property of PNSum is that its behavior is far more strongly correlated with the branching factor of the parent node. than in the other tested formulas. This, depending on the game and the test setting, may turn out as a benefit or a liability.

3.2.4 C_{pn} Constant Tuning

Previous research regarding PN-MCTS was somewhat based on the assumption that the C_{pn}constant is universal, and similarly to the MCTS C constant, the same value can be used across a variety of games. Thus, the relation between C_{pn} and each tested game was not properly studied. As our approach introduces alternative functions to serve as UCT-PN formulas, the need for game- and formula-based tuning seems even more important.

Although the experiments and associated discussion are presented in Section 4.1, we think it is worth stating here, as an essential part of the description of the algorithm that indeed, the C_{pn}value has a crucial impact on the performance of GPN-MCTS. Moreover, the peak performance may occur at every point of the spectrum of tested values, as well as the peak for different UCT-PN formulas may be achieved by different C_{pn} values for the same game.

3.2.5 Backpropagation Optimization

All three UCT-PN bias formulas share the same characteristic: their value for a given node depends on all children of the same parent. In case of PNMax and PNSum, these are factors that can be associated with the parent node and simply stored there as precomputed values, but for PNRank the formula is more complicated and strongly benefits computing UCT-PN value for all children at once.

This, however, complicates the usual backpropagation step of PNS. Originally, it is a cheap operation requiring an update of a parent value if any child value was modified. Now, apart from just a proof number, we also need to take into account a UCT-PN value, and thus, all children of the parent should be updated, which is significantly more expensive, especially for PNRank.

However, we can at least partially mitigate that cost. The main observation is that not all values that would be changed during backpropagation will actually be called by the UCT-PN during the MCTS selection step. Thus, in our implementation, we optimized this step by setting the *needRecalc* flag on a node with updated proof number, and do not precompute UCT-PN value for this parent nor its siblings. UCT-PN formula is computed in a call-by-need manner, only if a *needRecalc* node is encountered during the selection phase (and after, the flag is set to false).

3.3 Mobility-based Initialization

Most classic improvements of PNS are focused on handling the issue of memory consumption when attempting to solve the game tree [6, 17], so they have no application when proof numbers are used as a selection bias inside the game-playing algorithm.

However, the *mobility initialization* enhancement [24] can be straightforwardly applied to GPN-MCTS. The idea is to initialize unknown leaf nodes in a more elaborate way than the one described in Section 2.2. In an AND node, the proof number can be set to the number of legal moves in this node. (In PNS, the disproof numbers are initialized that way in OR nodes.) This optimization improves the quality of (dis)proof numbers, as it works as one-step lookahead for computing the estimated sizes of the subtrees to prove.

3.4 Score Bounded GPN-MCTS

In practical implementations, algorithms like MCTS are nearly never applied in a vanilla format. Usually, the resulting algorithm consists of the union of a few general enhancements plus game-dependent improvements. For this reason, one of our goals was to perform tests of the proposed GPN-MCTS using a well-established advanced algorithm setup and see if the improved results carry over. We decided to put as our baseline an MCTS version with two improvements. One is simply a standard technique of tree reuse [7], and the other is Score Bounded MCTS [8].

Score-Bounded Monte-Carlo Tree Search (SB MCTS) is an extension of MCTS Solver [27] that can handle draws, and is generalized to games with many outcomes, as well as games with more than two players. Subtree-solving is probably the only line of MCTS enhancements that may be considered obligatory, as it comes with nearly no drawbacks. It is easy to implement, has a negligible computational cost, and more often than not increases the playing strength of an agent when applied to zero-sum games.

Proof-Number Search and Score-Bounded MCTS are based on a similar premise; in both cases, we are tracking which subtrees of the search tree are solved. On the one hand, PNS stores data allowing it to

predict which subtrees can be solved with less effort, the information SB MCTS is lacking. On the other hand, SB MCTS requires only two values (lower bound, upper bound) per tree node to store guaranteed payoffs, regardless of the number of possible outcomes in the game. For the purpose of biasing UCB in PN-MCTS, this would require a number of proof trees linear with respect to possible game outcomes. Such an approach was proposed in [14], introducing a second PN tree to handle draws. (Note that keeping just *attracting outcome* as in Multiple-Outcome PNS [21] does not work when applied to dynamic in-game search.)

