Nested Depth Search

Junkang Li*[1,2], Tristan Cazenave*[3†], Swann Legras*[1], Arthur Queffelec*[4], Véronique Ventos[1]

[1]NukkAI, Paris, France
[2]Université Caen Normandie, ENSICAEN, CNRS, Normandie Univ, GREYC UMR6072, F-14000 Caen, France
[3]LAMSADE, Université Paris Dauphine - PSL, Paris, France
[4]WorldWise, Paris, France
junkang.li@nukk.ai,tristan.cazenave@lamsade.dauphine.fr,slegras@nukk.ai,arthur.queffelec@worldwise.fr,vventos@nukk.ai

Abstract

Nested Monte Carlo Search (NMCS) has numerous applications, ranging from chemical retrosynthesis to quantum circuit design. We propose a generalization of NMCS that we named Nested Depth Search (NDS), in which a fixed depth search is used during a higher-level playout to generate the states sent to lower-level exploration. We establish the runtime of NDS and provide algorithms to compute the exact probability distribution of sequences generated by NDS. Experiments with the Set Cover problem and the Multiple Sequence Alignment problem show that NDS outperforms NMCS with the same time budget.

Code — https://github.com/arqueffe/NDS

Introduction

Nested Monte Carlo Search (NMCS) (Cazenave 2009) is a variant of Monte Carlo Tree Search (MCTS) that uses recursive levels of playouts to better explore the search space. In contrast to MCTS, NMCS optimizes the playouts at the different levels until the terminal states, making it better suited to sequential optimization problems (Baier and Winands 2012). Recent applications of NMCS include chemical retrosynthesis planning (Segler, Preuss, and Waller 2018; Genheden et al. 2020), where NMCS significantly improves the AiZynthFinder MCTS (Roucairol and Cazenave 2024). Novel NMCS-type algorithms have the potential to enhance solutions to important practical problems, such as quantum circuit design (Wang et al. 2023; Dhankhar and Cazenave 2025), molecular design for generating novel drug-like molecules (Roucairol et al. 2024b), cryptanalysis (Dwivedi, Morawiecki, and Srivastava 2019), automated guided vehicle scheduling (von Bothmer, Mihalák, and Winands 2024), coalition structure generation (Roucairol et al. 2024a), radar pulse repetition frequency selection (Ardon, Briheche, and Cazenave 2024), etc.

In this paper, we present the Nested Depth Search (NDS), a generalization of the NMCS algorithm. In NDS, a higher-level search generates all the states at a fixed depth and sends them to the lower-level search for exploration. Compared to NMCS, which is equivalent to NDS with a fixed depth of 1, NDS explores more and, therefore, discovers better outcomes. Our main contribution includes a theoretical analysis of NDS (hence, by extension, of NMCS).

This paper is organized as follows. The first section provides a survey on Nested Search (NS) along with its various applications and algorithmic improvements. The second section presents the NDS algorithm and analyzes its runtime. We then study the behavior of NS on a quintessential optimization problem called the Left Move Problem; the exact probability distribution of the outcomes of NDS is computed by algorithms. Finally, we present experimental results on the application of NDS to other problems, such as Set Cover and Multiple Sequence Alignment.

Related Work

Nested search originated from Monte Carlo search and was first used to improve a Backgammon program, thanks to nested rollouts (Tesauro and Galperin 1996). Subsequently, it was used in combinatorial optimization (Bertsekas, Tsitsiklis, and Wu 1997) and in stochastic scheduling problems (Bertsekas and Castanon 1999). Nested rollouts combined with a heuristic to choose the next move at the base level were used to improve a Klondike solitaire program (Yan et al. 2004). Nested rollouts were used with heuristics that change with the stage of the Thoughtful Solitaire game, a version of Klondike Solitaire in which the locations of all cards are known (Bjarnason, Tadepalli, and Fern 2007). They were also used to learn a control policy for planning (Fern, Yoon, and Givan 2004).

Nested Monte Carlo Search (NMCS) (Cazenave 2009) is a related algorithm that works well for puzzles and optimization problems. NMCS adopts a randomized playout policy at level zero and biases its higher-level playouts using lower-level playouts. The main improvement of NMCS over NS is the memorization of the best sequence at each recursion level; without this memorization, nested algorithms do not perform well at levels higher than one. NMCS applications include single-player general game playing (Méhat and Cazenave 2010), cooperative pathfinding (Bouzy 2013), software testing (Poulding and Feldt 2014), heuristic model checking (Poulding and Feldt 2015), the Pancake problem

*These authors contributed equally.
†Corresponding author.

Copyright © 2026, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

(Bouzy 2016), two-player games (Cazenave et al. 2016), and the RNA inverse folding problem (Portela 2018).

Online learning of playout policies combined with NMCS has yielded good results in optimization problems (Rimmel, Teytaud, and Cazenave 2011). This idea has been further developed into the Nested Rollout Policy Adaptation (NRPA) algorithm for puzzles and optimization (Rosin 2011), which leads to new world records in Morpion Solitaire and crossword puzzles. The principle of NRPA is to adapt the playout policy by reinforcing the moves of the best sequence of moves found so far at each level. NRPA has been applied to multiple problems: the TSPTW problem (Cazenave and Teytaud 2012; Edelkamp et al. 2013); 3D packing with object orientation (Edelkamp, Gath, and Rohde 2014); the physical traveling salesman problem (Edelkamp and Greulich 2014); the multiple sequence alignment problem (Edelkamp and Tang 2015); various problems of logistics, such as vehicle routing, container packing, and robot motion planning (Edelkamp et al. 2016; Cazenave et al. 2021); graph coloring (Cazenave, Negrevergne, and Sikora 2020); inverse folding (Cazenave and Fournier 2020); refutation of spectral graph theory conjectures (Roucairol and Cazenave 2022); etc.

