# High-Diversity Monte-Carlo Tree Search

**Tristan Cazenave and Stefan Edelkamp**


## Abstract

For combinatorial search in single-player games *nested Monte-Carlo search* is an apparent alternative to algorithms like UCT that are applied in two-player and general games. To trade exploration with exploitation the randomized search procedure intensifies the search with increasing recursion depth. If a concise mapping from states to actions is available, the integration of policy learning has leads to *nested rollout with policy adaptation* (NRPA).

In this paper we propose refinements to NRPA that improve solution diversity. As in *Beam-NRPA*, *High-Diversity NRPA* keeps a bounded number of solutions in each recursion level. It includes several improvements that further reduce the running time of the algorithm and improve its diversity. We illustrate effectiveness in a number of applications.


## Introduction

With the success of applying reinforcement learning to play expert-level Backgammon (Tesauro 1995), the concept of sampling the outcome of a game in random playouts has been around. Later on, bandit-based Monte-Carlo planning and UCT (Kocsis and Szepesvári 2006) extended the use of playouts and changed the way in which computer play many two-player and general games. The history of playing games with an increasing level of performance is long, with the current climax of Google Deep Mind's *AlphaGo* defeating a professional Go player in a match[1].

Cazenave 2009 has invented *nested Monte-Carlo search* (NMCS), a randomized search algorithm inspired by UCT (Kocsis and Szepesvári 2006) but specifically designed to solve single-player games. Instead of relying on a single rollout at each search tree leaf, the decision-making in level $l$ of the algorithm relies on a level $(l-1)$ search for its successors. Besides playing games (Browne et al. 2004), the NMCS algorithm solves mathematical problems (Bouzy 2006).

With *nested rollout policy adaptation* (NRPA), Rosin came up with the idea to learn a policy within the recursive procedure. As a randomized search procedure NRPA has been very successful in solving a variety of optimiztation problems, including puzzles like *morpion solitaire* (Rosin 2011), but also hard optimization tasks in logistics like *constraint traveling salesman problems* (Edelkamp et al. 2013) combined *pickup-and-delivery tasks* (Edelkamp and Gath 2014b), *vehicle routing* (Gath, Herzog, and Edelkamp 2013), and *container packing* (Edelkamp and Gath 2014a) problems.

Monte-Carlo tree search algorithms balance entering unseen areas of the search space (exploration) with working on already establisehd good solutions (exploitation). Many Monte-Carlo search algorithms including NRPA, however, suffer from a solution process that has many inferior solutions in the beginning of the search. If policies are learnt too quickly, the number of different solutions reduces, and if not strong enough they will not help sufficiently well to enter parts of the search space with good solutions.

In other words, the diversity of the search remains limited. Beam NMCS is a combination of memorizing a set of best playouts instead of only one best playout at each level. This set is called beam and all the positions in the set are developed. The algorithm has been parallelized and applied to solve instances to *morpion solitaire* (Cazenave 2012).

Beam search carries over to improve NRPA, enforcing an increased diversity in a set of solutions. In Beam-NRPA (Cazenave and Teytaud 2012), for each level of the search, instead of a singleton the algorithm keeps a bounded number of solutions together with their policies in the recursion tree. In selected applications like TSPTW problems, Beam-NRPA improves over NRPA.

In this paper, we reengineer the implementation of Beam-NRPA. We show that the solution quality of Beam-NRPA (and NRPA) in some domain can be improved by applying a selection of refinements. In particular, we study more closely how to increase the diversity in Beam-NRPA. The proposed search algorithm is called *High-Diversity-NRPA*, HD-NRPA for short.

The paper is structured as follows. First, we revisit NRPA and Beam-NRPA. Then, we turn to the proposed implementation refinements that increase the performance of these search algorithms. Next, we look at options to increase the diversity in the search, both in the generic search algorithm and its policy adaptation subroutine. We show the impact of the new approach in three challenging problem domains: the *same game*, the *vehicle routing problem* (both with mini-



[1] https://deepmind.com/alpha-go.html

```
procedure NRPA(level l, policy p)
begin
    if l = 0 then
        Best ← Playout(p)
    else
        p'_l ← p
        Best ← (Init,⟨⟩)
        for i = 1, ..., N
            r ← NRPA(l − 1, p'_l)
            if r better than Best then
                Best ← r
                Adapt(Best, p, p'_l)
    return Best
end
```

Figure 1: Nested rollout with policy adaptation. To cover both minimization and maximization problems, score ordering is imposed by means of implementing *Init*, *better*, and *best*. $L$ is implemented as a sorted list.

```
procedure Beam-NRPA(level l, policy p)
begin
    if l = 0 then
        (score, rollout) ← Playout(p)
        return (score, rollout, p)
    Beam_l ← (Init,⟨⟩,p)
    for i = 1, ..., N
        L ← ∅
        for t ← (score, rollout, p) ∈ Beam_l
            L ← L ∪ {t}
            T ← Beam-NRPA(l − 1, p)
            for t' ← (score', rollout', p') ∈ T then
                Adapt(rollout', p', p)
                L ← L ∪ {t'}
        Beam_l ← B best in Beam_l ∪ L
    return Beam
end
```

Figure 2: Beam nested rollout with policy adaptation.

mization objective function), and the *snake- and coil-in-the-box* problems (with maximization objective function).