We argue that using a well-established and more general solution, such as the Score Bounded extension, is a better option than further extending PN-MCTS into overlapping tasks of skipping solved subtrees and biasing a final move selection, as intended by [14]. Although PN trees can be used that way, it is inefficient, and the resulting code is less manageable.

The results (shown in Section 4.2) confirm that GPN-MCTS works well when combined with the Score Bound method, and there is no significant loss of quality between results on GPN-MCTS vs. MCTS and Score Bounded GPN-MCTS vs. Score Bounded MCTS.

3.5 The GPN-MCTS Algorithm

The pseudocode for a single iteration of GPN-MCTS is shown as Alg. 1. The overall frame of the algorithm follows the MCTS definition from [7], and uses the same terminology when possible. For example, DEFAULTPOLICY(s) encodes the standard simulation for the given game state to a terminal state.

A flag *needRecalc* of a node is the one introduced by optimization described in Section 3.2.5, and is set to false inside the UPDATECHILDRENPNSCORES() procedure. The details of this procedure are omitted, as it just calculates selection bias values of the children nodes for the node's moving player, according to the formulas from Section 3.2.

BESTUCTPNCHILD returns a child that maximizes the value of the UCT-PN Formula (2), with a comment that other bias functions, instead of PNRank might be encoded there.

The BACKUP(v_l, Δ) function consists of two separate parts. One is the standard MCTS backpropagation. The other is based on *updateAncestors* procedure (c.f. [12]) to backup proof numbers in an optimized way (early stop when no change is detected).

Finally, UPDATEPROOFNUMBER(p), updates the proof number in a node for a given player according to the PNS *setProofAndDisproofNumbers* procedure. The main differences here is that we update only proof numbers and that the OR/AND node distinction is based on whether p is a player performing a move at that node or not. A *proofWinner* in a leaf node is either Unknown if the associated game state is not terminal, or the player that won the game otherwise.

Algorithm 1 The GPN-MCTS Algorithm

```
Input: v0 – current root node of the GPN-MCTS tree
function GPN-MCTS-ITERATION(v0 )
    vl ← TREEPOLICY(v0)
    Δ ← DEFAULTPOLICY(vl.STATE())
    BACKUP(vl, Δ)

Input: v – root node of the GPN-MCTS tree
function TREEPOLICY(v)
    while v.STATE().ISNOTTERMINAL() do
        if v.needRecalc then v.UPDATECHILDRENPNSCORES()
        if v.ISNOTFULLYEXPANDED() then
            return EXPAND(v)
        else
            v ← v.BESTUCTPNCHILD()
    return v

Input: v – last node of iteration in GPN-MCTS tree
function EXPAND(v)
    a ← RANDOMELEMENT(v.UNTRIEDMOVES())
    v' ← CREATENODE(v.STATE().APPLY(a))
    for all p ∈ PLAYERS() do v'.UPDATEPROOFNUMBER(p)
    v.ADDNODE(v', a)
    return v'

Input: vl – last node of iteration in GPN-MCTS tree
Input: Δ – final scores for each player of iteration
function BACKUP(vl, Δ)
    v ← vl
    while v ≠ None do
        v.scoreSum ← v.scoreSum + Δ[leaf.player]
        v.iterations ← v.iterations + 1
        v ← v.PARENT()
    for all p ∈ PLAYERS() do
        v ← vl
        c ← True
        while c and v ≠ None do
            c ← v.UPDATEPROOFNUMBER(p)
            if c then v.needRecalc ← true
            v ← v.PARENT()

Input: p – player for whom the update takes place
function NODE.UPDATEPROOFNUMBER(p)
    if ISNOTEXPANDED() then
        if proofWinner = Unknown then proofNumber ← 1
        else if proofWinner = p then proofNumber ← 0
        else proofNumber ← ∞
        return True
    oldProof ← proofNumber
    if p = nodePlayer then                          ▷ PNS OR node
        proofNumber ← ∞
        for all child ∈ CHILDREN() do
            proofNumber ← MIN(proofNumber, child.proofNumber)
    else                                            ▷ PNS AND node
        proofNumber ← 0
        for all child ∈ CHILDREN() do
            proofNumber ← proofNumber + child.proofNumber
    return proofNumber ≠ oldProof
```

4 Experiments

The experiments have been conducted using the Ludii general game-playing system, which provides an environment for developers to test their implementation of general game-playing agents [19]. It was chosen as it contains over 1,000 games described in its game description language, and implementations of various standard algorithms and enhancements (such as several variants of MCTS), with a single, unified API for the development of AI agents.