Nested Depth Search

In this section, we describe a generalization of NMCS that we call Nested Depth Search (NDS). Throughout this paper, NDS will be solely studied in the context of single-player games (i.e. optimization problems); applying NDS to two-player games is a straightforward extension.

NDS, shown in Alg. 2, is a recursive meta-algorithm parameterized by three meta-parameters: a base policy, a depth $d \geq 1$, and a stride $s \geq 1$. The role of each parameter will be explained below. Note that at each level of recursion, NDS returns a terminal state reachable from the current state (i.e. the state given as an argument to Alg. 2).

At recursion level 0, NDS is reduced to its base policy, the generic form of which is given in Alg. 1. Given an internal state, this base policy determines how to continue the playout until reaching a terminal state, which will be returned by the algorithm. The move selection in Line 3 can be, for example, uniformly random. It can also be based on an informed search heuristic, a neural network evaluation function, or even another sophisticated search algorithm.

The core idea of NDS is to bias higher-level playouts using lower-level playouts, thereby improving upon the algorithm serving as base policy. More concretely: For a given internal state and some level $l > 0$, NDS develops all children at depth d (the depth meta-parameter) of the current state (Line 6), then evaluates each of these children by an NDS at level $l-1$ (Line 8), and descends s (the stride meta-parameter) steps towards the best terminal state found by the recursive calls at level $l-1$ (Line 10).

To summarize, at each level of recursion, NDS exploits the exploration and terminal states found by lower levels to determine which move to make in the current state. From the description above, it can be observed that NDS is an extension of NMCS: NMCS is NDS with $d = s = 1$.

Due to the memorization of the best terminal state found so far (stored in the variable `best`), NDS is naturally an

Algorithm 1: Base Policy for the playout at level 0

```
1 def BasePolicy(state):
2 |   while state is not terminal:
3 |   |   move ← choose a move
4 |   |   state ← Play (state, move)
5 |   return state
```

Algorithm 2: Nested Depth Search parameterized by a base policy and two positive integers d and s

```
1  def NDS(level l, state):
2  |   if l = 0:
3  |   |   return BasePolicy(state)
4  |   best ← None /* best terminal state
   |    found so far */
5  |   while state is not terminal :
6  |   |   nodes ← all children of state at depth d
7  |   |   for node ∈ nodes:
8  |   |   |   terminal ← NDS(l − 1, node)
9  |   |   |   Update best with terminal
10 |   |   state ← descend s steps from state
   |   |    towards best
11 |   return state
```

anytime algorithm. It should be noted that the best terminal state found by a higher-level is *not* passed down to lower-level NDS; this is by design in order to force lower-level recursive calls to explore more and to search independently of each other and from higher-levels.

Remark. *We assume that we have a method of comparing two terminal states, which is used in Line 9 to update the best terminal state found so far. This can be carried out by simply comparing the score of these terminal states or by using some evaluation function, which allows for the possibility that these terminal states do not necessarily correspond to true terminal states in the search tree, thereby adding more flexibility to NDS.*

Remark. *To implement the descent of s steps in Line 10, the sequence of moves leading to the best terminal state found by lower-level NDS can either be memorized as part of the information of this terminal state or recomputed.*

For the update in Line 9, there is a choice to be made about how to treat terminal states with the same score. The default behavior we assume is to do nothing when a new terminal state with the same score is found. However, we may also opt to replace the old terminal state with the new one with the same score; we refer to the NDS with this update behavior as NDS_r, where r stands for "replacement". Intuitively, the principal variation leading to the newly found terminal state is less explored than the one leading to the previous terminal state;[1] it is more likely that exploration around this new principal variation would yield better results. Probabilistic analysis and experimental results in later sections

[1]This can be the case when these two terminal states are found from states at different tree heights.

do confirm that NDS_r performs slightly better than NDS.

In the following, we use the notation D2S1-NDS-L2 to denote the NDS with meta-parameters $d = 2$ and $s = 1$, and running at recursion level $l = 2$. An NDS instantiated with other parameter values will be denoted similarly.

Illustrating Examples of NDS

We now illustrate the behavior of NDS on some binary trees. Below, the left and right actions are denoted by 0 and 1, respectively. Each node is then labeled by the sequence of actions taken to reach it from the root. For example, the node reached by following left and then right from the root is denoted by 01.

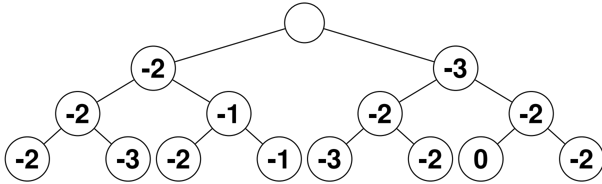

Figure 1: A tree with nodes evaluated by some base policy.

In Fig. 1, we have a binary tree with non-root nodes labelled by their values given by some evaluation function. We assume that the base policy of the NDS discussed below is based on this particular evaluation function.

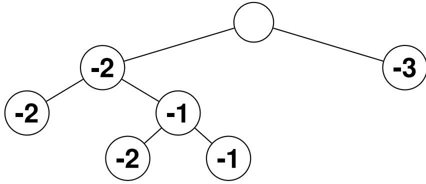

Figure 2: The nodes explored by D1S1-NDS-L1.

We consider first the behavior of D1S1-NDS-L1, which is equivalent to a level-1 NMCS. The nodes explored by this NDS are shown in Fig. 2. At the root, NDS-L1 develops the nodes 0 and 1 and calls the base policy to evaluate them, which returns respectively -2 and -3 as score. NDS-L1 therefore descends to node 0 and then develops nodes 00 and 01, for which the base policy returns respectively -2 and -1. In the end, NDS arrives at leaf 011 with score -1.

