Generation and Evaluation of Exploration Terms for Monte Carlo Tree Search

Tristan Cazenave *
LAMSADE, Université Paris Dauphine - PSL, Paris, France
E-mail: tristan.cazenave@dauphine.psl.eu

Abstract. Monte Carlo Tree Search has shown good results for many difficult problems. We propose to automatically generate mathematical expressions that are used as exploration terms for Monte Carlo Tree Search (MCTS) algorithms, and we evaluate the resulting expressions in the game of Go. We use a systematic generation of small exploration terms. This is in contrast to our initial conference paper, which used Monte Carlo Search to generate exploration terms for Sequential Halving Using Scores (SHUSS). In this journal paper, we extend the work to automatically design three exploration terms: the exploration term near the leaves of Prior Upper Confidence for Trees (PUCT), the exploration term at the root of PUCT and the SHUSS exploration term. All three resulting exploration terms improve upon standard PUCT in the game of Go.

Keywords: Monte Carlo Tree Search, Exploration Term, Computer Go

1. INTRODUCTION

Monte Carlo Tree Search (Coulom, 2006; Kocsis and Szepesvári, 2006) is a well-known family of algorithms designed for the game of Go and has since been applied to many different combinatorial problems (Browne et al., 2012; Świechowski et al., 2023).

Our goal in this paper is to automatically find exploration terms to improve Monte Carlo Tree Search. This is part of a long-standing goal of using Artificial Intelligence to improve Artificial Intelligence (Pitrat, 2008). Our approach is to generate and evaluate all possible exploration terms below a given size and to retain those that best reproduce the choices of long PUCT searches.

The usual way to design an exploration term is to perform a theoretical analysis (Auer et al., 2002). We take another empirical approach. We automatically generate exploration terms and retain those that have the greatest accuracy in a test set. This is a simpler approach. However, exploration terms that work well in practice and surpass those found through theoretical analysis can be identified.

Another family of approaches to the automatic design of MCTS components relies on evolutionary computation, and in particular, on Genetic Programming. Cazenave (2007) evolved Monte Carlo Tree Search algorithms, improving UCT and RAVE for the game of Go. Bravi et al. (2017) used Genetic Programming to evolve game-specific alternatives to UCB for general video game playing, finding selection policies tailored to individual games that outperform standard UCB1. More recently, Ameneyro and Galván (2022) studied the effects of evolving the UCT selection policy and analyzed which structural patterns tend to emerge from evolution. Closer to the design of simulation heuristics than to selection policies, Holmgård et al. (2018) evolved heuristics for MCTS to drive automated playtesting with procedural personas, and Alhejali and Lucas (2013) used Genetic Programming

*Corresponding author. E-mail: tristan.cazenave@dauphine.psl.eu.

to evolve playout heuristics for an MCTS Ms. Pac-Man agent. Compared to these evolutionary approaches, our method exhaustively enumerates small expressions rather than evolving them, evaluates each candidate against a cached search dataset rather than through competitive play or full simulations, and does not rely on tuning a fitness function based on game outcomes. As a consequence, the evaluation of a single expression takes milliseconds rather than minutes or hours, which is what makes exhaustive enumeration over the constrained space tractable.

Monte Carlo Tree Search, combined with Deep Reinforcement Learning, has been used to improve algorithms. AlphaTensor discovered new, fast matrix multiplication algorithms while playing the tensor game (Fawzi et al., 2022). AlphaTensor, as well as other Monte Carlo Tree Search algorithms, has been used for quantum circuit optimization (Wang et al., 2023; Rosenhahn and Osborne, 2023; Ruiz et al., 2024; Dhankhar and Cazenave, 2026; Lipardi et al., 2025). New fast sorting algorithms were discovered thanks to Monte Carlo Tree Search with the AlphaDev system (Mankowitz et al., 2023).

Monte Carlo Search (Cazenave, 2010) and Monte Carlo Tree Search (Cazenave, 2013) have been used to discover mathematical expressions that maximize a given score function. This has been applied to different domains, including physics (Sun et al., 2022), finance (Cazenave and Hamida, 2015), and the automated design of functions (Illetskova et al., 2019).

Refinements of the Monte Carlo Search approach to mathematical expression discovery include incorporating actor-critic in Monte Carlo Tree Search for symbolic regression (Lu et al., 2021), using a grammar of Monte Carlo Search algorithms (Maes et al., 2013), controlling the size (Moudřík et al., 2017), and using a Large Language Model as a prior (Li et al., 2024).

The automated discovery of optimization algorithms through symbolic program search has recently enabled the identification of a simple and effective optimization algorithm, Lion (evoLved sIgn mOmeNtum). Lion is more memory efficient than the widely used Adam optimizer, as it only keeps track of the momentum (Chen et al., 2024).

Beyond the discovery of individual algorithms, Large Language Models are increasingly used as general-purpose programming assistants that accelerate the development of AI systems themselves. Claude Code[1] is an agentic coding assistant developed by Anthropic that operates directly from the terminal, capable of reading and modifying codebases, running experiments, and iterating on results with minimal human intervention. A striking illustration of this trend is that the codebase of Claude Code is itself largely written by Claude Code: as of mid-2025, 90–95% of new Claude Code source code was reportedly produced by the tool itself (Flanagan, 2026). Such self-referential development blurs the line between researcher and research subject and exemplifies a broader movement in which AI systems contribute not only the algorithmic ideas — as in AlphaTensor, AlphaDev, or Lion — but also the engineering effort required to build and refine the next generation of AI tools, further closing the loop of Artificial Intelligence improving Artificial Intelligence.

Our work aligns with these applications of Artificial Intelligence to discover new Artificial Intelligence algorithms. Our goal is to automatically generate and evaluate mathematical expressions in order to discover new exploration terms for MCTS.

This journal version extends our earlier conference paper (Cazenave, 2024) as the scope is broader: the conference paper designed only the SHUSS exploration term, whereas this paper additionally designs an exploration term for low-visit nodes inside PUCT and an exploration term at the root of PUCT, with all three terms compared head-to-head against standard PUCT. Moreover, the empirical findings are stronger: the discovered near-leaves term improves on PUCT across the full range of search budgets

[1] https://www.anthropic.com/claude-code

we tested (128, 256, 512, and 1,024 tree descents), and we discovered a better SHUSS exploration term. The cached-search dataset is also released as a stand-alone artifact for further research.

Our contributions are as follows:

- An efficient method for empirically designing exploration terms in MCTS.
- The design of a dataset to discover exploration terms.
- Better exploration terms for PUCT near the leaves.
- Better exploration terms for PUCT at the root.
- A better way to select moves for SHUSS according to the priors given by the policy network.

We evaluate the proposed method in the game of Go, using a lightweight policy-value network trained on Katago self-play games. The discovered exploration terms are then validated against standard PUCT in head-to-head match play.

The second section presents various Monte Carlo Tree Search algorithms. The third section explains how we generate mathematical expressions for the exploration terms. The fourth section details how we evaluate exploration terms. The fifth section gives the experimental results. The sixth section concludes and discusses directions for future work.

2. MONTE CARLO TREE SEARCH

In this section, we present various Monte Carlo Tree Search algorithms, starting with PUCT, the most popular one, which is used in AlphaZero and is standard in computer game playing. We then present the Sequential Halving algorithm, as well as the related Sequential Halving Using Scores (SHUSS) algorithm.