The presented GPN-MCTS algorithm has been implemented as an enhancement of the agents available in Ludii, and merged into the official Ludii codebase.[1] Experiments were run using Ludii version 1.3.14. Two versions of GPN-MCTS are available: with Score Bounded enhancement and without. If not stated otherwise, the experiments are performed by playing with Score Bounded GPN-MCTS with tree reuse against Score Bounded MCTS with tree reuse. For both agents, the MCTS C parameter is set to $\sqrt{2}$. Player positions are swapped in all tests so that the agents play both sides equally often, and draws count as half wins. The experiments were performed on different machines; however, for any game, all results regarding this game were computed using the same hardware, which makes them comparable. For every result, if any error margins are

[1] https://github.com/Ludeme/Ludii

presented, they represent a 95% confidence interval.

4.1 UCT-PN Bias Formulas and C_{pn}

The main experiments were conducted on a set of two-player, zero-sum board games, including the games used for experiments in [14]. Our aim is to test the behavior of each bias formula variant (PNRank, PNMax, PNSum) and the influence of C_{pn} constant on the improvements over the baseline agent. We tested $C_{pn} \in \{0.0, 0.1, 0.5, 1.0, 2.0, 5.0\}$ to be consistent with the previous research regarding PN-MCTS. Although this range of parameters may be insufficient to provide the exact highest winrate available for a given game and UCT-PN variant, it provides a good estimation of them, as well as of the general behavior of the winrate function for these settings. Each of the tests consists of 500 matches with 1 second per turn. The results are presented in Table 1.

GPN-MCTS achieved 80% win rate against the Score Bounded MCTS on 8 out of the 11 tested board games, in two cases (Ataxx and Lines of Action) even reaching 90%. For the remaining tested games, the results are also confident wins, with the lowest best score 63.2% obtained for Knighttrough.

The first observation is that the best C_{pn} values greatly differ for various games. For some games, the best value was the lowest one tested, and for some, the largest one (which suggests that even the better C_{pn} values can potentially be found outside of the tested range). Also, when looking at most of the games, the spread of the winrate between the best and worst choice of C_{pn} is vast (in extreme case, for Lines of Action 8×8, the winrate degraded from 82.5% for $C_{pn} = 1$ to 25.9% for $C_{pn} = 5$). This is a strong indicator that picking a value that behaves best on average is not a good idea, and the results obtained that way may be very far from the optimal ones.

The second observation is based on comparing the behavior and results for the three tested bias formulas. There is no clear winner on which of the formulas is the best. Especially as often the peak winrates are not far from each other, and local C_{pn} tuning can possibly reverse the results. (And sometimes, especially for PNSum which has a tendency to have peaks for larger C_{pn} values, it may be even further outside of the tested parameter range.) Generally, PNRank seems like a safe choice, obtaining the best results, or relatively close to best, for most games. In some cases, like for Minishogi, this bias formula shows a clear advantage over the others. In others, e.g., for Surakarta, PNMax is clearly better with over 20 percent point advantage. Although for many games PNSum has lower results, there are also cases (Reversi) when it performs the best out of three.

Summarizing, if the game is susceptible to PN-based exploration, then using any of the formulas should lead to improvement, but the actual amount of this improvement depends on the particular pick and may differ significantly.

4.2 Influence of Score Bounded on GPN-MCTS

For some of the games and the PNRank formula, we run experiments without the Score Bounded enhancement (for both GPN-MCTS and MCTS). Thus, we can analyze if the PN-based works as well as the vanilla algorithm as with a union with other improvements. The potential problem of any improvement to any complex AI algorithm is that although it works standalone, its benefits are degraded when applied with other improvements. Also, in more real-world scenarios, we cannot expect our opponent to be a basic implementation, so the enhancements should be able to show improvements also against a more advanced opponent.