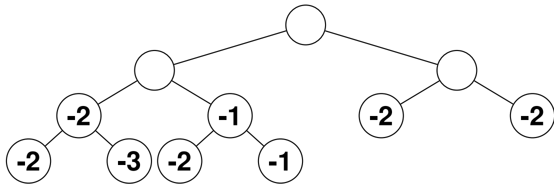

Figure 3: The nodes explored by D2S1-NDS-L1.

We now turn to D2S1-NDS-L1; the nodes explored by it are shown in Fig. 3. Since $d = 2$, NDS develops all 4 nodes 2 steps below the root (00, 01, 10, 11). The best one (evaluated by the base policy) being node 01, NDS-L1 descends one step to node 0 and develops all 4 grand-children of node 0 (000, 001, 010, 011). In the end, NDS also arrives at leaf 011 as in the case $d = 1$ above.

Notice that NDS explores all nodes up to depth $d \cdot l$; a larger value of d leads to a more exhaustive search from the current state. For the tree in Fig. 1, D3S1-NDS develops all the 8 leaves and eventually arrives at leaf 110 with a score of 0. This illustrates how greater d overcomes inaccurate evaluation from lower levels (e.g. the score of node 1 is underestimated by the base policy to be -3, stopping NDS with $d = 1$ from exploring the subtree rooted at node 1).

Runtime Analysis

To analyze the runtime of NDS, we assume that it takes the base policy $\mathcal{O}(h)$ time to evaluate a tree of height h, which is typically the case (e.g. for random playout policy).

For a tree with a branching factor b and height h, the runtime of NMCS at level l (i.e. D1S1-NDS-Ll),

$$\mathcal{O}\left(b^l h^{l+1}\right), \tag{1}$$

is proven by Méhat and Cazenave (2010).[2]

With depth d and stride s, it is as if NDS traversed a tree with a branching factor b^d and height h/s (except at level 0, where the playout is assumed to be performed with a stride of one). Hence, from Eq. (1), we deduce the runtime of NDS to be

$$\mathcal{O}\left((b^d)^l \cdot h \cdot (\frac{h}{s})^l\right). \tag{2}$$

Alternatively, we can verify Eq. (2) using the recurrence relation between the runtime at two different levels of recursion. Let us write $T^l_{d,s}(b,h)$ for the runtime of DdSs-NDS-Ll for a tree with a branching factor b and height h. Then, from the description of Alg. 2, we deduce that

$$T^l_{d,s}(b,h) = b^d \cdot \sum_{i=0}^{h/s} T^{l-1}_{d,s}(b, i \cdot s), \tag{3}$$

with the boundary condition $T^0_{d,s}(b,h) = C \cdot h$ from the base policy, where C is a constant independent of all parameters. It can be easily verified that Eq. (2) is indeed the solution of Eq. (3) with this boundary condition.

NDS vs NMCS

Comparing Eq. (2) and Eq. (1), NDS is faster than NMCS for the same tree if $b^d < bs$. Rewritten as $b^d \cdot d/s < bd$, this can be easily interpreted: the left-hand side is the number of recursive calls made by NDS in order to descend d steps, whereas the right-hand side is the number of calls by NMCS.

Hence, for sufficiently small branching factors (i.e. b close to 1),[3] it may be beneficial to use NDS over NMCS, as it allows for better exploration (by using a large d) while maintaining a similar runtime. As we shall see later, even with

[2]Technically, the runtime is $\mathcal{O}\left(b^l h^{l+1}/(l+1)!\right)$. The factorial is omitted since we only consider NDS with small l (e.g. $l \leq 4$).

[3]For example, in constraint satisfaction problems with binary variables, one unit propagation can be viewed as one step in the search tree, which means $b < 2$.

the same runtime (e.g. when $b = d = s = 2$), NDS can still empirically outperform NMCS.

Probabilistic Analysis of NDS

Clearly, larger d and l yield better performance but worse runtime, which can be compensated for by a larger s. To gain quantitative insight into this compromise among the meta-parameters d, s, and l, we analyze the behavior of NDS instantiated on the Left Move Problem defined below.

Definition 1 (Left Move Problem)**.** The Left Move Problem (LMP) of height $h > 0$, denoted by LMP(h), is an abstract optimization problem that involves selecting between two possible actions (left/right) for a sequence of h actions. The score of a terminal state is determined by the number of left actions taken.

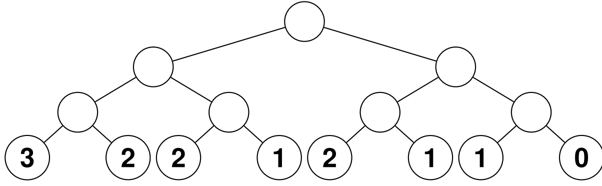

Figure 4: A binary tree representation of LMP(3) with leaves labelled by their score.

LMP(h) can be represented by a binary tree of height h, as shown in Fig. 4. LMP has been shown to have a similar score distribution to various combinatorial optimization problems (Cazenave 2009); this motivates our analysis of the behavior of NDS applied to LMP.

We assume in the following: (1) the base policy of NDS is the uniformly random policy, which chooses between left and right with equal probability at every internal node; (2) the children at depth d (Line 6 in Alg.2) are ordered uniformly at random, which means the 2^d recursive calls are completed in a uniformly random order.[4]

Since the goal of LMP is to take left actions as often as possible, it is of interest to analyze how higher-level NDS biases actions more towards the left. Such an analysis has been partially carried out for NMCS (i.e. D1S1-NDS) (Cazenave 2009). Even at level 1, there is no known closed formula for the probability distributions.[5] To bypass this obstacle, we will use algorithms to compute the exact probability distribution over terminal states of LMP(h) reached by NDS and NDS_r (i.e. NDS with replacement).

