Improved Diversity in Nested Rollout Policy Adaptation

Stefan Edelkamp[2] and Tristan Cazenave[1]

[1] Fakultät Mathematik und Informatik, Universität Bremen
Am Fallturm 1, 28359 Bremen, Germany
[2] LAMSADE - Université Paris-Dauphine
Place du Maréchal de Lattre de Tassigny 75775 Paris Cedex 16

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 yields *nested rollout with policy adaptation* (NRPA), while Beam-NRPA keeps a bounded number of solutions in each recursion level. In this paper we propose refinements for Beam-NRPA that improve the runtime and the solution diversity.

Introduction

Cazenave [4] has invented *nested Monte-Carlo search* (NMCS), a randomized search algorithm inspired by UCT [12] but specifically designed to solve single-player games. Instead of relying on a playout at each search tree leaf, the decision-making in level l of the algorithm relies on a level $(l-1)$ search for its successors.

With *nested rollout policy adaptation* (NRPA), Rosin [14] came up with the idea to learn a policy within the recursive procedure. NRPA has been very successful in solving a variety of optimiztation problems, including puzzles like *morpion solitaire*, but also hard optimization tasks in logistics like *constraint traveling salesman problems* [9] combined *pickup-and-delivery tasks* [8], *vehicle routing* [10], and *container packing* [7] 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 [5] is a combination of memorizing a set of best playouts instead of only one best playout at each level. This set is called a beam and all the positions in the set are developed. Beam search carries over to improve NRPA, enforcing an increased

diversity in a set of solutions. In Beam-NRPA [6], for each level of the search, the algorithm keeps a bounded number of solutions together with their policies in the recursion tree. In selected applications, Beam-NRPA improves over NRPA.

In this paper, we reengineer the implementation of Beam-NRPA. We show that the solution quality can be improved by applying a selection of refinements. In particular, we study more closely how to increase the diversity.

NRPA and Beam-NRPA

NRPA is a randomized optimization scheme that belongs to the wider class of Monte-Carlo tree search (MCTS) algorithms [3]. The main concept of MCTS is the random *playout* (or *rollout*) of a position, whose outcome, in turn, changes the likelihood of generating successors in subsequent trials. Prominent members in this class of reinforcement learning algorithms are *upper confidence bounds applied to trees* (UCT) [12], and *nested monte-carlo search* (NMCS) [4]. MCTS is state-of-the-art in playing many two-player games [11] or puzzles [2], 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.

What makes NRPA [14] different to UCT and NMCS is the concept of learning a policy through an explicit mapping of moves to selection probabilities. The pseudo-code of the recursive search procedure is shown in Fig. 1 (left). 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. If r is not better than *best*, on the first glance it looks like the same call to NRPA is performed within the loop on i. However, rollouts change due to randomness, and policy adaptation in other level of the recursion.

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 Beam-NRPA algorithm [5] is shown in Fig. 1 (right). Each solution is stored together with its score and the policy that was used to generate it. Better solutions are inserted into a list that is sorted wrt. 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. 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 the algorithm Beam-NRPA to perform $\prod_{l=1}^{L} N_l B_l$ rollouts.

Refinements

We propose several refinements to Beam-NRPA.

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

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

Fig. 1. NRPA and Beam NRPA. To cover both minimization and maximization problems, score ordering is imposed by means of implementing *Init*, *better*, and *best*. SL is implemented as a sorted list.

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 information 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 [14].

The implementation in Fig. 2 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

```
procedure NRPA-Adapt (rollout best,
level l, policy p, policy p')
    p' ← p
    for i ∈ best_l
        c_i ← code of move best_{l,i}
        Succ_{l,i} ← successors codes of best_{l,i}
            p'[c_i] ← p'[c_i] + α
        z ← 0
        for c' ∈ Succ_{l,i}
            z ← z + exp(p[c'])
        for c' ∈ Succ_{l,i}
            p'[c'] ← p'[c'] − α · exp(p[c'])/z
```

```
procedure NRPA-Adapt-Improved(level
l, policy p, policy p')
    for c_i ∈ Code_l
        p'[c_i] ← p'[c_i] + α
    z ← 0
    for c' ∈ Succ_{l,i}
        z ← z + exp(p[c'])
    for c' ∈ Succ_{l,i}
        p'[c'] ← p'[c'] − α · exp(p[c'])/z
```

Fig. 2. Old and new policy adaptation procedure for NRPA reproducing rollout data and its successors (left) and recorded solution information (right); z is used for normalization, α is the learning rate, usually $\alpha = 1$.