The results of our comparison show that, in general, improving both GPN-MCTS and the opponent with the Score Bounded extension keeps the win rates in similar ranges. Note, however, that the PN-MCTS, as its core goal is to bias exploration towards potentially fast-to-prove nodes, is especially suited to work well with the Score Bounded extension. This may explain why often the results including SB have a tendency to be (slightly) better than for vanilla MCTS.

4.3 Overhead

The requirement of maintaining PNS-related structures on top of the standard MCTS tree implies that PN-MCTS is usually slower than the pure MCTS. Thus, it will generally perform fewer iterations within any given time budget. For this reason, all experiments in this paper use time-based budgets, which we consider a fair comparison, as simulation-based budgets ignore the factor of computation overhead. GPN-MCTS is developed as an extension of the MCTS implementation provided by the Ludii system to ensure that any difference in performance is solely due to the implementation of the proposed enhancement.

To measure the overhead, we included $C_{pn} = 0$ in the results of Table 1. This winrate value represents the match between two Score Bounded MCTS algorithms (in terms of behavior), but with one spending additional computation time on managing proof numbers and UCT-PN values. Most of these win rates are within the confidence interval distance to 50%, which means that the computational overhead of GPN-MCTS seems to be small enough not to affect the expected results.

5 Conclusion

The experimental results show that GPN-MCTS is a rather impactful MCTS enhancement, and we argue that it has all the reasons to be considered as a staple improvement to implement alongside Solver/MAST/RAVE and other classic developments.

It is relatively simple to implement, based on a classic, well-described algorithm. It pairs with obligatory Score Bounded MCTS enhancement. It is easy to test whether its application will be beneficial for a given game – comparing an algorithm without extension with any bias formula and one or two small C_{pn} values should give the right indication. And the potential gains can range from a trustworthy 60% win up to a 90% decisive victory.

In this paper, we focused our experiments on two-player games, despite the fact that GPN-MCTS can actually handle games with more players. Although multi-player games are an interesting challenge, so far integrating solving capabilities in MCTS has led only to limited improvements [18]. Therefore, we have considered the application of GPN-MCTS in these domains as future research.

Our preliminary experiments have indicated that there are games where GPN extension makes no visible impact (Pentago, Pentalah) or even worsens the results (Connect Four, Diagonals). This ought to be expected, as Proof-Number Search does not always work when applied in a game-agnostic manner. PNS takes advantage of situations when there are deep, narrow winning paths. For some games, introducing domain knowledge [12] is required to shape the search tree in such a way that narrow and forced paths emerge. Testing if transferring such knowledge to GPN-MCTS framework is possible and it will positively influence results for such games is one of the promising paths for future work.

Table 1: The results of GPN-MCTS versus MCTS. Variant including SB means both algorithms use Score Bounded and tree reuse, otherwise it is just tree reuse. (500 games, 1s per turn).