## NRPA

NRPA is a randomized optimization scheme that belongs to the wider class of Monte-Carlo tree search (MCTS) algorithms (Browne et al. 2004). The main concept of MCTS is the random *playout* or *rollout* of a position, whose outcome, in turn, changes the likelihood of generation successors for subsequent trials. Other prominent members in this class of reinforcement learning strategies are *upper confidence bounds applied to trees* (UCT) (Kocsis and Szepesvári 2006), and *nested monte-carlo search* (NMCS) (Cazenave 2009). MCTS is the state-of-the-art in playing two-player games such as Go or Hex (Huang et al. 2013) or puzzles like the Pancake problem (Bouzy 2015), and has been applied also to other problems than games like mixed-integer programming, contraint problems, mathematical expression, function approximation, physics simulation, cooperative pathfinding, as well as planning and scheduling (Bouzy 2013).

What makes NRPA (Rosin 2011) different to UCT and NMCS is the concept of learning a policy through an explicit mapping of encoded moves to selection probabilities. The algorithm helped finding a new world record in *morpion solitaire*, and high-quality solutions in *crossword puzzles*. As we will see, NRPA is a general search procedure and applies to many games as well as practical domains.

The pseudo-code of the recursive search procedure is shown in Fig. 1. NRPA has two main parameters that trade exploitation with exploration: the number of levels $l$ and the branching factor $N$ of successors in the recursion tree.

## Beam-NRPA

Beam-NRPA is an extension of NRPA that maintains $B$ instead of one best solution in each level of the recursion. The motivation behind Beam-NRPA is to warrant search progress by an increased diversity of existing solutions to prevent the algorithm from getting stuck in local optima.

The basic implementation of the Beam-NRPA algorithm has been proposed by (Cazenave 2012) and is shown in Fig. 2. Each solution is stored together with its score and the policy that was used to generate it. Better solutions are inserted into a list, which is kept sorted wrt. to the objective to be optimized.

As the NRPA recursion otherwise remains the same, the number of playouts to a search with level $L$ and (iteration) width $N$ rises from $N^L$ to $(N \cdot B)^L$. To control the size of the beam, we allow different beam widths $B_l$ in each level $l$ of the tree[2]. At the end of the procedure, $B_l$ best solutions together with their scores and policies are returned to the next higher recursion level. For each level $l$ of the search, one may also allow the user to specify a varying iteration width $N_l$. This yields a complexity of the algorithm Beam-NRPA of performing $\prod_{l=1}^{L} N_l B_l$ rollouts.

## Refinements

We propose several refinements to Beam-NRPA.

### Dropping Policy Information

First, we have observed that copying the policy in each rollout of Beam-NRPA is a rather expensive operation that can dominate the runtime of the entire algorithm.

In fact, further code analysis showed that the policy update is always performed wrt. the currently best solution found in a level and the policy one level up, so that it is not required to store the policy attached each solution, as long as we keep $B_l$ best policies alive for each level $l$ of the recursive search procedure.

### Employing Faster Adaptation

For a faster processing of policy adaptation, we avoid the regeneration of successors by providing all the informa-

[2]Common values for $B_l$ in a level 4 search are $(1, 10, 10, 10)$

**procedure** NRPA-Adapt-Improve(level $l$, policy $p$, policy $p'$)
**begin**
  **for** $c_i \in Code_l$
    $p'[c_i] \leftarrow p'[c_i] + \alpha$
  $z \leftarrow 0$
  **for** $c' \in Succ_{l,i}$
    $z \leftarrow z + exp(p[c'])$
  **for** $c' \in Succ_{l,i}$
    $p'[c'] \leftarrow p'[c'] - \alpha \cdot exp(p[c'])/z$
**end**

Figure 3: An implementation of policy adaptation for NRPA that refers to recorded data for successor information, $\alpha$ is the learning rate, usually $\alpha = 1$.

tion that is needed at the time we construct the solution in the rollout. Hence, we store the sequence of codes $Code_l$ and successor node codes $Succ_l$ for each best solution (relative to a level $l$) produced, where the *code* is a user-specified domain-specific address into the policy table calculated based on the current state and the current move executed in this state (Rosin 2011).

The implementation in Fig. 3 shows that this strategy is already applicable to the original NRPA algorithm. It leads to minor extensions to the implementation of the generic *playout* function: each time a successor is checked for availability the corresponding code is stored.

We see that the update in *Adapt* affects only the codes of the good solution to be adapted and its successor codes, to balance the postive effect put on choosing it as negative effect to all of its successors.

### Avoiding Memory Defragmentation

To avoid fragmented access to the memory and operating system calls to provide memory, high-speed algorithm implementations often avoid dynamic memory allocation or have their own memory maintenance and allocators.