Probabilistic Analysis Applied to LMP

Let $d, s, l \in \mathbb{N}_{>0}$ and $h \geq d$. Every possible sequence of h actions in LMP(h) (i.e. every leaf of the binary tree of LMP(h)) can be written as a binary string of length h, in which 0 means going left and 1 means going right. For example, the string $00\ldots0$ represents the sequence of h left actions, i.e. the leftmost leaf of the tree. We write S_h for the set of all binary strings of length h, and $\Delta(S_h)$ for the set of all probability distributions over S_h.

NDS applied to the root of LMP(h) induces a probability distribution over S_h, which is given by the probability of NDS returning each terminal state (element of S_h). Notice that NDS has one persistent memory: the best terminal state it has found so far (the variable `best` in Alg. 2, initialized to be void on Line 4). When NDS begins to descend, the memory of the best state found by the lower-level recursive calls influences its move selection (Line 10).

This observation motivates us to write $\mathbb{P}^{d,s}_{\text{NDS}}[l][h, \mathbb{P}^h_{\text{best}}] \in \Delta(S_h)$ for the probability distribution over S_h induced by DdSs-NDS-Ll applied to the root of LMP(h) when `best` is initialized with $\mathbb{P}^h_{\text{best}} \in \Delta(S_h) \cup \{\textbf{None}\}$.[6] In this notation, our goal is to compute $\mathbb{P}^{d,s}_{\text{NDS}}[l][h, \textbf{None}] \in \Delta(S_h)$.

Notice that `best` can be viewed as a random variable over S_h; we need to analyze how its distribution changes during the execution of Alg. 2. At the beginning of each loop (Line 5), `best` has a certain value: void, if this is the first loop; or, the value of `best` at the end of the last loop. In the middle of the loop, the terminal states returned by all lower-level recursive calls are compared to produce a potentially better terminal state, which we name `rec_best`.[7] The value of `best` can then be replaced by `rec_best` (Line 9) if the latter has a strictly better score (in NDS) or at least the same score (in NDS_r). Therefore, to compute the new distribution of `best` after each while loop, we need two procedures:

- The first one to compute $\mathbb{P}^{d,s}_{\text{rec}}[l][h]$, the distribution of `rec_best` after the recursive calls made during the first while loop by DdSs-NDS-Ll at the root of LMP(h);[8]
- The second one to compute, given the distribution $\mathbb{P}^h_{\text{best}}$ of `best` and the distribution of `rec_best`, the new distribution of `best`.

Computation of $\mathbb{P}^{d,s}_{\text{rec}}[l][h]$ Since DdSs-NDS-Ll applied to the root of LMP(h) develops all 2^d children at depth d of the root, the recursive calls are made to NDS at level $l-1$ with the root of a subtree of height $h-d$ (the same tree as LMP($h-d$)).

Let us label each child at depth d by $0 \leq i \leq 2^d - 1$, where i can be interpreted as a binary string of length d. For every $c \in S_{h-d}$, let us write $[i, c]$ for the concatenation of the strings i (of length d) and c (of length $h-d$). Then $[i, c] \in S_h$ is a leaf of the tree of LMP(h). Notice that the recursive call at level $l-1$ starting at child i can only reach leaves of the form $[i, c]$ with $c \in S_{h-d}$. To know with what probability each leaf is reached, we can use $\mathbb{P}^{d,s}_{\text{NDS}}[l-1][h-d, \textbf{None}] \in$

[4] This could be the case when, for example, recursive calls are carried out by parallel computation. In any case, ordering the calls randomly could help decrease bias in the default move ordering.

[5] Indeed, the probability of NMCS going left at the root is the probability of a simple symmetric 1D random walk starting at 1 finishing below 0 after $h-1$ steps, which has no closed formula.

[6] If $\mathbb{P}^h_{\text{best}} = \textbf{None}$, `best` is initialized to be void; otherwise, its value is chosen from S_h according to the distribution $\mathbb{P}^h_{\text{best}}$.

[7] Alg. 2 may be rewritten to make an explicit mention of the variable `rec_best`; we choose the current form of the algorithm to highlight its anytime nature.

[8] Notice that $\mathbb{P}^{d,s}_{\text{rec}}[l][h]$ is also the distribution of `best` after the first while loop, since this variable is initialized to be **None**.

Algorithm 3: Computation of $\mathbb{P}^{d,s}_{\text{rec}}[l][h] \in \Delta(S_h)$

```
1  def P^{d,s}_rec(level l, height h):
2      P^{d,s}_rec(c) ← 0 for all c ∈ S_h
3      Compute P^{d,s}_NDS[l − 1][h − d, None] ∈ Δ(S_{h−d})
4      for (c_0, ..., c_{2^d −1}) ∈ (S_{h−d})^{2^d}:
5          P ← ∏_{0≤i≤2^d −1} P^{d,s}_NDS[l − 1][h − d, None](c_i)
6          max_score ← max_{0≤i≤2^d −1} Score([i, c_i])
7          count ← number of [i, c_i] with score
            max_score
8          for every [i, c_i] with score max_score:
9              P^{d,s}_rec([i, c_i]) += P/count
10     return P^{d,s}_rec
```

Algorithm 4: Update of $\mathbb{P}^h_{\text{best}} \in \Delta(S_h) \cup \{\textbf{None}\}$ with any distribution $\mathbb{P}$ of rec_best

```
1  def Update(P^h_best, P ∈ Δ(S_h)):
2      if P^h_best = None:
3          return P
4      P^h_new(c) ← 0 for all c ∈ S_h
5      for (c, c') ∈ S_h × S_h with Score(c) > Score(c'):
6          P^h_new(c) += P^h_best(c) · P(c') + P^h_best(c') · P(c)
7      for (c, c') ∈ S_h × S_h with Score(c) = Score(c'):
8          if with replacement:
9              P^h_new(c) += P^h_best(c') · P(c)
10         else:
11             P^h_new(c) += P^h_best(c) · P(c')
12     return P^h_new
```