Variant	C_{pn} value					
	0.0	0.1	0.5	1.0	2.0	5.0
Ataxx						
PNRank	45.9 ± 4.37	58.0 ± 4.32	82.3 ± 3.33	87.0 ± 2.95	89.4 ± 2.70	**89.5** ± 2.67
PNRank+SB	50.1 ± 4.38	60.6 ± 4.27	85.3 ± 3.09	89.2 ± 2.72	91.4 ± 2.46	**92.0** ± 2.36
PNMax+SB	49.2 ± 4.38	65.3 ± 4.17	**92.4** ± 2.33	92.3 ± 2.33	91.0 ± 2.51	89.1 ± 2.71
PNSum+SB	45.6 ± 4.36	56.5 ± 4.33	55.7 ± 4.34	56.2 ± 4.34	61.6 ± 4.25	**71.2** ± 3.93
Awari						
PNRank	49.0 ± 4.11	**72.9** ± 3.46	70.4 ± 3.53	61.2 ± 3.61	50.9 ± 3.95	43.0 ± 4.27
PNRank+SB	48.1 ± 3.97	71.0 ± 3.53	**74.3** ± 3.32	62.2 ± 3.63	50.4 ± 3.92	43.2 ± 4.21
PNMax+SB	50.9 ± 3.95	**78.2** ± 3.21	69.8 ± 3.50	56.3 ± 4.28	45.5 ± 4.27	45.1 ± 4.25
PNSum+SB	49.8 ± 4.04	62.2 ± 3.73	78.8 ± 3.28	**79.7** ± 3.07	77.2 ± 3.34	67.9 ± 3.77
Knightthrough						
PNRank	48.0 ± 4.38	50.6 ± 4.39	52.2 ± 4.38	52.8 ± 4.38	62.6 ± 4.25	**74.0** ± 3.85
PNRank+SB	53.2 ± 4.38	45.8 ± 4.37	46.2 ± 4.37	56.3 ± 4.33	54.8 ± 4.37	**59.2** ± 4.31
PNMax+SB	46.8 ± 4.38	56.6 ± 4.35	50.8 ± 4.39	62.8 ± 4.24	**63.2** ± 4.21	54.4 ± 4.37
PNSum+SB	49.0 ± 4.39	50.0 ± 4.39	56.0 ± 4.36	52.0 ± 4.38	55.8 ± 4.36	**60.0** ± 4.30
Lines of Action (7×7)						
PNRank	52.1 ± 4.38	67.4 ± 4.11	85.0 ± 3.13	87.0 ± 2.95	**88.4** ± 2.81	80.2 ± 3.49
PNRank+SB	47.6 ± 4.37	67.1 ± 4.12	82.8 ± 3.31	91.3 ± 2.47	**91.8** ± 2.39	83.6 ± 3.25
PNMax+SB	52.6 ± 4.38	66.3 ± 4.14	81.3 ± 3.42	**84.3** ± 3.14	68.6 ± 4.06	65.8 ± 4.13
PNSum+SB	54.6 ± 4.37	55.1 ± 4.36	60.9 ± 4.28	64.9 ± 4.18	67.9 ± 4.08	**69.3** ± 4.04
Lines of Action (8×8)						
PNRank	50.0 ± 4.39	56.4 ± 4.35	83.0 ± 3.30	**85.8** ± 3.06	80.3 ± 3.48	63.9 ± 4.21
PNRank+SB	51.8 ± 4.38	63.6 ± 4.22	83.8 ± 3.23	**87.4** ± 2.91	81.9 ± 3.37	55.2 ± 4.35
PNMax+SB	50.8 ± 4.39	55.8 ± 4.36	68.8 ± 4.07	**82.5** ± 3.33	63.7 ± 4.21	25.9 ± 3.83
PNSum+SB	48.0 ± 4.38	54.5 ± 4.33	52.4 ± 4.38	55.6 ± 4.41	55.5 ± 4.75	**64.3** ± 4.48
Los Alamos chess						
PNRank	53.2 ± 4.10	71.9 ± 3.76	**85.5** ± 2.94	82.4 ± 3.22	80.5 ± 3.36	80.5 ± 3.39