2.1. PUCT

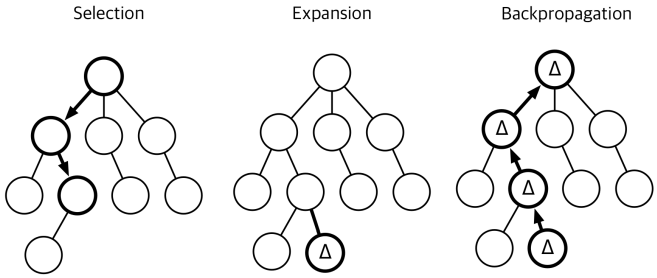

Fig. 1.

The three steps of MCTS. The first step is the tree descent using the exploration term to choose among the children. The second step is adding a new leaf associated to an evaluation of the state by the value network. The third step is updating the statistics in the tree with the evaluation.

Monte Carlo Tree Search was designed for computer Go and revolutionized computer Go (Coulom, 2006; Kocsis and Szepesvári, 2006), followed by computer game playing (Finnsson and Björnsson,

2008; Méhat and Cazenave, 2008). Within deep-reinforcement-learning applications of Monte Carlo Tree Search, PUCT has become the standard selection rule. This is the search algorithm used in AlphaGo (Silver et al., 2016), AlphaGo Zero (Silver et al., 2017), AlphaZero (Silver et al., 2018), and MuZero (Schrittwieser et al., 2020).

Figure 1 presents the three steps of MCTS. The principle of the algorithm is to memorize the states already explored, as well as the associated statistics for the possible actions in these states. When the algorithm encounters a state that has already been explored, it uses an exploration term to choose the next action to take. The average of previous evaluations associated with an action a in state s is $Q(s,a)$. The number of descents that have passed through state s is $N(s)$, and the number of descents in s that have taken action a is $N(s,a)$.

A neural network is used at the leaves of the tree to evaluate the leaves and calculate probabilities for the possible actions. The probability, as estimated by the neural network, that action a is the best in state s is $P(s,a)$.

The exploration term that is added to $Q(s,a)$ in PUCT is:

$$c_p \times P(s,a) \times \frac{\sqrt{N(s)}}{1+N(s,a)}$$

The constant c_p is a hyperparameter that must be tuned for each problem.

In the following, we use p for $P(s,a)$ the probability of action a in state s as given by the head of the neural network policy.

2.2. Sequential Halving

Sequential Halving (Karnin et al., 2013) is an algorithm that minimizes simple regret.

It has been used successfully as an alternative to UCB in Monte Carlo Tree Search, particularly as a replacement in the root node, with UCB used in the rest of the tree, or even in the entire tree with SHOT (Cazenave, 2015). It was applied to perfect information games (Pepels et al., 2014a; Cazenave, 2015) as well as to games of imperfect information (Pepels et al., 2015). Sequential Halving was also used as a root policy with Gumbel MuZero (Danihelka et al., 2021). The outputs of Sequential Halving at the root were used for reinforcement learning in the MuZero algorithm. Gumbel MuZero was successfully applied to Go, Chess, and Atari games.

Algorithm 1 presents the Sequential Halving algorithm used in SHUSS (Fabiano and Cazenave, 2021). This is the one we use in this paper with $\lambda = \frac{1}{2}$. The principle of the algorithm is to allocate the same number of evaluations to all the actions in the set of actions S_r. Then, select half of the actions in S_r that have the best empirical average. This best half constitutes S_{r+1}. The algorithm continues to allocate evaluations to the remaining actions and to select the best half until there is only one action remaining in S_R.

2.3. Sequential Halving Using Scores

SHUSS (Fabiano and Cazenave, 2021) is an improvement on Sequential Halving that uses a prior to enhance move selection at the root. In the case of a prior given by a neural network, it utilizes the classic Sequential Halving algorithm, which is restricted to a fixed number of moves that have the best priors.

Algorithm 1 Sequential Halving

Parameter: cutting ratio λ
Input: total budget T, set of arms S
$S_0 \leftarrow S$, $T_0 \leftarrow T$
$R \leftarrow$ number of rounds before $|S_R| = 1$
for $r = 0$ **to** $R - 1$ **do**
    $t_r \leftarrow \left\lfloor \frac{T_r}{|S_r| \cdot (R-r)} \right\rfloor$
    $T_{r+1} \leftarrow T_r - t_r |S_r|$
    sample t_r times each arm in S_r
    $S_{r+1} \leftarrow S_r$ deprived of the fraction $1 - \lambda$ of the worst arms
end for
Output: arm in S_R

3. GENERATION OF MATHEMATICAL EXPRESSIONS

Expression trees are represented as stacks in reverse polish notation. For example, the generated expression [+, s, *, h, log, p] corresponds to the exploration term $s + h \times log(p)$, where s is the sum of the evaluations starting with the move to evaluate, h is a hyperparameter to tune, and p is the prior for that move. The evaluation of an expression in reverse polish notation is algorithmically simple, as it uses a straightforward depth-first search.

In order to limit the size of the generated expressions, we maintain the number of open leaves of an incomplete expression. This is the total number of children of the atoms in the expression that are not yet associated with an atom. In the root node, the number of open leaves of the empty expression is 1. If, for example, a + atom is assigned to the root, then there is one open leaf fewer because of the assignment and two open leaves more because the + atom has two children that are not yet assigned. In order to generate expressions that are smaller than the maximum length, the legal moves function does not return atoms that, when added to the number of already assigned atoms, the number of open leaves, and the number of children of the atom, exceed the maximum length.

Algorithm 2 Generation of all the possible shapes

Parameter: maximum size $MaxSize$
Procedure GENERATESHAPE($shape, leaves, l$)
    if $leaves = 0$ **then**
        $l.append(shape)$
        return
    end if
    for $m \in [0, 1, 2]$ **do**
        if $len(shape) + leaves + m \leqslant MaxSize$ **then**
            $s_1 \leftarrow copy(shape)$
            $s_1.append(m)$
            $l_1 \leftarrow leaves - 1 + m$
            GENERATESHAPE (s_1, l_1, l)
        end if
    end for

The algorithm 2 generates all the expression shapes. A shape is a list of integers. Each integer in the list represents the number of children of the atom at this index in the expression. For example, the

shape [2,0,2,0,1,0] represents expressions that start with an atom with two children, followed by an atom with no children, and so on. The expression [+, s, *, h, log, p] follows this shape.

Algorithm 3 Generation of all the expressions of a given shape

```
1:  Procedure GENERATE(shape, expression, l)
2:      if len(shape) = len(expression) then
3:          l.append(expression)
4:          return
5:      end if
6:      n ← shape[len(expression)]
7:      for a ∈ atoms[n] do
8:          e_1 ← copy(expression)
9:          e_1.append(a)
10:         GENERATE (shape, e_1, l)
11:     end for
```

The algorithm 3 generates all the expressions that have a given shape. It uses a list of lists of atoms. The atoms in the first list are those that have no children. The atoms in the second list have exactly one child, and the atoms in the last list have two children. An example of such a list is atoms = [['s','h','p'],['sqrt','log','exp'],['+','-','*','/']].