Beam-NRPA pre-allocates the information in the beam in static arrays and operates on the stored information directly. Besides faster insertion and deletion this allows to follow the progress of the search by showing the top $k \leq B_l$ elements.

## Improving the Diversity

Beam-NRPA itself is inspired by the objective of higher diversity in the solution space of NRPA. In larger search spaces NRPA often got stuck with inferior solutions. It simply takes too long to backtrack to less determined policies in order to visit other parts in the search space.

The beam is stored in a bounded number of *buckets*. The information contained in the buckets of a beam is visualized in Fig. 7. Instead of the moves executed in a rollout we store the *Code* of the chosen move and the code of its successors *Succ*. Additionally, the length of the rollout and its score is stored for each bucket in the beam.

### Improving Diversity in the NRPA Driver

When looking at a beam, a natural question is to warrant that the solutions kept in the beam are substantially different. This can be imposed by a matching the best obtained rollout with of the ones stored in the beam. Duplicate solutions wrt. this criterion are excluded from the beam. Fig. 4 provides a pseudo-code implementation.

The application of a filter to improve diversity is implemented in method *Similar*. We expect that $s_i = s_j$ implies $Similar(s_i, s_j)$ and $Similar(s_i, s_j) = Similar(s_j, s_i)$. The output is a truth value (interpreted as a number in $\{0, 1\}$). The beam is scanned for similar states, and if present, the new insertion request is rejected. Such similarity can be implemented on top of the score of the solution, the solution length, or other features of the rollout. The example implementation in Fig. 5 looks at the score and the length of the rollout.

The concept of similarity implies a formal characterization of *solution diversity*.

**Definition 1 (Diversity)** *Let $\mathcal{S}$ be a set of solutions of an optimization problem with and let $Similar$ be a pairwise similarity score between every two solution $s_i$ and $s_j$ with $1 \leq i, j \leq |\mathcal{S}|$, then the diversity is defined as the sum of the pairwise similarities, i.e.,*

$$div(\mathcal{S}) = \sum_{s_i, s_j \in \mathcal{S}} Similar(s_i, s_j).$$

This means that if the solutions are pairwise similar the diversity is low. A similar concept is that of pre-sortedness in an input array by adding the pairwise number of inversions.

One important aspect is that adaptation is now applied in every iteration, while before it was applied only for improved solutions. This increases the number of calls significantly but allows more information to be passed between the members in the beam. If the parameters are chosen carefully, the efforts for the playouts and for executing policy adaption are roughly the same.

We also skip some $\Theta_l$ iterations before we start learning. The motivating objective is the *secretary problem*, in which the best secretary out of $n$ rankable applicants should be hired for a position. Applicants are interviewed one after the other and the final decision has to be made immediately after the interview. The optimal stopping rule rejects the first $n/e$ applicants after the interview and then stops at the first applicant, who is better than every applicant interviewed so far[3].

Diversity is an objective that has to be dealt with care. In some domains the solution length (like the *snake-in-the-box*) already is the score, so that only solutions of different lengths are kept in the beam. This may limit the number of good solutions in the beam (too) drastically. As a solution to this problem, we propose to include other state features into the fractional part of the solution.

A good compromise has to be found. Using the entire state vector for similarity detection requires comparing regenerated solutions, which can be slow, or storing the full state in

[3] we experimented with a value of 10%.

```
procedure HD-NRPA(level l, policy p)
begin
    for b = 1, ..., B_l
        score_{l,b} ← Init
    if l = 0 then
        Score_{0,1} ← Playout(p)
        return Score_{0,1}
    for i = 1, ..., N_l
        score ← HD-NRPA(l − 1, p)
        if score better than Score_{l,B_l} then
            for b' = 1, ..., B_{l−1} then
                if ¬ Similar(Score_{l,b'}, Length_{l,b'}, l) then
                and Score_{l−1,b'} better than Score_{l,B_l} then
                    insert (Score_{l−1,b'}, Length_{l−1,b'},
                            Code_{l−1,b'}, Succ_{l−1,b'}) into Beam_l
        if (i > Θ) then
            HD-Adapt (l, p'_l)
    return Score_{l,1}
end
```

Figure 4: Beam-NRPA with high diversity.

.

```
procedure Similar(score s, length r, level l)
begin
    for b = 1, ..., B_l
        if Score_{l,b} = s ∧ Length_{l,b} = r then
            return true
    return false
end
```

Figure 5: Example of applied similarity measure.

.

the rollout to be retrieved in later calls of the policy adaptation, which would results in a significant overhead in space and time.

## Improving Diversity in the Policy Adaptation

We refine the beam search by a reduction of elements eligible to be included in the beam. Therefore, we use $(c_j, c_i) \in Beam_{l,1..b-1}$ to denote that the best rollout code (defined by $(c_j, c_i)$) in a given level is already present in the prefix of the beam to bucket $b$ in level $l$. This avoids overly stressing good solutions that have already influenced the policy to be learnt. We also do not want to update elements twice. The according code is shown in Fig. 6. The main function *HD-Adapt* calls the function *HD-Other* which works as a filter, and collects the codes of moves that should be used to change the policy.