PNRank+SB	47.3 ± 4.16	71.7 ± 3.73	81.0 ± 3.33	**84.5** ± 3.02	80.5 ± 3.38	79.3 ± 3.43
PNMax+SB	48.7 ± 4.16	71.1 ± 3.74	73.4 ± 3.73	76.5 ± 3.52	**80.5** ± 3.35	78.7 ± 3.45
PNSum+SB	49.7 ± 4.11	60.3 ± 4.10	64.5 ± 3.95	72.7 ± 3.75	**77.5** ± 3.46	77.0 ± 3.54
Minishogi						
PNRank	47.4 ± 4.38	55.0 ± 4.37	67.8 ± 4.17	**69.2** ± 4.05	56.8 ± 4.35	45.6 ± 4.37
PNRank+SB	45.2 ± 4.37	54.4 ± 4.37	**67.8** ± 4.10	66.2 ± 4.15	63.8 ± 4.22	46.0 ± 4.37
PNMax+SB	52.4 ± 4.38	49.6 ± 4.39	**51.2** ± 4.39	34.6 ± 4.17	31.6 ± 4.08	36.2 ± 4.22
PNSum+SB	49.6 ± 4.39	52.4 ± 4.38	57.0 ± 4.34	**59.4** ± 4.31	51.8 ± 4.38	48.8 ± 4.39
Reach Chess						
PNRank	44.4 ± 4.36	58.6 ± 4.32	64.8 ± 4.19	66.2 ± 4.15	71.8 ± 3.95	**76.8** ± 3.70
PNRank+SB	50.2 ± 4.39	62.0 ± 4.26	64.6 ± 4.20	72.2 ± 3.93	**79.0** ± 3.57	78.0 ± 3.64
PNMax+SB	49.2 ± 4.39	58.8 ± 4.32	**70.2** ± 4.01	65.2 ± 4.18	69.8 ± 4.03	55.2 ± 4.36
PNSum+SB	46.2 ± 4.37	57.4 ± 4.34	59.8 ± 4.30	63.6 ± 4.22	64.4 ± 4.20	**68.2** ± 4.09
Reversi						
PNRank	48.7 ± 4.28	65.1 ± 4.11	**71.6** ± 3.89	49.9 ± 4.32	43.4 ± 4.33	42.5 ± 4.32
PNRank+SB	50.0 ± 4.32	**71.0** ± 3.87	69.4 ± 3.99	53.2 ± 4.34	42.4 ± 4.27	37.2 ± 4.22
PNMax+SB	50.0 ± 4.24	**63.1** ± 4.16	59.0 ± 4.26	45.2 ± 4.30	36.2 ± 4.18	35.0 ± 4.15
PNSum+SB	49.5 ± 4.28	55.0 ± 4.27	60.7 ± 4.21	71.7 ± 3.86	**76.7** ± 3.64	64.1 ± 4.14
Skirmish						
PNRank	51.0 ± 4.25	60.0 ± 4.14	**60.3** ± 4.14	49.5 ± 4.26	54.7 ± 4.29	50.6 ± 4.25
PNRank+SB	50.6 ± 4.16	**62.4** ± 4.07	62.3 ± 4.11	61.7 ± 4.17	60.8 ± 4.17	55.8 ± 4.29
PNMax+SB	51.5 ± 4.20	65.1 ± 4.00	**65.3** ± 4.05	58.0 ± 4.17	56.2 ± 4.26	39.0 ± 4.18
PNSum+SB	50.5 ± 4.23	49.9 ± 4.19	56.4 ± 4.11	55.3 ± 4.12	63.4 ± 4.38	**63.8** ± 4.03
Surakarta						
PNRank	49.6 ± 4.34	**60.2** ± 4.25	59.8 ± 4.25	53.8 ± 4.35	36.8 ± 4.17	13.7 ± 2.97
PNRank+SB	49.5 ± 4.34	60.5 ± 4.23	**60.6** ± 4.25	59.1 ± 4.26	46.7 ± 4.33	15.2 ± 3.11
PNMax+SB	48.6 ± 4.36	65.1 ± 4.13	**82.0** ± 3.33	68.4 ± 4.05	44.4 ± 4.32	22.4 ± 3.67
PNSum+SB	47.8 ± 4.35	52.5 ± 4.63	54.3 ± 4.50	50.9 ± 4.35	54.6 ± 4.34	**57.7** ± 4.27