The expressions we generate contain the usual operators: +, -, *, /, exp, log, sqrt. They also contain, as leaves: *total* the number of tree descents of the node plus one, p the prior of the move, *nb* the number of tree descents of the move plus one, and h a hyperparameter that is tested for values ranging from 0.1 to 1.0.

A consequence of allowing atoms to repeat is that an expression of length L may contain several occurrences of the hyperparameter h. We do not treat distinct occurrences of h as independent free variables. Instead, all occurrences of h within a given expression are tied to a single value, and the evaluation procedure performs a one-dimensional sweep over a fixed grid $H = \{0.1, 0.2, \ldots, 1.0\}$ of $|H| = 10$ candidate values, retaining the value that yields the best score. The number of evaluations of a single expression is therefore $|H| = 10$, regardless of how many times h appears in the expression. Treating each occurrence of h as an independent free parameter would have required $|H|^k$ evaluations for an expression with k occurrences of h, which would have made exhaustive enumeration over expressions of size up to 8 more costly. Tying all occurrences of h together also matches the standard practice in MCTS exploration terms, where a single exploration constant has to be tuned.

All possible exploration terms that are smaller than a given size are generated. In order to eliminate exploration terms that we consider inappropriate for PUCT, we use a function that verifies certain constraints on them. The constraints we impose on the generated expressions encode minimal structural priors about what a sensible PUCT-style exploration term looks like, and they substantially reduce the size of the search space. We require three properties. First, the expression must contain both $*$ and /. The role of an exploration term in PUCT is to balance the prior p against the visit count *nb*, which is fundamentally a multiplicative-divisive operation: the prior should reward moves that the policy considers strong, and the visit count should penalize moves that have already been explored. An expression built only from additive operators cannot express this trade-off in any meaningful way, since adding p and *nb* does not capture the inverse relationship between exploration bonus and visit count. The constraint that both $*$ and / appear ensures that the candidate expression has the structural ingredients to balance these two quantities. Second, increasing the prior p must increase the value of the expression. This monotonicity requirement reflects the intended semantics of an exploration term: a

move with a higher prior should be considered more attractive, all else being equal. Without this constraint, the search would consider expressions that punish moves the policy considers strong, which contradicts the role of the prior as a learned signal of move quality. Third, increasing the visit count *nb* must decrease the value of the expression. This is the discrete analogue of the standard exploration-exploitation balance: a move that has already been visited many times should be considered less attractive at the margin, so that exploration concentrates on under-visited alternatives. Without this constraint, the search would consider expressions that further reward already-visited moves, which would collapse exploration. The monotonicity constraints are checked numerically by sampling test values of *p* and *nb* and verifying the sign of the resulting change in the expression value.

These three constraints are not arbitrary: they are exactly the properties that the standard PUCT formula satisfies, and they are the minimal structural requirements for a candidate expression to be interpretable as an exploration term rather than as an arbitrary mathematical formula. The constraints prune the search space by a factor of approximately 45: among the $4{,}485{,}696$ syntactically legal expressions of size up to 8, only $99{,}571$ pass the constraints and are evaluated. Without this filtering, the cached-evaluation budget would be largely consumed by expressions that cannot serve as exploration terms in the first place. We did not perform a formal ablation of the constraints because removing the monotonicity properties produces, by design, expressions that are not exploration terms, and removing the $*/$ requirement does the same; the constraints are therefore best understood as a definitional choice of what counts as an exploration term in our search, not as tunable hyperparameters whose effect on performance can be cleanly measured.

We do not perform semantic deduplication of generated expressions: distinct syntactic expressions that reduce to the same mathematical function (e.g., $exp(log(nb)) + total$ and $log(exp(nb)) + total$, both equal to $nb + total$ on positive inputs) are evaluated as separate candidates. This is conservative but wasteful – semantic equivalence cannot exclude any expression of interest, but it does enlarge the search space. Symbolic simplification of candidates before evaluation is a natural optimization that we leave to future work.

4. EVALUATION OF EXPLORATION TERMS

The possible atoms for an expression contain the *h* hyperparameter. When evaluating an expression, the algorithm loops over possible values for h and keeps the best performing one.

For the exploration terms for PUCT, the algorithm compares the moves identified with the exploration term to those determined by PUCT with a greater number of evaluations.

In our experiments with PUCT, we dissociated the exploration term for the root from the exploration term for the states near the leaves. At the root, we aim to minimize the simple regret, while near the leaves, we are interested in minimizing another regret (Pepels et al., 2014b).

We now explain how the generated exploration terms can be used for PUCT and SHUSS.

4.1. The Exploration Terms for Go dataset

We use the term *descent* for one full iteration of the MCTS loop shown in Figure 1: traversing the tree from the root to a leaf using the exploration term at each step, expanding a new leaf, evaluating the leaf with the value network, and back-propagating the evaluation up the tree. A search of *N* tree descents therefore performs *N* such iterations.

The time required to precisely evaluate a generated exploration term in a real Go playing program can be substantial. For example, testing a Sequential Halving algorithm with a generated exploration term to play against a standard PUCT for 800 games with $1{,}024$ tree descents per move takes days. In order to obtain a fast evaluation of a given exploration term, we built a dataset of states associated with their cached searches so that the neural network no longer has to be queried during the evaluation of a candidate term.

The dataset is built as follows. We start from a collection of $1{,}700$ Go positions sampled from the test set of the Katago dataset. For each such position s, we run the policy-value network and retain only the 32 moves with the largest priors $P(s,a)$; the remaining moves are discarded because their cumulative prior mass is negligible, so they are essentially never selected by PUCT in practice, particularly at the small search budgets we consider. The choice of 32 is a practical trade-off that essentially loses no information for our purposes while keeping the cached-search dataset compact and fast to build. For each retained move a, we play a from s and run a sequence of $1{,}024$ independent PUCT descents starting from the resulting child state $s' = s.\text{play}(a)$. The $1{,}024$ leaf evaluations returned during these descents are stored in order. Concretely, each position s is stored as one file in which each of the 32 rows corresponds to one retained move a and contains the index of a, its prior $P(s,a)$, the cached-sequence length (which is always $1{,}024$ in our experiments), and the $1{,}024$ cached leaf evaluations themselves; the first row additionally records the side to play in s. This caching mechanism is what makes the exhaustive evaluation of nearly 10^5 candidate terms tractable: replaying a cached search consumes successive values from the cached sequences in microseconds per descent, rather than triggering a forward pass through the network.

The Exploration Terms for Go dataset is available at the following address:

https://www.lamsade.dauphine.fr/~cazenave/generate.zip

4.2. Using an exploration term for the selection of moves near the leaves of PUCT

Our first study designs an exploration term specifically for low-visit nodes – i.e., nodes whose visit count *total* is small. The motivation is provided by Table 1: in a PUCT search of 256 tree descents, 48.68% of all node visits during descents fall on nodes with *total* $\leqslant 4$. Such low-visit nodes are necessarily close to the frontier of the tree expansion, since a freshly created node starts with *total* $= 1$ and accumulates visits only as the search progresses. Optimizing the exploration term specifically for this regime, therefore, directly affects the largest fraction of the descents, while the high-*total* regime concerns nodes near the root that have already been visited many times.

We refer to this low-visit regime as "near the leaves" throughout the paper. This terminology is operational rather than topological: it does not refer to depth in the tree but rather to nodes with a small number of visits, regardless of their depth. In our experiments, "near the leaves" designates nodes with *total* $\leqslant 4$, 5, or 6 (see Table 2).