We used simple arrays for the data structure to check that a code and set of successor codes is contained in the beam and thus learnt already. Profiling revealed that a significant part of the running time is spent here. Surely, a hash map would be more efficient for checking $(c_j, c_i) \in Beam_{l,1..b-1}$ . However, the algorithm has to be modified as the hash map then has to support deletion, given that elements in the buckets being dominated by incoming solutions are removed from the beam, and, thus, do no longer serve for

```
procedure HD-Other(level l, index b, i, j)
begin
    L ← ∅
    for c_j ∈ Succ_{l,b,i}
        if (c_j, c_i) ∉ Beam_{l,1..b−1} then
            L ← L ∪ {c_j}
    for b' = b + 1, ..., B_l, c_{i'} ∈ Code_{l,b'}
        if c_i = c_{i'} then
            for c_{j'} ∈ Succ_{l,b',i'}
                if c_{j'} ∉ L ∧ (c_{j'}, c_i) ∉ Beam_{l,1..b−1} then
                    L ← L ∪ {j'}
    return L
end

procedure HD-Adapt(level l, policy p, policy p')
begin
    p' ← p
    for b ∈ 1, ..., B
        for c_i ∈ Code_{l,b}
            if c_i ∉ Beam_{l,1..b−1} then
                p'[c_i] ← p[c_i] + α
                L ← HD-Other(l, b, i, j)
        z ← 0
        for c ∈ L
            z ← z + exp(p[c])
        for c ∈ L
            p'[c] ← p'[c] − α · exp(p[c])/z
end
```

Figure 6: Policy adaptation within HD-NRPA.

duplicate detection in form of membership queries.

## Case Study: Same Game

The *same game* (Fig. 8) is an interactive game frequently played on hand-held devices. The input a board with $k$ colored tiles in an arrangement of $n \times m$ (usually, $n = m = 15$ and $k = 5$). Tiles can be removed, if they form a connected group of $l > 1$ elements. The reward of the move is $(l-2)^2$ points. If a group of tiles is removed, others fall down. If a column becomes empty, others move to the left, so that all non-empty columns are aligned. The objective is to maximize the total reward until no more move is possible. Total clearance yields an additional bonus of 1,000 points.

For a growing size of the board, the problem is known to be hard (Biedl et al. 2001). It is solvable in polynomial time for one column of tiles but NP-complete for two or more columns and five or more colors of tiles, or five or more columns and three or more colors of tiles.

Successor generation and evaluation of the score has to reach out for the tiles that have the same color. Fig. 9 illustrates a recursive implementation for counting the number of successors. To assist the compiler, in the tuned implementation of the *same game* we use an explicit stack for building the moves. Termination can be checked faster by testing each of the four directions of every tile location for having the same color.

Let us briefly look to the efficiency of the implementa-

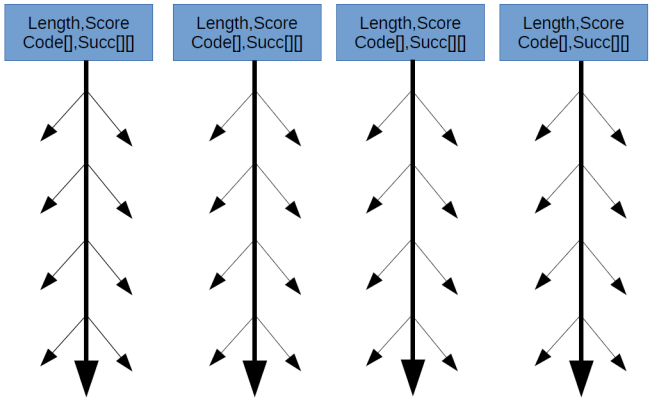


Figure 7: Visualization of the information stored in HD-NRPA; buckets stand for the beam, thin arrors indicate successors (codes, stored in $Succ$), the thick arrow the best solution (codes, stored in $Code$). Duplicates are checked, comparing *forks* of state and successor states.

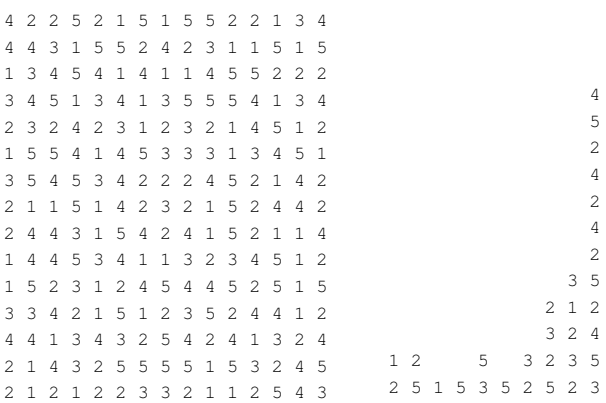


Figure 8: Initial and terminal position in the *same game*.

tion. Let $n^2$ be the board's total number of cells and $t$ be the number of tiles in a given move. The critical aspect is the adjustment of the board according to the gravity of tiles. For each tile remove we *bubble* the blank upwards.