Acknowledgements

This article is based on the work of COST Action CA22145 – GameTable, supported by COST (European Cooperation in Science and Technology).

This research was supported in part by the National Science Centre, Poland, under project number 2021/41/B/ST6/03691 (Jakub Kowalski).

References

[1] L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-Number Search. *Artificial Intelligence*, 66(1):91–123, 1994.

[2] B. Arneson, R. B. Hayward, and P. Henderson. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-Time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47(2–3):235–256, 2002.

[4] H. Baier and M. H. M. Winands. MCTS-Minimax Hybrids. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2): 167–179, 2015. doi: 10.1109/TCIAIG.2014.2366555.

[5] Y. Björnsson and H. Finnsson. CadiaPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[6] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik. The PN^2-search algorithm. In *Advances in Computer Games 9*, pages 115–132. Universiteit Maastricht, Maastricht, The Netherlands, 2001.

[7] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[8] T. Cazenave and A. Saffidine. Score Bounded Monte-Carlo Tree Search. In *Computers and Games*, volume 6515 of *LNCS*, pages 93––104, 2011.

[9] G. M. J.-B. Chaslot, M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.

[10] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games (CG 2006)*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83, 2007.

[11] E. Doe, M. H. M. Winands, D. J. N. J. Soemers, and C. Browne. Combining Monte-Carlo tree search with proof-number search. In *Proceedings of the 2022 IEEE Conference on Games*, pages 206–212, 2022.

[12] A. Kishimoto, M. H. M. Winands, M. Müller, and J.-T. Saito. Game-Tree Search Using Proof Numbers: The First Twenty Years. *ICGA Journal*, 35(3):131–156, 2012.

[13] L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, Berlin, Heidelberg, 2006.

[14] J. Kowalski, E. Doe, M. H. M. Winands, D. Górski, and D. J. N. J. Soemers. Proof Number Based Monte-Carlo Tree Search. *IEEE Transactions on Games*, 17(1):148–157, 2024.

[15] M. Lanctot, M. H. M. Winands, T. Pepels, and N. R. Sturtevant. Monte Carlo Tree Search with Heuristic Evaluations using Implicit Minimax Backups. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014*, pages 341–348, 2014.

[16] R. J. Lorentz. Amazons Discover Monte-Carlo. In *Computers and Games (CG 2008)*, volume 5131 of *Lecture Notes in Computer Science*, pages 13–24, 2008.

[17] A. Nagai. A new AND/OR tree search algorithm using proof number and disproof number. In *Proceedings of Complex Games Lab Workshop*, pages 40–45. ETL, Tsukuba, Japan, 1998.

[18] J. A. M. Nijssen and M. H. M. Winands. Enhancements for multi-player Monte-Carlo tree search. In H. J. van den Herik, H. Iida, and A. Plaat, editors, *Computers and Games (CG 2010)*, volume 6515 of *Lecture Notes in Computer Science*, pages 238–249. Springer Berlin Heidelberg, 2011.

[19] É. Piette, D. J. N. J. Soemers, M. Stephenson, C. F. Sironi, M. H. M. Winands, and C. Browne. Ludii – The Ludemic General Game System. In *Proceedings of the 24th European Conference on Artificial Intelligence (ECAI 2020)*, volume 325 of *Frontiers in Artificial Intelligence and Applications*, pages 411–418, 2020.

[20] O. Randall, M. Müller, T.-H. Wei, and R. Hayward. Expected work search: Combining win rate and proof size estimation. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, IJCAI '24, pages 7003 – 7011, 2024.

[21] A. Saffidine and T. Cazenave. Multiple-outcome Proof Number Search. In *ECAI*, pages 708–713. 2012.

[22] J.-T. Saito and M. H. M. Winands. Paranoid Proof-Number Search. In *Proceedings of the Computational Intelligence and Games Conference (CIG'10)*, pages 203–210, 2010.

[23] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the Game of Go Without Human Knowledge. *Nature*, 550: 354–359, 2017.

[24] H. J. van den Herik and M. H. M. Winands. Proof-Number Search and Its Variants. In *Oppositional Concepts in Computational Intelligence*, pages 91–118. 2008. ISBN 978-3-540-70829-2.

[25] M. H. M. Winands and Y. Björnsson. $\alpha\beta$-based Play-outs in Monte-Carlo Tree Search. In *2011 IEEE Conference on Computational Intelligence and Games (CIG 2011)*, pages 110–117. IEEE, 2011.

[26] M. H. M. Winands, J. W. H. M. Uiterwijk, and H. J. van den Herik. PDS-PN: A new proof-number search algorithm: Application to Lines of Action. In *Computers and Games*, volume 2883 of *Lecture Notes in Computer Science (LNCS)*, pages 170–185, 2003.

[27] M. H. M. Winands, Y. Björnsson, and J.-T. Saito. Monte-Carlo Tree Search Solver. In *Computers and Games (CG 2008)*, volume 5131 of *Lecture Notes in Computer Science (LNCS)*, pages 25–36, 2008.

[28] M. H. M. Winands, Y. Björnsson, and J.-T. Saito. Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):239–250, 2010.

[29] M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk. Monte Carlo Tree Search: A review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, 2023. doi: 10.1007/S10462-022-10228-Y.