What we are designing is the *selection rule* used inside the descent step of MCTS (the left-most stage of Figure 1), specifically as applied at low-visit nodes. This is distinct from two other rules with which it could be confused. We are not estimating the value of a move (that is the role of the value network at the leaves). We are not selecting the exploration term close to the root (that is the role of the exploration term addressed separately in Section 4.3). We are designing the rule that decides which child of the current node is chosen during a single tree descent. The quality of such a rule is measured by how closely the resulting small-budget search converges, in terms of root-level visit distribution, to the output of standard PUCT at a much larger budget.

We now clarify the meaning of n_i. When evaluating a candidate near-leaves exploration term, we simulate, for each starting position s, a budget-N PUCT search rooted at s, in which the standard PUCT formula is replaced by the candidate expression at every node whose visit count *total* is below the chosen threshold (e.g. $total \leqslant 4$). The leaf evaluations returned during these simulated descents are read sequentially from the cached sequences (s,a) associated with the move a being descended. Let m_i be the move chosen at the root during descent i of this simulated search, and let n_i be the visit count at the root associated with m_i once the simulated search has completed; that is, $n_i = N(s,m_i)$ at the end of the N-descent search. The quantity $\text{policy}[m_i] = n_i/N$ is then the fraction of the simulated search budget spent on m_i at the root. We compare this distribution against the root visit distribution produced by standard PUCT with $N = 1{,}024$ on the same position, which acts as our reference. The score of the candidate expression aggregates this comparison across descent indices i, weighted by the empirical frequency p_i of nodes with i visits during a real PUCT search (Table 1), so that visits that occur more often during real searches contribute more to the score. Note that n_i is therefore a property of the simulated search at the root, not a property of a leaf or the descent path itself; the candidate expression is judged by how well its small-budget search at the root reproduces the choices of a much larger reference search.

Formally, this score is given by:

$$score = \sum_{i=1}^{n} p_i \times policy[m_i]$$

When using the exploration term in PUCT, the First Play Urgency (FPU) of an unvisited child is set to the average Q value of the node plus the evaluation of the exploration term for $nb = 1$ (Cazenave, 2022a).

4.3. Using an exploration term for the selection of moves at the root of PUCT

To evaluate the exploration terms at the root of PUCT, we take, at the end of the exploration, the move with the greatest number of associated tree descents and count it for the expression if it is the same move as the one found by PUCT with a large number of tree descents.

4.4. Using an exploration term for the selection of moves in SHUSS

In the same spirit as SHUSS, it is possible to use various exploration terms for choosing the moves to keep at the end of a round of Sequential Halving. Algorithm 4 provides the move selection process with an exploration term. The principle is to take the best half of the moves that maximize the expression. If the expression is the usual empirical average, then the algorithm is the usual Sequential Halving algorithm.

The algorithm evaluates the exploration term for the SHUSS algorithm with a given number of tree descents for states where the standard SHUSS has been run with a significantly greater number of tree descents. The accuracy of the exploration term is the percentage of times it finds the same move as standard SHUSS with a greater number of tree descents.

Algorithm 4 Selection of the moves to keep for the next round

Parameter: cutting ratio λ
Input: set of moves S_r
$S_{r+1} \leftarrow \emptyset$
for $i = 0$ **to** $\lceil \lambda \times |S_r| \rceil$ **do**
  $bestScore \leftarrow -\infty$
  for $j = 0$ **to** $|S_r|$ **do**
    if $S_r[j] \notin S_{r+1}$ **then**
      if $expression(S_r[j]) > bestScore$ **then**
        $bestMove \leftarrow S_r[j]$
        $bestScore \leftarrow expression(S_r[j])$
      end if
    end if
  end for
  $S_{r+1} \leftarrow S_{r+1} \cup \{bestMove\}$
end for
return S_{r+1}

5. EXPERIMENTAL RESULTS

In this section, we experiment with the discovery of Monte Carlo Tree Search exploration terms in the game of Go. We present the computer Go dataset that was used to train the neural network for the game of Go. The neural network is used to generate the Exploration Term dataset, which is, in turn, employed to evaluate the generated exploration terms.

We apply the framework to the discovery of near-leaves exploration terms for PUCT, root exploration terms for PUCT and SHUSS. We also test the discovered exploration terms in a Go program, making the new algorithms play against standard PUCT.

We run 100 processes, and each process plays 8 games, alternating between Black and White, with random seeds ranging from 0 to 99. The seed controls the stochastic sampling of the first 40 moves of each game from the policy distribution of the neural network: identical seeds produce identical opening sequences, and different seeds produce different opening sequences. The neural network itself is deterministic (fixed trained weights), and the generation and ranking of candidate exploration terms are deterministic as well; the seed, therefore, plays no role in either of these two stages and only affects the diversity of starting positions across the 800 games used to compute each winrate. The motivation for this stochastic opening is twofold. First, it diversifies the starting positions seen across the 800 games, ensuring that the comparison between PUCT and the proposed algorithm is averaged over a wide range of openings rather than being dominated by a single deterministic line of play; this reduces the variance of the estimated winrate and avoids overfitting the comparison to one particular opening sequence. Second, fully deterministic play between two networks based on the same policy would result in identical games on every seed, which would make the winrate either 0.5 (if both algorithms select the same moves) or some other degenerate value entirely determined by tie-breaking, neither of which carries useful statistical information. This stochastic-opening protocol is standard practice in the evaluation of AlphaZero-style agents, where it is used for the same reasons.

The winrate is calculated with 800 games against PUCT, playing Black and playing White on 400 starting states. The same network is used by both algorithms. The same number of tree descents is used for both algorithms.

In the SHUSS experiments, we use 400 games against PUCT.

We now specify the PUCT hyperparameters used as the baseline against which all discovered exploration terms are compared. The baseline implements the standard PUCT formula $Q(s,a)+c_p \cdot P(s,a) \cdot \sqrt{N(s)}/(1+N(s,a))$ with $c_p = 0.2$. This value was selected by tuning c_p for the baseline PUCT itself, on the same range of search budgets used in the experiments: we evaluated PUCT against itself across a sweep of c_p values and retained the value that gave the strongest play. The discovered exploration terms are then compared against this tuned baseline. The First Play Urgency at an unvisited child is set to the parent's average Q value, which is the standard convention. Tree-reuse across moves is disabled in both algorithms, so each move starts from a fresh tree of the same fixed size. Both algorithms — the baseline and any algorithm using a discovered exploration term — operate under identical conditions: they share the same trained neural network (Section 5.2), the same number of tree descents per move, the same starting positions, the same opening sequences (controlled by the seed described above), and the same value-network calls per leaf evaluation. The discovered terms are not given any additional information, training, or computational budget beyond what the baseline receives. The fact that c_p was tuned specifically for the baseline rather than for the discovered terms is conservative with respect to our claim that the discovered terms outperform PUCT: the comparison is between a hyperparameter-tuned PUCT and a discovered term that was selected on a separate criterion (accuracy against a long PUCT search) and has no comparable tuning of its own.