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 is visualized in Fig. 6. 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 natual aim is to keep solutions in the beam 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. 3 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. 4 looks at the score and the length of the rollout.

The concept of similarity implies a formal characterization of *solution diversity*. Let $\mathcal{S}$ be a set of solutions of an optimization problem with and let *Similar* be a pairwise similarity score (being large for high similarity and small

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

Fig. 3. Beam-NRPA with high diversity; B_l is the size of the beam $Beam_l$ maintained in level l, $Score_{l,b}$ is the score of the rollout in bucket b_l in level l, $Length_{l,b}$ is the length of the rollout in b_l, $Code_{l,b}$ is an array with codes for the rollout in bucket b_l, $Succ_{l,b}$ is a matrix for the successor codes for the rollout in bucket b_l.
.

for low similarity) between every two solution s_i and s_j in $\mathcal{S}$, then the diversity is defined as the sum of the pairwise similarities, i.e., $Diversity(\mathcal{S}) = \sum_{s_i,s_j \in \mathcal{S}} Similar(s_i, s_j)$. 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 exchange 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 Θ_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 stopping rule rejects the first applicants after the interview and then stops at the first applicant, who is better than every applicant interviewed so far.

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,

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

Fig. 4. Example of applied similarity measure, returning true iff both the score and the length of a solution matches one in the beam $Beam_l$ of size B_l.
.

or storing the full state in 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 beam NRPA 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. 5. The main function *Diversity-Adapt* calls the function *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 duplicate detection in form of membership queries.

Given that the selection strategy of the successors does not prune away moves that are required to generate an optimal rollout sequence, NRPA and Beam-NRPA are *probabilistically complete* in the sense that an optimal solution can eventually be found. This, however, does not imply any performance quality like the ϵ-optimality of the resulting search algorithms.

Experiments

Same Game The *same game* (Fig. 7) is an interactive game frequently played on hand-held devices. The input is an $n \times m$ board with tiles each of which having one (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.

```
procedure Other(level l, index b, i, j)
    LC ← ∅
    for c_j ∈ Succ_{l,b,i}
        if (c_j, c_i) ∉ Beam_{l,1..b−1}
            LC ← LC ∪ {c_j}
    for b′ = b + 1, ..., B_l, c_{i′} ∈ Code_{l,b′}
        if c_i = c_{i′}
            for c_{j′} ∈ Succ_{l,b′,i′}
                if c_{j′} ∉ LC ∧ (c_{j′}, c_i) ∉ Beam_{l,1..b−1}
                    LC ← LC ∪ {j′}
    return LC

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

Fig. 5. Policy adaptation within Diversity-NRPA, with function *Other* checking for *fork* duplicates; CL and LC are list of codes, $(c_j, c_i) \in Beam_{l,1..b-1}$ is shorthand for checking that the code c_i and successor codes c_j match the one stored in any bucket smaller than b of $Beam_l$.

The problem is known to be hard [1]. 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.

Table 1 shows the scores in a level 4 (iteration 100) Diversity-NRPA and 30 × level 3 (iteration 100) Diversity-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 takes about half a day of computation, while 30 level 3 searches finish in about two hours on our computer[3]. The sum of the high scores of Diversity-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

[3] 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`

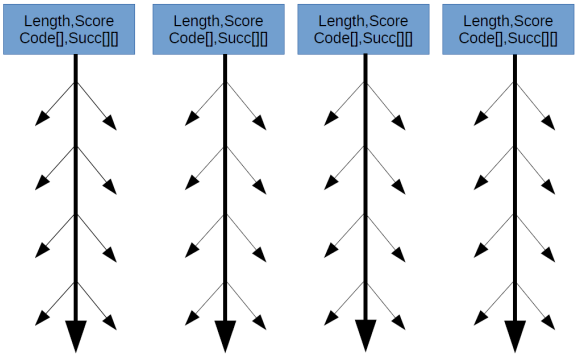

Fig. 6. Sketch of information that is stored in a beam of Diversity-NRPA; the buckets on top stand for the beam, thin arrows indicate successors (codes, stored in *Succ*), the thick arrow the best solution (codes, stored in *Code*). A duplicate are checked wrt. *forks* of state and set of successor states.