- *LegalMoves* (Fig. 10), once applied for each state: The construction of on move including all stack operations is proportional to the size $t$ of each color group $O(t)$, by the virtue of recording tiles visiting for one state the total of generatng all successors is $O(n^2)$.
- *Play* (Fig. 11), once applied for each state: Besides the removal executing a move is proportional to $t$. Removal of a tile group with adjustment in one column (one for each move): $O(mt)$, for an amortized total of $O(n^3)$ in the playout, as at most $n^2$ tiles can be removed. Removal of one column (selective execution): $O(n^2)$, but the work amortizes, at most $n$ columns that can be removed in the playout. Prior to the removal we sort the tiles affected in each move, which in total (based on the fact that $N = N_1 + \ldots + N_k$ implies $\sum_{i=1}^{k} N_i \lg N_i = O(N \lg N)$) is bounded by $O(n^2 \lg n^2) = O(n^2 \lg n)$.

**procedure** Count(index i, color c)
**begin**
  **if** $Outside(i,j) \vee\ color[i] \neq c \vee visited[i]$
    **return** 0
  $visited[i] \leftarrow 1$
  **return** 1+ Count$(i+1,c)$+ Count$(i-1,c)$+
    Count$(i+n,c)$+ Count$(i-n,c)$
**end**

Figure 9: Counting tiles of color $c$ present at $i$.

- *Terminate* (Fig. 12), once applied for each rollout: $O(n^2)$.

The selection of successors is based on the computation of legal moves and the roulette rule selection, where the latter is dominated by the former. In summary, we obtain the following result.

**Theorem 1 (Complexity Same Game Playout)** *Let $n \times n$ be the board's dimensions of a* same game. *The time for generating a solution of length $l$ in one playout is bounded by $O(ln^2 + n^3)$.*

There are simple refinements to the above algorithm to improve practical performance, but the do not affect the overall complexity of the algorithm.

Table 1 shows the scores in a level 4 (iteration 100) HD-NRPA and 30 $\times$ level 3 (iteration 100) HD-NRPA searches both obtained with beam width 10 and initial offset for learning 10. This is compared to NRPA and NMCS. An entire level 4 search uses selective policy and has been reported in (Cazenave ), while 30 level 3 searches finish in about two hours on our computer[4].

The sum of the high scores of HD-NRPA is 81706 (+144 if the 30 level 3 searches are included). While this is best wrt. all published results on the game, it is still inferior to the results published in the Internet[5]. Little is known about the holders of these records. However, we could exchange emails with a record holder who told us he is using beam search with a complex domain specific evaluation function.

We can see that improving the diversity generally gives better results than NMCS and NRPA, even though, through randomization, there are problem instances where the opposite is true.

## Case Study: Snake-in-the-Box

The *snake-in-the-box* problem is a longest path problem in a $d$-dimensional hypercube. The design of a long snake has impact for the generation of improved error-correcting codes. During the game the snake increases in length, but

[4]We used one core of an Intel® Core™ i5-2520M CPU @ 2.50GHz × 4. The computer has 8 GB of RAM but all invocations of the algorithm to any problem instance used less than 10 MB of main memory. Moreover, we had the following software infrastructure. Operating system: Ubuntu 14.04 LTS, Linux kernel: 3.13.0-74-generic, the compiler: `g++` version 4.8.4, and the compiler options: `-O3 -march=native -funroll-loops -std=c++11 -Wall`

[5]`http://www.js-games.de/eng/games/samegame`

```
procedure LegalMove(Move moves[], length l)
begin
    succs ← 0
    visited ← (0..0)
    if only one move for tabu color then
        tabu ← blank
    for i = 0..n^2 − 1
        if color[i] ≠ blank then
            if ¬visited[i] then
                moves[succs] ← buildMove(i)
                if |moves[succs]| > 1 then
                    if color[i] = tabu then
                        if |moves[succs]| ≤ 2 ∧ l > 10
                            succs ← succs + 1
                    else succs ← succs + 1
    if succs = 0 then
        for i = 0..n^2 − 1
            if color[i] ∉ {blank, tabu} then
                if ¬visited[i] then
                    moves[succs] ← buildMove(i)
                    succs ← succs + |moves[succs]| > 1
    if succs = 0 then
        visited ← (0..0)
        for i = 0..n^2 − 1
            if color[i] ≠ blank then
                if ¬visited[i] then
                    moves[succs] ← buildMove(i)
                    succs ← succ + |moves[succs]| > 1)
    return succs
end
```

Figure 10: Generating the successors in the *same game*.

```
procedure Play(Move move)
begin
    Sort(move)
    for i in 0..|move| − 1
        Remove(i)
    c ← 0
    for i = 0..n − 1
        if column c is empty
            RemoveColumn(c)
        else
            c ← c + 1
    Score ← Score + (|move| − 2)^2
    if board is empty
        Score ← Score + 1000
    end
end
```

Figure 11: Executing a move in the *same game*.

```
procedure Terminal()
begin
    for i = 0..n^2 − 1
        if color[i] ≠ blank then
            if possibleMove(i) then
                return 0
    return 1
end
```

Figure 12: Termination criterion in the *same game*.

must not approach any of its previous visited vertices with Hamming distance 1 or less.