Throughout the experimental section, we measure the cost of a search by its *number of tree descents*, that is, the total number of paths from the root to a leaf performed by the algorithm before it returns a move. This is the parameter set by the user, reported in Tables 3 and 5. At a given tree node s, *total* denotes the number of tree descents that have passed through s so far plus 1, and at a given child move a of s, *nb* denotes the number of tree descents that have descended through a so far plus 1. We use "tree descents" exclusively for the algorithm-level quantity, and "*total*" and "*nb*" with their precise per-node and per-move meanings.

5.1. The computer Go dataset

The computer Go dataset is composed of games played by Katago (Wu, 2019) against itself in 2022. There are 1,000,000 different games in total in the training set. The input data consists of 31 19x19 planes (color to play, ladders, the current state on two planes, and two previous states on four planes). The output targets are the policy (a vector of size 361 with 1.0 for the move played and 0.0 for the other moves) and the value (close to 1.0 if White wins or close to 0.0 if Black wins). The test set is composed of 50,000 states taken randomly from 50,000 games that are not used in the training set.

5.2. The neural network

In order to conduct fast experiments, we test the exploration terms with PUCT using a lightweight neural network. The Student network is a Mobile network (Cazenave, 2022b): a residual architecture in which each block combines a depthwise-separable convolution with a squeeze-and-excitation unit, rectified linear activations, and batch normalization. The specific instantiation used in the experiments has 10 residual blocks of 120 channels each, for a total of fewer than 100,000 trainable parameters. The input is a tensor of shape $31 \times 19 \times 19$ (see Section 5.1). The network has two heads: a policy head that outputs a probability distribution over the moves of the 19×19 board, and a value head that outputs a scalar in $[0, 1]$ representing the predicted probability that White wins the game.

The network is trained on the $1{,}000{,}000$ self-play games from the Katago dataset described in Section 5.1. Rather than using one-hot move targets for the policy head, we use a teacher-student setup: a much larger network (also trained on the same Katago games) is used as a teacher, and its output policy distributions serve as the soft targets for the Student network. This significantly improves the policy accuracy of the small network compared to direct training on one-hot Katago moves. The value head is trained on the Katago evaluation for the state. Training is performed with the Adam optimizer, using a categorical cross-entropy loss for the policy head and a binary cross-entropy loss for the value head, with the two losses summed with equal weight. The learning rate follows a cosine annealing schedule. Training proceeds for 500 epochs, where each epoch consists of $100{,}000$ randomly sampled states from the training set.

The Student network regularly wins the tournaments among networks that are organized as an evaluation in our Deep Learning course. All models in these tournaments have fewer than $100{,}000$ parameters, including the Student network. The trained network used in our experiments is included in the dataset distribution at https://www.lamsade.dauphine.fr/~cazenave/generate.zip. The code to train Go networks is available at https://www.lamsade.dauphine.fr/~cazenave/importGolois2026.ipynb.

Table 1

Empirical distribution of node visits during a PUCT search with 256 tree descents. The value p_i is the average percentage of node visits that occur at nodes whose count *total* equals *i* at the time of the visit. Nodes with *total* $\leqslant 4$ account for 48.68% of node visits during a search.

i	1	2	3	4	5	6	7	8
p_i	26.43	10.62	6.55	5.08	3.98	3.25	2.69	2.34
i	9	10	11	12	13	14	15	16
p_i	2.07	1.86	1.70	1.60	1.41	1.33	1.25	1.16

Table 2

The best exploration terms near-leaves for nodes with 4, 5 and 6 tree descents or fewer. 99,571 expressions of size smaller than 8 that satisfy the constraints have been evaluated. The best evaluated expressions for nodes with 4 and 5 descents are the same.

n	Best Expression	Score
4	$p/\log(nb + e^{e^{total}})$	0.1255
5	$p/\log(nb + e^{e^{total}})$	0.1418
6	$p + \sqrt{total + p/nb}$	0.1540

Table 3

Match results for PUCT augmented with the discovered exploration term $p/\log(nb + \exp(\exp(total)))$ at nodes with *total* $\leqslant 4$, against standard PUCT. The leftmost column gives the number of tree descents allocated to both algorithms.

Tree descents	Winrate against PUCT
128	0.52875 ± 0.0176
256	0.55250 ± 0.0176
512	0.53125 ± 0.0176
1,024	0.54250 ± 0.0176

5.3. Using an exploration term for the selection of moves near the leaves of PUCT

Table 1 should be read as the empirical distribution of node visit counts encountered during a typical PUCT search of 256 tree descents: p_i is the average percentage of node visits that fall on a node currently holding i visits. The values are not "good" or "bad" in themselves; they describe the workload that the exploration term has to cope with. Two observations are immediate. First, the distribution is strongly right-skewed: the single-visit regime ($i = 1$) accounts for 26.43% of all node visits during a descent, the two-visit regime accounts for an additional 10.62%, and the percentages decay rapidly as i grows. Concretely, $p_6 = 3.25\%$ means that, on average, 3.25% of the node visits during a 256-tree descent PUCT search occur at nodes that currently have exactly 6 recorded visits. Second, the cumulative mass at small i is large: nodes with $i \leqslant 4$ visits account for 48.68% of node visits, and nodes with $i \leqslant 16$ visits account for more than 73%. This is the empirical justification for the near-leaves regime: the exploration term applied at low-visit nodes is the term that is exercised most often during a search, so an improvement in this regime translates into an improvement in a large fraction of the descent steps. Conversely, exploration terms specialized for high-visit nodes (large i) act on a small minority of node visits, and improvements in that regime have less impact on the search as a whole. Table 1 therefore plays a dual role: it motivates the very existence of a separate near-leaves study, and it provides the weights p_i used in the score function defined in Section 4.2 to aggregate per-regime accuracy into a single number.

Table 2 gives the best expressions found for nodes with 4 tree descents or fewer, 5 tree descents or fewer, and 6 tree descents or fewer.

Two questions arise about Table 2: how to interpret the scores, and how to interpret the expressions themselves. Concerning the scores, the absolute value of the score is bounded above by $\sum_{i=1}^{n} p_i$, the cumulative mass of the regime $total \leqslant n$ in Table 1, which equals 0.4868 for $n = 4$, 0.5266 for $n = 5$, and 0.5591 for $n = 6$. A score of 0.4868 for $n = 4$ would correspond to a candidate term whose move choices coincide perfectly with those of standard PUCT at $1{,}024$ tree descents on every descent that visits a node with $total \leqslant 4$. Comparable upper bounds hold for $n = 5$ and $n = 6$. The discovered terms in the table reach 0.1255, 0.1418, and 0.1540 for $n = 4$, 5, and 6 respectively; these are the highest scores among the $99{,}571$ candidate terms tested, and they exceed the score of standard PUCT at the same n, which is what motivates their use in the head-to-head experiments of Table 3. The score is therefore best read on a relative scale: not as a probability of agreement with PUCT, but as a number that admits ranking and is bounded above by a known constant.

Expressed as a fraction of the corresponding upper bound, the discovered terms achieve a proportion of $0.1255/0.4868 \approx 25.8\%$ for $n = 4$, $0.1418/0.5266 \approx 26.9\%$ for $n = 5$, and $0.1540/0.5591 \approx 27.5\%$ for $n = 6$. The discovered terms therefore not only improve in absolute score as n grows, but also recover a slightly larger fraction of the best possible score at the same n, confirming that the gain from $n = 4$ to $n = 6$ is real rather than an artifact of the larger upper bound.