Algorithm 5: Computation of $\mathbb{P}^{d,s}_{\text{NDS}}[l][h, \mathbb{P}^h_{\text{best}}]$

```
1  def P^{d,s}_NDS(l, h, P^h_best ∈ Δ(S_h) ∪ {None}):
2      if l = 0:
3          return the uniform distribution over S_h
4      if h ≤ d:
5          return the uniform distribution over all
            terminal states with the maximum score
6      P^h_best ← Update(P^h_best, P^{d,s}_rec[l][h])
7      for 0 ≤ i ≤ 2^s − 1:
8          define P^{h−s}_{best,i}(·) as P^h_best([i, ·])
9          for c ∈ S_{h−s}:
10             P^{d,s}_NDS([i, c]) ← P^{d,s}_NDS[l][h − s, P^{h−s}_{best,i}](c)
11     return P^{d,s}_NDS
```

$\Delta(S_{h-d})$: $[i, c]$ is reached with probability $\mathbb{P}^{d,s}_{\text{NDS}}[l-1][h-d, \textbf{None}](c)$ by the recursive call at level $l-1$ starting at child i.

The computation of $\mathbb{P}^{d,s}_{\text{rec}}[l][h]$ can then be performed as in Alg. 3. We first initialize the list of probabilities to be all zeros and compute $\mathbb{P}^{d,s}_{\text{NDS}}[l-1][h-d, \textbf{None}]$ (by Alg. 5 detailed later). Then, for each combination of 2^d terminal states (each written as $[i, c_i]$) found by the 2^d lower-level recursive calls, we compute the probability of this combination (Line 5), which is to be shared by all terminal states with the maximum score (Line 9).[9]

Update of $\mathbb{P}^h_{\text{best}}$ In Alg. 4, we show how to update the distribution of best with the distribution of rec_best. If $\mathbb{P}^h_{\text{best}} = \textbf{None}$, the distribution of best is taken to be the one of rec_best. Otherwise, the new value of best can be $c \in S_h$ if its current value is c with a strictly better score than the value of rec_best, or its current value has a strictly worse score than the value of rec_best, which is c; this explains the two terms on Line 6. When the values of best and rec_best have equal scores, it depends on the replacement policy: the value of best is replaced by the value of rec_best in NDS_{r} (Line 9); best keeps its current value in NDS (Line 11).

Computation of $\mathbb{P}^{d,s}_{\text{NDS}}[l][h, \mathbb{P}^h_{\text{best}}]$ With the help of Alg. 3 and 4, this probability can be computed by recursion on both l and h, as in Alg. 5. When $l = 0$, the uniformly random base policy is invoked, hence all terminal states are reached with equal probability. When $h \leq d$, the tree will be exhaustively explored by NDS, hence only terminal states with the maximum score are reached with equal probability.

Otherwise, NDS launches recursive lower-level calls and updates the variable best (the new distribution of which is computed on Line 6) and descends s steps towards best. Let $0 \leq i \leq 2^s - 1$ be a potential destination of this descent. All possible values of best leading NDS to i can be written as $[i, c]$ with $c \in S_{h-s}$. The probability of NDS eventually reaching some terminal state $[i, c^*]$ is given by the probability that best is of the form $[i, c]$ for some c, multiplied by the probability that NDS reaches c^* when starting at the subtree rooted at i with its memory initialized to the current value of best. Taking into account the linearity with respect to the probabilities, we can simplify the recurrence relation into the one on Line 10.

Validation

Notice that Alg. 3 and Alg. 5 are mutually recursive; their well-definedness is guaranteed by the fact that each recursion strictly decreases either h (Line 10 in Alg. 5) or l (Line 3 in Alg. 3).

We present in Table 1 the exact probability distributions of the scores of NDS and NDS_{r} on LMP(8) with varying parameters. The negligible Mean Squared Error between the results computed by Alg. 3–5 and those obtained from simulations testifies to the correctness of our algorithms and our implementation of them.

From Table 1, we observe that, unsurprisingly, larger l or d increases score performance, while larger s diminishes

[9]The probability is to be shared since we made the assumption that the recursive calls are completed in a uniformly random order; each terminal state with the maximum score has an equal chance of being picked as the rec_best.

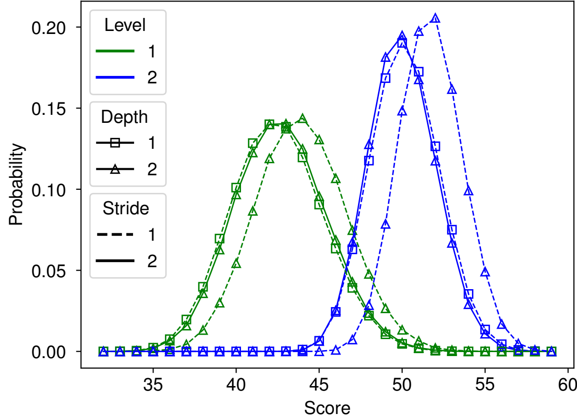

Figure 5: Distributions of the scores of NDS_r on LMP(60) with varying parameters. As indicated in the legend, each parameter variation is represented by a different plot style. For example, D2S2-NDS_r-L1 is depicted with a full green line and triangular markers.