The formal definition of the problem and its variants as well as heursitic search techniques for solving it are studied by (Palombo et al. 2015). The HD-NRPA implementation applies bit manipulation to integers in $0..2^d-1$. Information on snake visits are kept in a perfect hash table of size $2^d$. One optimal solution of length 50 for $d = 7$ is as follows: 0, 1, 33, 35, 43, 42, 10, 26, 27, 25, 57, 56, 48, 52, 53, 55, 63, 62, 126, 122, 123, 115, 113, 81, 80, 88, 92, 93, 95, 87, 86, 22, 6, 7, 15, 13, 12, 44, 108, 104, 105, 73, 75, 67, 66, 98, 102, 103, 101, 69, 68,

There are known generalization to the problem. First, instead of having a Hamming distance of at least $k = 2$ for the incrementally growing head to all previous nodes of the snake (except the ones preceding the head), one may impose a minimal Hamming distance $k > 2$ to all previous nodes (inducing a Hamming sphere that must not be revisited). In Fig. 3 we give the best-known solutions lengths for the $(k, n)$ snake problem, where an asterisk (*) denoting that the optimal solution is known. Our validation of the results in generating a solution with HD-NRPA that matches the given bound is indicated with a $v$. For the first problem not solved, the best solutions are shown in brackets (all within one hour, $(11, 5) = 39$ within two days of computation in about 3.3 billion rollouts).

There is another variant, which asks for a closed cycle, by means that the snake additionally has to bite its own tail at the end of its journey. The algorithm's implementation has to take care that this is in fact possible. In Fig. 3 we give the best-known solutions lengths and our validation results. In summary, using HD-NRPA we could validate all but three optimal solutions in the snake- and coils-in-the-box problems. Approximate soluition lengths for the first unsolved problem are shown in brackets. Unfortunately, we have not generated any solution better than the known bounds.

## Case Study: VRP

In the *vehicle routing problem* (VRP) we are given a fleet of vehicles, a depot, and a time delay matrix for the pairwise travel between the customers' locations, service times, time windows and capacity constraints, the task is to find a minimized number of vehicles with a minimized total distances that satisfies all the constraints. Clearly, by choosing only one vehicle, VRP extends the capacitated traveling salesman with time windows. We chose instances to the Solomon VRPTW benchmark for our experiments[6]. It containts a well-studied selection of (N=)100-city problem instances. Different solvers have contributed to the state-of-the-art.

[6] https://www.sintef.no/projectweb/top/vrptw/solomon-benchmark http://web.cba.neu.edu/~msolomon/problems.htm

| ID | NMCS(4) | NRPA(4) | HD-NRPA(4) | HD-NRPA(3) |
|---|---|---|---|---|
| 1 | 3121 | **3179** | 3145 | 3133 |
| 2 | 3813 | **3985** | **3985** | 3969 |
| 3 | 3085 | 3635 | **3937** | 3663 |
| 4 | 3697 | **3913** | 3879 | 3887 |
| 5 | 4055 | 4309 | **4319** | 4287 |
| 6 | 4459 | **4809** | 4697 | 4663 |
| 7 | **2949** | 2651 | 2795 | 2819 |
| 8 | **3999** | 3879 | 3967 | 3921 |
| 9 | 4695 | 4807 | **4813** | 4811 |
| 10 | **3223** | 2831 | 3219 | 2959 |
| 11 | 3147 | 3317 | **3395** | 3211 |
| 12 | 3201 | 3315 | **3559** | 3461 |
| 13 | 3197 | **3399** | 3159 | 3115 |
| 14 | 2799 | 3097 | **3107** | 3091 |
| 15 | 3677 | 3559 | **3761** | 3423 |
| 16 | 4979 | 5025 | **5307** | 5005 |
| 17 | 4919 | **5043** | 4983 | 4881 |
| 18 | 5201 | 5407 | **5429** | 5353 |
| 19 | 4883 | 5065 | **5163** | 5101 |
| 20 | 4835 | 4805 | 5087 | **5199** |
| Sum | 77934 | 80030 | **81706** | 74753 |

Table 1: Results in the *same game*.

| $k/d$ | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 3 | 4*v | 3*v | 3*v | 3*v | 3*v | 3*v |
| 4 | 7*v | 5*v | 4*v | 4*v | 4*v | 4*v |
| 5 | 13*v | 7*v | 6*v | 5*v | 5*v | 5*v |
| 6 | 26*v | 13*v | 8*v | 7*v | 6*v | 6*v |
| 7 | 50*v | 21*v | 11*v | 9*v | 8*v | 7*v |
| 8 | 98*(95) | 35*v | 19*v | 11*v | 10*v | 9*v |
| 9 | 190 | 63(55) | 28*v | 19*v | 12*v | 11*v |
| 10 | 370 | 103 | 47*(46) | 25*v | 15*v | 13*v |
| 11 | 707 | 157 | 68 | 39*v | 25*v | 15*v |
| 12 | 1302 | 286 | 104 | 56(54) | 33*v | 25*v |
| 13 | 2520 | 493 | 181 | 79 | 47(46) | 31v |

Table 2: Best known results in snakes-in-the-box validated with HD-NRPA.