Concerning the expressions, two patterns emerge. The expression $p/\log(nb + e^{e^{total}})$, which is best for both $n = 4$ and $n = 5$, has an instructive limiting behavior: $e^{e^{total}}$ dominates almost immediately as $total$ grows, so $\log(nb + e^{e^{total}}) \approx e^{total}$, and the term reduces to $p \times e^{-total}$. In other words, in the low-visit regime, the discovered term assigns each move a score that depends mainly on its prior, modulated by an exponentially decaying function of the parent's visit count and barely on the move's own visit count nb. This is a strong departure from PUCT, where the role of nb in the denominator is central; the result of the search suggests that, when nb is small, balancing siblings via nb matters less than biasing toward the prior. The expression $p + \sqrt{total + p/nb}$, best for $n = 6$, recovers a more PUCT-like structure in which nb appears explicitly: the additive p acts as a prior bias, and

$\sqrt{total + p/nb}$ plays the role of the visit-count-balancing exploration bonus. The crossover between the two structural families as n grows from 5 to 6 is itself informative: it suggests that the visit count nb becomes a useful signal only after a few visits have been accumulated, which is consistent with the intuition that one or two playouts give too little information to discriminate between siblings on visit count grounds alone.

Table 3 gives the results of the near-leaves exploration term $p/\log(nb + e^{e^{total}})$ used for nodes with 4 tree descents or fewer. It yields better results than PUCT.

Table 4

Evaluation of different exploration terms at the root. The accuracy of a term is the average accuracy at finding the move chosen by PUCT with $2,048$ tree descents, where the average is taken over searches with 128, 256, and 512 tree descents using the candidate term. The table shows the 10 terms with accuracy at least 0.79900. Term number 4 rediscovers the standard PUCT formula. Each term is also evaluated in match play with 128 tree descents and 800 games. For each of the 400 starting positions, the term plays against PUCT both as White and as Black. The "Winrate at root" column gives the result when the candidate term is used only at the root and PUCT is used elsewhere in the tree; the "Winrate in tree" column gives the result when the candidate term is used at every node of the tree.

Number	Expression	Accuracy	Winrate at root	Winrate in tree
1	$0.4 \times (\sqrt{\frac{p}{nb}} + total)$	0.80350	0.51125 ± 0.0177	0.47500 ± 0.0177
2	$e^{\frac{p \times \sqrt{\sqrt{total}}}{nb}}$	0.80050	0.52375 ± 0.0177	0.44625 ± 0.0176
3	$total - \frac{p}{log(0.8) \times nb}$	0.80050	0.49875 ± 0.0177	0.37250 ± 0.0170
4	$0.2 \times p \times \frac{\sqrt{total}}{nb}$	0.80000	0.50000 ± 0.0177	0.50000 ± 0.0177
5	$total + \sqrt{\frac{0.1 \times p}{nb}}$	0.80000	0.49375 ± 0.0177	0.48000 ± 0.0177
6	$p \times \frac{\sqrt{\sqrt{total}}}{nb}$	0.79900	0.53125 ± 0.0176	0.44375 ± 0.0175
7	$e \times (total + \frac{p}{nb})$	0.79900	0.50625 ± 0.0177	0.43125 ± 0.0175
8	$e^{0.7} \times (total + \frac{p}{nb})$	0.79900	0.49625 ± 0.0177	0.45250 ± 0.0176
9	$e^{\frac{p}{nb}} \times \sqrt{\sqrt{total}}$	0.79900	0.54500 ± 0.0176	0.43875 ± 0.0175
10	$0.3 \times (\sqrt{\frac{p}{nb}} + total)$	0.79900	0.46000 ± 0.0176	0.50750 ± 0.0177

5.4. Using an exploration term for the selection of moves at the root of PUCT

Table 4 presents various root exploration terms that were automatically found. The standard PUCT exploration term was rediscovered by our algorithm and appears as term number 4, which is reassuring evidence that the evaluation procedure is meaningful: the formula used by most MCTS practitioners is recovered by the empirical search, alongside other formulas of comparable accuracy. The other terms achieve accuracy scores within a narrow band (0.79900 to 0.80350), indicating that several structurally distinct expressions reproduce the choices of long PUCT searches almost equally well at this metric. The two columns "Winrate at root" and "Winrate in tree" report two different uses of the same candidate term: in the first, the candidate term is applied only at the root of the search and standard PUCT is used in the rest of the tree; in the second, the candidate term is used at every node of the tree. The two columns answer different questions: the "at root" column tests whether the term is a useful root-selection rule, while the "in tree" column tests whether it is a useful drop-in replacement for PUCT selection rule at every node of the search tree.

Table 5

Match results for the algorithm using a discovered simple-regret exploration term at the root and standard PUCT below the root, against standard PUCT throughout. The "Tree descents" column reports the number of tree descents allocated to both algorithms; the same network is used by both.

Root exploration term	Tree descents	Winrate against PUCT
$p \times \frac{\sqrt{\sqrt{total}}}{nb}$	128	0.53125 ± 0.0176
$p \times \frac{\sqrt{\sqrt{total}}}{nb}$	256	0.52000 ± 0.0177
$p \times \frac{\sqrt{\sqrt{total}}}{nb}$	512	0.53375 ± 0.0176
$p \times \frac{\sqrt{\sqrt{total}}}{nb}$	1,024	0.47500 ± 0.0177
$e^{\frac{p \times \sqrt{\sqrt{total}}}{nb}}$	128	0.52375 ± 0.0177
$e^{\frac{p \times \sqrt{\sqrt{total}}}{nb}}$	256	0.54375 ± 0.0176
$e^{\frac{p \times \sqrt{\sqrt{total}}}{nb}}$	512	0.50125 ± 0.0177
$e^{\frac{p \times \sqrt{\sqrt{total}}}{nb}}$	1,024	0.48625 ± 0.0177
$e^{\frac{p}{nb}} \times \sqrt{\sqrt{total}}$	128	0.54500 ± 0.0176
$e^{\frac{p}{nb}} \times \sqrt{\sqrt{total}}$	256	0.52875 ± 0.0176
$e^{\frac{p}{nb}} \times \sqrt{\sqrt{total}}$	512	0.53000 ± 0.0176
$e^{\frac{p}{nb}} \times \sqrt{\sqrt{total}}$	1,024	0.48375 ± 0.0177

A first observation from Table 4 is that high accuracy at predicting the long-PUCT move does not automatically translate into a high winrate, and that the two columns are loosely correlated at best: term number 3 and term number 10 have nearly identical accuracy (0.80050 and 0.79900) but very different winrates in the tree (0.37250 vs 0.50750). A term that closely tracks the reference search at the root may still suffer from instability deeper in the tree, which is the difference between the two winrate columns.

Table 5 gives the results of three of these root exploration terms matched against PUCT, this time used only at the root with standard PUCT in the rest of the tree, across a range of search budgets. For all three terms, the results are good up to 512 tree descents but fall behind standard PUCT at $1{,}024$ tree descents. We attribute this regime change to two compounding effects. First, the candidate terms in Table 4 were ranked by accuracy averaged over searches of 128, 256, and 512 tree descents, so they are, by construction, tuned to the small-budget regime; their behavior at $1{,}024$ tree descents lies outside the range over which they were evaluated. Second, at large budgets, PUCT itself approaches the asymptotic regime in which its theoretical guarantees are tight, so the headroom for an alternative formula to outperform it shrinks. The discovered terms are therefore best understood as small-budget specialists rather than as drop-in replacements at all budgets. This is a real limitation of the search at the root and a contrast with the near-leaves regime, where the discovered term keeps its advantage up to $1{,}024$ tree descents.