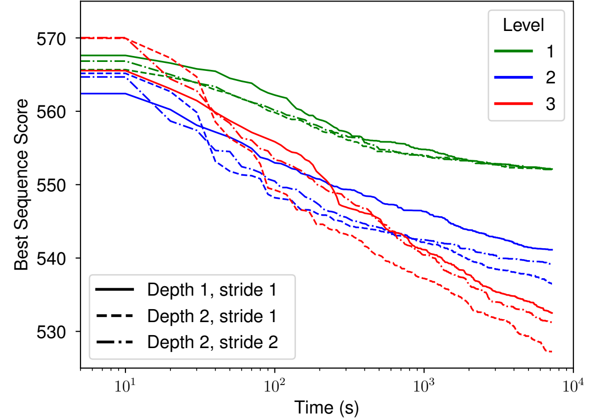

Figure 6: Evolution of the score (lower is better) of the best sequence of NDS_r on instance scp43 (best score of 515) of the OR-library using heuristic Chvatal's Greedy Procedure as ε-greedy playout algorithm with $\varepsilon = 0.05$. We reported the score averaged over 100 runs of 2 hours. As indicated in the legend, each parameter variation is represented by a different plot style. For example, D2S1-NDS_r-L2 is depicted with a blue and regularly dashed line.

it. Also, D2S2-NDS consistently outperforms NMCS (i.e. D1S1-NDS) at the same recursion level. Although less pronounced, NDS_r consistently surpasses NDS, especially at low recursion levels.

Beyond LMP

Although this section focuses on analyzing NDS on LMP under two assumptions (uniformly random base policy; random order of recursive calls), Alg. 3-5 have been carefully refactored to be generic and applicable to probabilistic analysis under other settings. For example, we can consider other base policies by adjusting Line 3 in Alg. 5 accordingly; other orders of recursive calls by modifying the probability sharing on Line 9 in Alg. 3; other problems beyond LMP by simply changing the function Score and replacing every occurrence of 2 with the new branching factor.

With Alg. 3-5, we can also study the variant of NDS without memorization, i.e. without using the variable `best`: it suffices to initialize $\mathbb{P}^h_{\text{best}}$ to **None** before the first line in Alg. 5. However, no memorization would lead to worse performance (cf. the section on related work).

Experimental Results

In this section, we present experimental results to compare NDS with different recursion levels, depths, and strides. The problems used for the experiments are LMP, Set Cover, and Multiple Sequence Alignment.

LMP

As the exact computation of the probability distribution of NDS scores has a high time complexity, we examine the behavior of NDS in large instances of LMP with 100,000 simulations. Fig. 5 shows the distributions of the scores of NDS_r on LMP(60). Increasing depth improves the score while maintaining a lower runtime compared to increasing the recursion level. In contrast, increasing the stride negatively impacts the search score while reducing the runtime. Note that on larger instances of LMP, NMCS_r (i.e. D1S1-NDS_r) and D2S2-NDS_r seem to have extremely close score performance. Also, while not presented here for the lack of space, we report that NDS_r outperforms NDS at recursion level 1, while the converse occurs at level 2.

Set Cover

The Set Cover Problem (SCP) is an optimization problem that computes the smallest number of subsets from a collection such that their union equals the union of the entire collection. The decision problem associated with SCP is known to be NP-complete (Karp 1972). A variant of SCP, the Weighted Set Cover Problem, seeks to identify a minimal cost covering, with each set assigned a cost. The OR-library offers datasets of problem instances for benchmarking various combinatorial optimization problems, including the Weighted Set Cover Problem (Beasley 1990).

We use a variation of DBNMCS described by Ardon, Bricheche, and Cazenave (2024), where rollouts have a probability $\varepsilon = 0.05$ of rejecting the action proposed by the heuristic. Fig. 6 presents the results of 100 runs, each lasting 2 hours, of repeated NDS_r calls for recursion levels from 1 to 3, with varying depths and strides on the scp43 instance.

We observe that for each recursion level, D2S2-NDS_r outperforms NMCS_r (i.e. D1S1-NDS_r), but only after a certain time point; this effect becomes more noticeable as the level increases. This can be explained by the fact that, despite having the same total runtime for a branching factor of 2, D2S2-NDS_r completes its first iteration later than NMCS_r.

Score	(1,1,1)	(1,2,1)	(1,2,2)	(2,1,1)	(2,2,1)	(2,2,2)	(3,1,1)	(3,2,1)	(3,2,2)
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	3.50e−14	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	1.15e−7	3.57e−15	4.75e−10	1.38e−41	0.0	0.0	0.0	0.0	0.0
3	0.000 195	0.0	1.00e−5	0.0	0.0	0.0	0.000 570	0.0	0.0
4	0.0106	0.000 518	0.003 50	4.26e−9	3.70e−10	6.09e−15	0.0	0.0	0.0
5	0.102	0.0263	0.0646	0.000 194	3.70e−10	2.99e−6	0.0	0.0	0.0
6	0.322	0.198	0.281	0.0327	0.001 06	0.0107	0.000 466	2.23e−10	1.05e−6
7	0.396	0.451	0.428	0.363	0.172	0.327	0.156	0.0239	0.100
8	0.169	0.325	0.223	0.604	0.827	0.662	0.844	0.976	0.900
Average	6.61	7.07	6.80	7.57	7.83	7.65	7.84	7.98	7.90
MSE	1.49e−5	1.60e−5	6.23e−6	3.83e−7	9.73e−7	2.58e−5	8.34e−6	2.21e−6	9.91e−8
Score	**(1,1,1)**	**(1,2,1)**	**(1,2,2)**	**(2,1,1)**	**(2,2,1)**	**(2,2,2)**	**(3,1,1)**	**(3,2,1)**	**(3,2,2)**
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	5.46e−16	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	1.21e−8	5.38e−16	8.76e−11	2.28e−49	0.0	0.0	0.0	0.0	0.0
3	6.59e−5	5.89e−8	7.09e−6	1.28e−22	0.0	0.0	5.37e−91	0.0	0.0
4	0.006 17	0.000 314	0.002 23	1.09e−10	5.46e−29	9.89e−16	1.17e−34	0.0	0.0
5	0.0769	0.0202	0.0502	4.85e−5	8.28e−11	1.40e−6	6.02e−13	0.0	0.0
6	0.289	0.176	0.255	0.0220	0.000 581	0.008 20	0.000 141	8.50e−11	1.05e−6
7	0.418	0.449	0.441	0.334	0.153	0.312	0.126	0.0227	0.100
8	0.210	0.355	0.252	0.644	0.846	0.679	0.874	0.977	0.900
Average	6.75	7.14	6.89	7.62	7.85	7.67	7.87	7.98	7.90
MSE	1.24e−6	0.000 175	0.000 196	0.000 232	8.75e−5	4.85e−5	6.72e−5	2.39e−7	1.42e−8