VRPs in practice are complex. For example, instead of the straight-line distances shortest path on a road network have to be precomputed, leaving a distance matrix to be forwarded to the VRP solver. Often concurrent pickups while delivering items to customers are requested, which have a immediate effect to the violation of capacity constraints. Similarly, for courier express services, items are collected at one site and brought to another. Additionally, there are same-day delivery requirements and reglementations of drivers breaks. The point we want to stress is that all of these additional constraints can be added into a VRP solver based on random playouts like NRPA, as it incrementally generates a tour.

Our implementation of the problem is based on the simple observation that a tour with $V$ vehicles can be generated by a single vehicle, where the time (makespan) and the capacity of the vehicle are reset at each visit of the depot. Of course, in difference to all other cities the depot is allowed to be visited more times. In the implementation the $i$-th visit to the depot gets the ID $i$ and has to be revisited. The tour again has size $N + V$ but the range of stored index of a city has increased from This imposes an order to set of depot IDs in every tour to $0, 1, \ldots, V-1, 0$. This form of symmetry reduction saves about factor $V!$ for the permutations of the depot visits. The solver has the selective strategy that whenever a candidate city invalidates reaching another city it is discarded from the successor set. We selected a level 5/50 search with threshold zero to start learning.

| $k/d$ | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|
| 3 | 6*v | 6*v | 6*v | 6*v | 6*v | 6*v |
| 4 | 8*v | 8*v | 8*v | 8*v | 8*v | 8*v |
| 5 | 14*v | 10*v | 10*v | 10*v | 10*v | 10*v |
| 6 | 26*v | 16*v | 12*v | 12*v | 12* | 12*v |
| 7 | 48*v | 24*v | 14*v | 14*v | 14* | 14*v |
| 8 | 96*(92) | 36*v | 22*v | 16*v | 16* | 16*v |
| 9 | 188 | 64(55) | 30*v | 24v* | 18*v | 18*v |
| 10 | 358 | 102 | 46*v | 28v* | 20*v | 20*v |
| 11 | 668 | 160 | 70(64) | 40v* | 30*v | 22*v |
| 12 | 1276 | 288 | 102 | 60(56) | 36*v | 32*v |
| 13 | 2468 | 494 | 182 | 80 | 50*v | 36*v |

Table 3: Best known results in coils-in-the-box and validated with HD-NRPA.

We could repeat the experiment of solving r101 in our implementation and found the optimal solution of cost $1650.79$ in about $20$ minutes after $625$ thousand playouts. With 20 vehicles we a found slightly better solutions than this one, but the published results often assume a hierarchical objective of first reducing the number of vehicles, and only after that, reducing the score.

With about 2.5 days of computation we could solve the r102 problem. After $215.125$ million rollouts in total, we then found a new high score $1486.664889$, slightly improving the reported best solution. The sequence of cities we found was 73, 22, 72, 54, 24, 80, 12, 0, 65, 71, 71, 20, 32, 70, 0, 92, 37, 98, 91, 16, 86, 85, 97, 13, 0, 83, 45, 61, 84, 5, 60, 89, 0, 94, 96, 99, 6, 0, 50, 33, 30, 51, 9, 67, 1, 0, 14, 44, 38, 43, 100, 95, 0, 27, 69, 76, 79, 68, 0, 52, 7, 11, 19, 49, 48, 82, 0, 28, 29, 78, 34, 35, 3, 77, 0, 62, 88, 8, 46, 17, 93, 59, 0, 36, 47, 18, 0, 39, 23, 67, 55, 4, 25, 26, 0, 63, 64, 90, 10, 31, 0, 87, 57, 2, 58, 0, 40, 53, 0, 42, 15, 41, 75, 56, 74, 21, 0.

In about a week of computation and more than 550 million rollouts we could not finish solving the r103 problem. Our best solution was 1332.77670, while the best known has value 1292.68. The learning process of the cost function is visualized in Fig. 13. We see that even after considerable time of no visible progress, there is continuation in the solving process. Fig. 14 compares the different single-agent Monte Carlo search processes for the first 100 thousand playouts of the r101 problem. We see that HD-NRPA shows the fastest learning progress.