5.5. The SHUSS exploration term

The accuracy of an exploration term for SHUSS is defined as the percentage of moves found by the standard SHUSS algorithm with $1{,}024$ tree descents that are also found by the SHUSS algorithm using the exploration term and 64 tree descents.

Table 6

Evaluation of different exploration terms for 64 tree descents per move and the 6 best prior moves. The accuracy is the percentage of moves by standard SHUSS with $1{,}024$ tree descents, found by SHUSS with the exploration term and with 64 tree descents. The winrate is the percentage of games won by SHUSS with 64 tree descents and the exploration term against PUCT with 64 tree descents. The statistics are evaluated with 400 games.The s exploration term is standard SHUSS. The 400 starting states are taken from the Katago dataset test set games by playing 20 moves by Katago from the beginning of the games. The resulting states are balanced according to Katago. The best exploration term found is $s+0.04\times log(p)$. It was discovered using the $s + h \times log(p)$ exploration term with h = 0.05. Then the 0.04 value was tried for h and gave a winrate of 56.50 % against PUCT even though the accuracy was slightly worse (66.20 % against 66.50 % for h = 0.05).

Exploration term	Accuracy	Winrate against PUCT
s	64.60 %	0.4150 ± 0.0246
$s + 0.05 \times log(p)$	66.50 %	0.5575 ± 0.0248
$s + 0.04 \times log(p)$	66.20 %	0.5650 ± 0.0248

We evaluated the exploration terms with the best accuracies by using them in a computer Go program. The SHUSS algorithm, using an exploration term, plays 400 games against PUCT. Both algorithms use 64 tree descents to choose their moves. PUCT chooses the most simulated move, and SHUSS chooses the only remaining move in S_R. We use the atom s to represent the sum of the evaluations of a move, and the atom p to represent the prior of a move. The results for standard SHUSS with the usual exploration term s are given in Table 6. We can observe that standard SHUSS is weaker than PUCT. The exploration term $s + 0.04 \times log(p)$ is also tested against PUCT. SHUSS, with the discovered exploration term, is better than PUCT.

We now explain why incorporating $\log(p)$ improves SHUSS performance, particularly in the early rounds. Standard SHUSS uses the exploration term s, the sum of the evaluations of a move, to rank the candidate moves at the end of each round of Sequential Halving. In the first round, however, s is the sum of only a small number of evaluations per move: with a budget of 64 tree descents distributed over 6 moves and $\log_2(6) \approx 2.6$ rounds, each move receives 4 tree descents in the first round. With so few samples, s is a high-variance estimator of the true value of the move: the empirical sum is dominated by the noise of the value-network estimates at 4 leaves rather than by any reliable signal. As a consequence, the standard SHUSS algorithm risks eliminating moves whose true value is high but whose first-round empirical sum happens to be low.

The discovered term $s + 0.04 \times \log(p)$ adds a small bias derived from the policy prior. The function $\log(p)$ is large and negative for moves with low prior and approaches zero for moves with high prior, so adding $0.04 \times \log(p)$ subtly down-ranks moves that the policy considers weak. The coefficient 0.04 is small enough that once a move has accumulated several evaluations and s has grown large in absolute value, the prior contribution is dominated by the empirical sum, and the ranking reverts to the standard SHUSS rule. The crossover regime is precisely the one where s is unreliable: when $|s|$ is comparable to $0.04 \times |\log(p)|$, the prior provides additional information that the few tree descents cannot.

In effect, $s + 0.04 \times \log(p)$ acts as a regularized empirical-mean estimator: it biases the early-round ranking toward what the policy prior already knew, and it lets the empirical sum take over once enough tree descents have accumulated to make s a reliable signal on its own. This is the same principle that motivates the prior term in PUCT itself, and the discovery that SHUSS benefits from it confirms that the prior remains a useful anchor for move selection, even within an algorithm that, by design,

attempts to allocate tree descents uniformly. The net effect on the winrate from 0.4150 for standard SHUSS to 0.5650 for $s + 0.04 \times \log(p)$ is consistent with what the analysis predicts: most of the improvement comes from not eliminating strong-prior moves prematurely in the first round.

6. CONCLUSION

We presented a simple yet efficient method to find new exploration terms for Monte Carlo Tree Search. It uses the automatic generation of mathematical expressions and a fast evaluation of the generated expressions. The generated exploration terms are simple. The method discovered exploration terms that work better than the usual PUCT for three different algorithms: near-leaves exploration in PUCT, root exploration in PUCT for budgets of fewer than 512 tree descents, and SHUSS.

Our method for discovering new exploration terms is simple, fast, general, and empirically adapts the generated mathematical expressions to the problem at hand. Several directions are natural for future work. The most direct extension is to evaluate the proposed method on games other than Go. Within the family of perfect-information board games, Hex, Shogi, and Chess all admit policy-value networks of the same general form as the one used here, and the cached-search dataset construction transfers without modification. The framework is also a candidate for general game playing, where the absence of game-specific tuning makes an automatically discovered exploration term particularly valuable; in that setting, one could discover a single robust term that performs well across a portfolio of games. A second direction is to scale the search to larger expressions, which exhaustive enumeration cannot reach: beam search, evolutionary search, MCTS, or Large Language Model-guided search over expressions are all plausible substitutes once the size budget grows beyond what enumeration can cover. A third direction is to apply the same approach to components of MCTS beyond the exploration term, such as the recommendation policy at the root or the FPU rule. Finally, the cached-search trick used here to make exhaustive expression evaluation tractable is itself reusable beyond exploration-term discovery: it could be applied to the empirical tuning of MCTS hyperparameters more broadly, to the evaluation of novel MCTS algorithms, or to the comparison of move-ordering heuristics in any setting where the cost of an evaluation is dominated by neural-network inference.

From a more general point of view, Artificial Intelligence is becoming powerful enough to help discover new Artificial Intelligence algorithms. There are many further developments in the use of Artificial Intelligence to improve Artificial Intelligence.

REFERENCES

Alhejali, A.M. & Lucas, S.M. (2013). Using genetic programming to evolve heuristics for a Monte Carlo Tree Search Ms Pac-Man agent. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)* (pp. 1–8). IEEE.

Ameneyro, F.V. & Galván, E. (2022). Towards understanding the effects of evolving the MCTS UCT selection policy. In *2022 IEEE Symposium Series on Computational Intelligence (SSCI)* (pp. 1683–1690). IEEE.

Auer, P., Cesa-Bianchi, N. & Fischer, P. (2002). Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.*, *47*(2-3), 235–256.

Bravi, I., Khalifa, A., Holmgård, C. & Togelius, J. (2017). Evolving game-specific UCB alternatives for general video game playing. In *European Conference on the Applications of Evolutionary Computation* (pp. 393–406). Springer.

Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. & Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE TCIAIG*, *4*(1), 1–43.

Cazenave, T. (2007). Evolving Monte Carlo tree search algorithms. *Dept. Inf., Univ. Paris*, *8*.

Cazenave, T. (2010). Nested monte-carlo expression discovery. In *ECAI 2010* (pp. 1057–1058). IOS Press.