Table 1: Exact probability distributions of scores of NDS (above) and NDS_r (below) with parameters (l, d, s) on on LMP(8). We also provide the Mean Square Error (MSE) between the exact computation and the simulation of 100,000 executions.

n	(l, d, s)	Sum
3	(1,1,1)	47 374
3	(1,2,2)	47 182
3	(2,1,1)	45 441
3	(2,2,2)	45 114
4	(1,1,1)	96 550
4	(1,2,2)	96 062
4	(2,1,1)	92 900
4	(2,2,2)	92 267

Table 2: Sums of the edit distances of n RNA sequences randomly chosen from the MIR database. Sum over 1,000 different subsets of n sequences (smaller is better).

Although it may take even longer, D2S1-NDS_r eventually outperforms the other variants by a clear margin (improving the solution by around 1–2%).

Multiple Sequence Alignment

We also compared NMCS and NDS for Multiple Sequence Alignment (Carrillo and Lipman 1988) with the edit distance as the objective function. The algorithms are tested with greedy playouts that never insert a gap in the microRNA sequence database (Griffiths-Jones 2006); the results are shown in Table 2. Again, D2S2-NDS always outperforms NMCS (i.e. D1S1-NDS) at the same recursion level.

Conclusion

In this work, we introduced Nested Depth Search (NDS), a generalization of NMCS with the meta-parameters depth d and stride s. The depth parameter d broadens the search at higher-level playouts, while the stride parameter s accelerates traversal down the search tree.

The runtime of NDS has been established and shows that, in particular, NDS with $d = 2$ and $s = 2$ has the same runtime as NMCS for binary trees. We also provided algorithms to compute the exact probability distribution of NDS's outcomes on the Left Move Problem (LMP), which models combinatorial optimization problems with binary choices. Probabilistic analysis shows that on LMP, NDS with $d = 2$ and $s = 2$ clearly outperforms NMCS at the same level of recursion; NDS_r, a variant that replaces its best sequence on equality, outperforms NDS with the same parameters.

In the last part, experimental results from LMP, Set Cover, and Multiple Sequence Alignment provided better insight into the performance of NDS with different values of depth, stride, and recursion level. Again, NDS_r with $d = 2$ and $s = 2$ outperformed NMCS_r, and, predictably, NDS_r with $d = 2$ and $s = 1$ consistently yielded better results.

We believe that the new meta-parameters of NDS help improve upon NMCS; thus, NDS has the potential to find better solutions to important practical problems. Our runtime and probabilistic analysis can provide guidance on selecting appropriate values for these parameters. The generic construction of the probability analysis algorithms also makes it possible to apply them, with only minor modifications, to problems or settings not discussed in this work.

While not explored in this work, NDS also has the potential to enhance parallelization by leveraging the depth meta-parameter to optimize core usage beyond the branching factor limitation of NMCS. This would be part of our future work, along with the runtime analysis for the probabilistic analysis algorithms and other variants of NDS.

References

Ardon, M.; Briheche, Y.; and Cazenave, T. 2024. Binarized Monte Carlo Search for Selection Problems. In *International Conference on Learning and Intelligent Optimization*, 13–31. Springer.

Baier, H.; and Winands, M. H. 2012. Nested Monte-Carlo Tree Search for Online Planning in Large MDPs. In *ECAI*, volume 242, 109–114.

Beasley, J. E. 1990. OR-Library: Distributing Test Problems by Electronic Mail. *Journal of the Operational Research Society*, 41(11): 1069–1072.

Bertsekas, D. P.; and Castanon, D. A. 1999. Rollout algorithms for stochastic scheduling problems. *Journal of Heuristics*, 5(1): 89–108.

Bertsekas, D. P.; Tsitsiklis, J. N.; and Wu, C. 1997. Rollout algorithms for combinatorial optimization. *Journal of Heuristics*, 3(3): 245–262.

Bjarnason, R.; Tadepalli, P.; and Fern, A. 2007. Searching solitaire in real time. *ICGA Journal*, 30(3): 131–142.

Bouzy, B. 2013. Monte-Carlo Fork Search for Cooperative Path-Finding. In *Computer Games Workshop at IJCAI*, 1–15.

Bouzy, B. 2016. Burnt Pancake Problem: New Lower Bounds on the Diameter and New Experimental Optimality Ratios. In *SOCS*, 119–120.

Carrillo, H.; and Lipman, D. 1988. The multiple sequence alignment problem in biology. *SIAM journal on applied mathematics*, 48(5): 1073–1082.

Cazenave, T. 2009. Nested Monte-Carlo Search. In Boutilier, C., ed., *IJCAI*, 456–461.

Cazenave, T.; and Fournier, T. 2020. Monte Carlo Inverse Folding. In *Monte Search at IJCAI*.