## Conclusion

Nested Monte-Carlo tree search is a class of random search algorithms that has lead to a paradigm shift in AI game play-

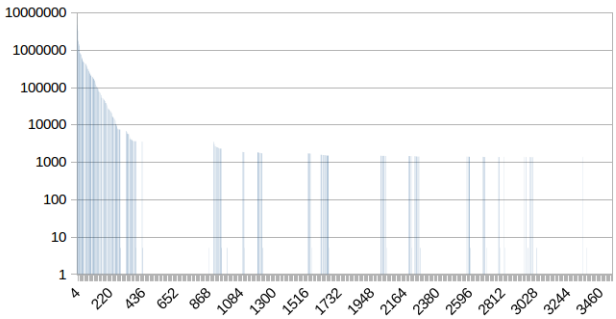


Figure 13: Learning curve solving a VRP with HD-NRPA (y-axis shows the change in the score, x-axis denotes the number of completed level 4 search).

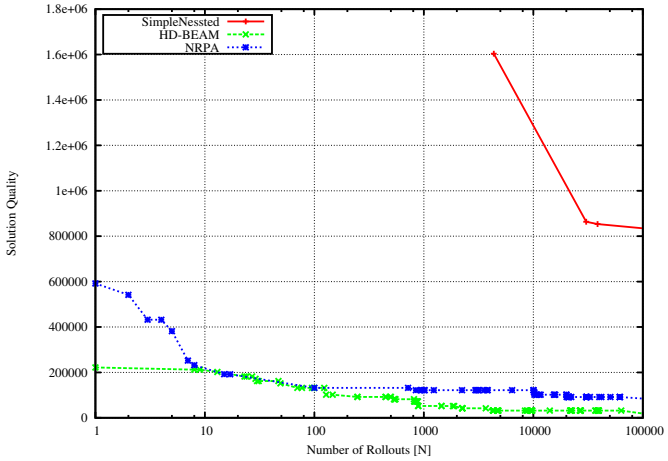


Figure 14: Comparing the learning in VRP of Nested MCS, NRPA, and HD-NRPA.

ing from enumeration to randomization, and nested rollout policy adaptation (NRPA) has proven to be a viable option to solve hard combinatorial problems, combining random exploration with learning.

In this paper we proposed HD-NRPA, designed to add more diversity to the NRPA search, making it faster in several domains. A number of implementation refinements and a more careful handling of the diversity of solutions in the beam made the algorithm perform convincingly across three domains. For the same game we exemplified the interface with the generic solver and analyze the complexity of one playout. Besides elaborating on the proposed setting and its impact, for the eager algorithm engineer, we also provide pseudo-code implementations.

## References

Biedl, T. C.; Demaine, E. D.; Demaine, M. L.; Fleischer, R.; Jacobsen, L.; and Munro, J. I. 2001. The complexity of clickomania. *CoRR* cs.CC/0107031.

Bouzy, B. 2006. An abstract procedure to compute weak Schur number lower bound. In *Computers and Games*.

Bouzy, B. 2013. Monte-carlo fork search for cooperative path-finding. In *IJCAI-Workshop on Computer Games Workshop*, 1–15.

Bouzy, B. 2015. An experimental investigation on the pancake problem. In *IJCAI-Workshop on Computer Games*.

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2004. A survey of Monte Carlo tree search methods. In *IEEE Transactions on Computational Intelligence and AI in Games*, volume 4, 1–43.

Cazenave, T., and Teytaud, F. 2012. Beam nested rollout policy adaptation. In *ECAI-Workshop on Computer Games*, 1–12.

Cazenave, T. Nested rollout policy adaptation with selective policies. In *This volume*.

Cazenave, T. 2009. Nested monte–carlo search. In *IJCAI*, 456–461.

Cazenave, T. 2012. Monte-Carlo beam search. *IEEE Transactions on Computational Intelligence and AI in Games* 4(1):68–72.

Edelkamp, S., and Gath, M. 2014a. Monte-Carlo tree search for 3d packing with object orientation. In *KI*.

Edelkamp, S., and Gath, M. 2014b. Pickup-and-delivery problems with time windows and capacity constraints using nested Monte-Carlo search. In *ICAART*.

Edelkamp, S.; Gath, M.; Cazenave, T.; and Teytaud, F. 2013. Algorithm and knowledge engineering for the TSPTW problem. In *IEEE SSCI*.

Gath, M.; Herzog, O.; and Edelkamp, S. 2013. Agent-based planning and control for groupage traffic. In *IEEE-CEWIT*.

Huang, S.-C.; Arneson, B.; Hayward, R. B.; Mueller, M.; and Pawlewicz, J. 2013. Mohex 2.0: A pattern-based MCTS Hex player. In *Computers and Games*, 60–71.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *ECML*, 282–293.

Palombo, A.; Stern, R.; Puzis, R.; Felner, A.; Kiesel, S.; and Ruml, W. 2015. Solving the snake in the box problem with heuristic search: First results. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015, 11-13 June 2015, Ein Gedi, the Dead Sea, Israel.*, 96–104.

Rosin, C. D. 2011. Nested rollout policy adaptation for Monte-Carlo tree search. In *IJCAI*, 649–654.

Tesauro, G. 1995. Temporal difference learning and TD-Gammon. *Communications of the ACM* 38(3):56–58.