Cazenave, T. (2013). Monte-Carlo expression discovery. *International Journal on Artificial Intelligence Tools*, *22*(01), 1250035.

Cazenave, T. (2015). Sequential Halving Applied to Trees. *IEEE Trans. Comput. Intell. AI Games*, *7*(1), 102–105.

Cazenave, T. (2022a). Batch Monte Carlo Tree Search. In *International Conference on Computers and Games* (pp. 146–162). Springer.

Cazenave, T. (2022b). Mobile Networks for Computer Go. *IEEE Trans. Games*, *14*(1), 76–84. doi:10.1109/TG.2020.3041375.

Cazenave, T. (2024). Monte Carlo Search Algorithms Discovering Monte Carlo Tree Search Exploration Terms. In M. Hartisch, C. Hsueh and J. Schaeffer (Eds.), *Computers and Games*. Lecture Notes in Computer Science (Vol. 15550, pp. 103–115). Springer.

Cazenave, T. & Hamida, S.B. (2015). Forecasting financial volatility using nested Monte Carlo expression discovery. In *IEEE SSCI* (pp. 726–733).

Chen, X., Liang, C., Huang, D., Real, E., Wang, K., Pham, H., Dong, X., Luong, T., Hsieh, C.-J., Lu, Y., et al. (2024). Symbolic discovery of optimization algorithms. *Advances in Neural Information Processing Systems*, *36*.

Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H.J. van den Herik, P. Ciancarini and H.H.L.M. Donkers (Eds.), *CG 2006*. Lecture Notes in Computer Science (Vol. 4630, pp. 72–83). Springer.

Danihelka, I., Guez, A., Schrittwieser, J. & Silver, D. (2021). Policy improvement by planning with Gumbel. In *International Conference on Learning Representations*.

Dhankhar, H. & Cazenave, T. (2026). Nested Qubit Routing. In *Quantum Computing and Artificial Intelligence* (pp. 70–781). Springer.

Fabiano, N. & Cazenave, T. (2021). Sequential halving using scores. In *Advances in Computer Games* (pp. 41–52). Springer.

Fawzi, A., Balog, M., Huang, A., Hubert, T., Romera-Paredes, B., Barekatain, M., Novikov, A., R Ruiz, F.J., Schrittwieser, J., Swirszcz, G., et al. (2022). Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, *610*(7930), 47–53.

Finnsson, H. & Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. In *AAAI* (pp. 259–264).

Flanagan, D. (2026). Claude Code: One Year Later. In *Invited Talk at the AAAI 2026 Workshop: CodeMates 2026 Next-Gen Code Development with Collaborative AI Agents*. Statement that 95% of Claude Code is written by Claude Code.

Holmgård, C., Green, M.C., Liapis, A. & Togelius, J. (2018). Automated playtesting with procedural personas through MCTS with evolved heuristics. *IEEE Transactions on Games*, *11*(4), 352–362.

Illetskova, M., Elnabarawy, I., Da Silva, L.E.B., Tauritz, D.R. & Wunsch, D.C. (2019). Nested monte carlo search expression discovery for the automated design of fuzzy ART category choice functions. In *GECCO* (pp. 171–172).

Karnin, Z.S., Koren, T. & Somekh, O. (2013). Almost Optimal Exploration in Multi-Armed Bandits. In *ICML* (pp. 1238–1246).

Kocsis, L. & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *17th European Conference on Machine Learning (ECML'06)*. LNCS (Vol. 4212, pp. 282–293). Springer.

Li, Y., Li, W., Yu, L., Wu, M., Liu, J., Li, W., Hao, M., Wei, S. & Deng, Y. (2024). Discovering Mathematical Formulas from Data via GPT-guided Monte Carlo Tree Search. *arXiv preprint arXiv:2401.14424*.

Lipardi, V., Dibenedetto, D., Stamoulis, G. & Winands, M.H.M. (2025). Quantum Circuit Design using a Progressive Widening Enhanced Monte Carlo Tree Search. *Advanced Quantum Technologies*, *8*(10), e2500093.

Lu, Q., Tao, F., Zhou, S. & Wang, Z. (2021). Incorporating Actor-Critic in Monte Carlo tree search for symbolic regression. *Neural Computing and Applications*, *33*, 8495–8511.

Maes, F., St-Pierre, D.L. & Ernst, D. (2013). Monte carlo search algorithm discovery for single-player games. *IEEE Transactions on Computational Intelligence and AI in Games*, *5*(3), 201–213.

Mankowitz, D.J., Michi, A., Zhernov, A., Gelmi, M., Selvi, M., Paduraru, C., Leurent, E., Iqbal, S., Lespiau, J.-B., Ahern, A., et al. (2023). Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, *618*(7964), 257–263.

Méhat, J. & Cazenave, T. (2008). Monte-Carlo Tree Search for general game playing. *Univ. Paris*, *8*.

Moudřík, J., Křen, T. & Neruda, R. (2017). Algorithm Discovery with Monte-Carlo Search: Controlling the Size. In *ICTAI* (pp. 390–395). IEEE.

Pepels, T., Cazenave, T. & Winands, M.H.M. (2015). Sequential Halving for Partially Observable Games. In T. Cazenave, M.H.M. Winands, S. Edelkamp, S. Schiffel, M. Thielscher and J. Togelius (Eds.), *CGW 2015*. CCIS (Vol. 614, pp. 16–29). Springer.

Pepels, T., Cazenave, T., Winands, M.H.M. & Lanctot, M. (2014a). Minimizing Simple and Cumulative Regret in Monte-Carlo Tree Search. In *CGW 2014*. CCIS (Vol. 504, pp. 1–15). Springer.

Pepels, T., Tak, M.J.W., Lanctot, M. & Winands, M.H.M. (2014b). Quality-based Rewards for Monte-Carlo Tree Search Simulations. In *ECAI* (pp. 705–710).

Pitrat, J. (2008). A Step toward an Artificial Artificial Intelligence Scientist. LIP6 Research Report.

Rosenhahn, B. & Osborne, T.J. (2023). Monte Carlo graph search for quantum circuit optimization. *Physical Review A*, *108*(6), 062615.

Ruiz, F.J., Laakkonen, T., Bausch, J., Balog, M., Barekatain, M., Heras, F.J., Novikov, A., Fitzpatrick, N., Romera-Paredes, B., van de Wetering, J., et al. (2024). Quantum Circuit Optimization with AlphaTensor. *arXiv preprint arXiv:2402.14396*.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., et al. (2020). Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, *588*(7839), 604–609.

Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*, 484–489.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *nature*, *550*(7676), 354–359.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*(6419), 1140–1144.

Sun, F., Liu, Y., Wang, J.-X. & Sun, H. (2022). Symbolic physics learner: Discovering governing equations via monte carlo tree search. *arXiv preprint arXiv:2205.13134*.

Świechowski, M., Godlewski, K., Sawicki, B. & Mańdziuk, J. (2023). Monte Carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, *56*(3), 2497–2562.

Wang, P., Usman, M., Parampalli, U., Hollenberg, L.C. & Myers, C.R. (2023). Automated quantum circuit design with nested monte carlo tree search. *IEEE Transactions on Quantum Engineering*.

Wu, D.J. (2019). Accelerating self-play learning in go. *arXiv preprint arXiv:1902.10565*.