Cazenave, T.; Lucas, J.; Triboulet, T.; and Kim, H. 2021. Policy Adaptation for Vehicle Routing. *AI Communications*.

Cazenave, T.; Negrevergne, B.; and Sikora, F. 2020. Monte Carlo Graph Coloring. In *Monte Search at IJCAI*.

Cazenave, T.; Saffidine, A.; Schofield, M. J.; and Thielscher, M. 2016. Nested Monte Carlo Search for Two-Player Games. In *AAAI*, 687–693.

Cazenave, T.; and Teytaud, F. 2012. Application of the Nested Rollout Policy Adaptation Algorithm to the Traveling Salesman Problem with Time Windows. In *Learning and Intelligent Optimization - 6th International Conference, LION 6*, 42–54.

Dhankhar, H.; and Cazenave, T. 2025. Nested Qubit Routing. In *Quantum Computing and Artificial Intelligence Workshop at AAAI*.

Dwivedi, A. D.; Morawiecki, P.; and Srivastava, G. 2019. Differential cryptanalysis of round-reduced speck suitable for internet of things devices. *IEEE Access*, 7: 16476–16486.

Edelkamp, S.; Gath, M.; Cazenave, T.; and Teytaud, F. 2013. Algorithm and knowledge engineering for the TSPTW problem. In *Computational Intelligence in Scheduling (SCIS), 2013 IEEE Symposium on*, 44–51. IEEE.

Edelkamp, S.; Gath, M.; Greulich, C.; Humann, M.; Herzog, O.; and Lawo, M. 2016. Monte-Carlo Tree Search for Logistics. In *Commercial Transport*, 427–440. Springer International Publishing.

Edelkamp, S.; Gath, M.; and Rohde, M. 2014. Monte-Carlo Tree Search for 3D Packing with Object Orientation. In *KI 2014: Advances in Artificial Intelligence*, 285–296. Springer International Publishing.

Edelkamp, S.; and Greulich, C. 2014. Solving physical traveling salesman problems with policy adaptation. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 1–8. IEEE.

Edelkamp, S.; and Tang, Z. 2015. Monte-Carlo Tree Search for the Multiple Sequence Alignment Problem. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015*, 9–17. AAAI Press.

Fern, A.; Yoon, S. W.; and Givan, R. 2004. Learning Domain-Specific Control Knowledge from Random Walks. In *ICAPS*, 191–199.

Genheden, S.; Thakkar, A.; Chadimová, V.; Reymond, J.-L.; Engkvist, O.; and Bjerrum, E. 2020. AiZynthFinder: a fast, robust and flexible open-source software for retrosynthetic planning. *Journal of Cheminformatics*, 12(1): 70.

Griffiths-Jones, S. 2006. miRBase: the microRNA sequence database. *MicroRNA Protocols*, 129–138.

Karp, R. M. 1972. Reducibility Among Combinatorial Problems. In Miller, R. E.; and Thatcher, J. W., eds., *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*, The IBM Research Symposia Series, 85–103. Plenum Press, New York.

Méhat, J.; and Cazenave, T. 2010. Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4): 271–277.

Portela, F. 2018. An unexpectedly effective Monte Carlo technique for the RNA inverse folding problem. *BioRxiv*, 345587.

Poulding, S. M.; and Feldt, R. 2014. Generating structured test data with specific properties using nested Monte-Carlo search. In *GECCO*, 1279–1286.

Poulding, S. M.; and Feldt, R. 2015. Heuristic Model Checking using a Monte-Carlo Tree Search Algorithm. In *GECCO*, 1359–1366.

Rimmel, A.; Teytaud, F.; and Cazenave, T. 2011. Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows. In *EvoApplications*, volume 6625 of *LNCS*, 501–510. Springer.

Rosin, C. D. 2011. Nested Rollout Policy Adaptation for Monte Carlo Tree Search. In *IJCAI*, 649–654.

Roucairol, M.; Arjonilla, J.; Saffidine, A.; and Cazenave, T. 2024a. Lazy Nested Monte Carlo Search for Coalition Structure Generation. In *ICAART (2)*, 58–67.

Roucairol, M.; and Cazenave, T. 2022. Refutation of Spectral Graph Theory Conjectures with Monte Carlo Search. In *COCOON 2022*.

Roucairol, M.; and Cazenave, T. 2024. Comparing search algorithms on the retrosynthesis problem. *Molecular Informatics*, e202300259.

Roucairol, M.; Georgiou, A.; Cazenave, T.; Prischi, F.; and Pardo, O. E. 2024b. DrugSynthMC: An Atom-Based Generation of Drug-like Molecules with Monte Carlo Search. *Journal of Chemical Information and Modeling*, 64(18): 7097–7107.

Segler, M. H.; Preuss, M.; and Waller, M. P. 2018. Planning chemical syntheses with deep neural networks and symbolic AI. *Nature*, 555(7698): 604–610.

Tesauro, G.; and Galperin, G. 1996. On-line policy improvement using Monte-Carlo search. *Advances in Neural Information Processing Systems*, 9.

von Bothmer, E.; Mihalák, M.; and Winands, M. H. 2024. Nested Monte-Carlo Search for Scheduling an Automated Guided Vehicle in a Blocking Job-Shop. In *BNAIC*.

Wang, P.; Usman, M.; Parampalli, U.; Hollenberg, L. C.; and Myers, C. R. 2023. Automated quantum circuit design with nested monte carlo tree search. *IEEE Transactions on Quantum Engineering*, 4: 1–20.

Yan, X.; Diaconis, P.; Rusmevichientong, P.; and Roy, B. 2004. Solitaire: Man versus machine. *Advances in Neural Information Processing Systems*, 17.