```
4 2 2 5 2 1 5 1 5 5 2 2 1 3 4
4 4 3 1 5 5 2 4 2 3 1 1 5 1 5
1 3 4 5 4 1 4 1 1 4 5 5 2 2 2
3 4 5 1 3 4 1 3 5 5 5 4 1 3 4
2 3 2 4 2 3 1 2 3 2 1 4 5 1 2
1 5 5 4 1 4 5 3 3 3 1 3 4 5 1
3 5 4 5 3 4 2 2 2 4 5 2 1 4 2
2 1 1 5 1 4 2 3 2 1 5 2 4 4 2
2 4 4 3 1 5 4 2 4 1 5 2 1 1 4
1 4 4 5 3 4 1 1 3 2 3 4 5 1 2
1 5 2 3 1 2 4 5 4 4 5 2 5 1 5
3 3 4 2 1 5 1 2 3 5 2 4 4 1 2
4 4 1 3 4 3 2 5 4 2 4 1 3 2 4
2 1 4 3 2 5 5 5 5 1 5 3 2 4 5
2 1 2 1 2 2 3 3 2 1 1 2 5 4 3
```

```
                    4
                    5
                    2
                    4
                    2
                    4
                    2
                  3 5
                2 1 2
                3 2 4
1 2     5   3 2 3 5
2 5 1 5 3 5 2 5 2 3
```

Fig. 7. Initial and terminal position in the *same game*.

published in the Internet[4]. 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.

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 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

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

ID	NMCS(4)	NRPA(4)	Diversity-NRPA(4)	Diversity-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*.

	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

	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 2. Best known results in snakes-in-the-box and coil-in-the-box problems validated with Diversity-NRPA (approximate solutions are shown in brackets).

as heursitic search techniques for solving it are studied by [13]. Information on snake visits are kept in a perfect hash table of size 2^d.

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 Table 2 (left) we show the best-known solutions lengths for the (k, n) snake problem, where an asterisk (*) denotes that the optimal solution is known. The validation of the results in generating a solution with Diversity-NRPA is indicated with suffix 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).

Another variant 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 Table 2 (right) the best-known solutions lengths and our validation results are given.

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[5]. It containts a well-studied selection of (N=)100-city problem instances. Different solvers have contributed to the state-of-the-art.

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 (5,50) Diversity-NRPA search with threshold 0 to start learning.

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 found a new high score 1486.664889, slightly improving the reported best solution[6].

In about a week of computation and more than 550 million rollouts we could not finish solving the r103 problem. Our best solution 1332.77670, while the best has value 1292.68. The learning process of the cost function is visualized in Fig. 8. We see that even after considerable time of no visible progress, there

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

[6] 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.

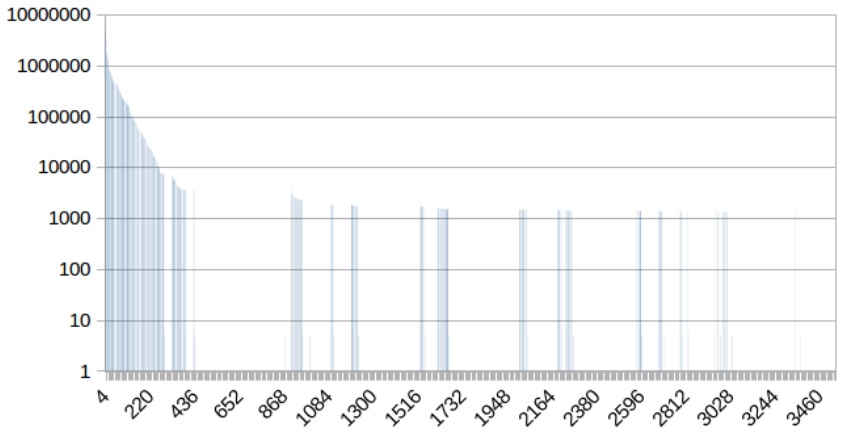

Fig. 8. Learning curve solving a VRP with Diversity-NRPA (y-axis shows the change in the score, x-axis denotes the number of completed level 4 search).

is continuation in the solving process. Fig. 9 compares the different single-agent Monte Carlo search processes for the first 100 thousand playouts of the r101 problem. We see that Diversity-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 playing from enumeration to randomization, and NRPA has proven to be a viable option to solve hard combinatorial problems, combining random exploration with learning. In this paper we proposed to add more diversity to Beam-NRPA search. Together with a number of implementation refinements the algorithm performed convincingly in our benchmark domains.

References

1. T. C. Biedl, E. D. Demaine, M. L. Demaine, R. Fleischer, L. Jacobsen, and J. I. Munro. The complexity of clickomania. *CoRR*, cs.CC/0107031, 2001.
2. B. Bouzy. An experimental investigation on the pancake problem. In *IJCAI-Workshop on Computer Games*, 2015.
3. C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. In *IEEE Transactions on Computational Intelligence and AI in Games*, volume 4, pages 1–43, 2004.
4. T. Cazenave. Nested monte–carlo search. In *IJCAI*, pages 456–461, 2009.

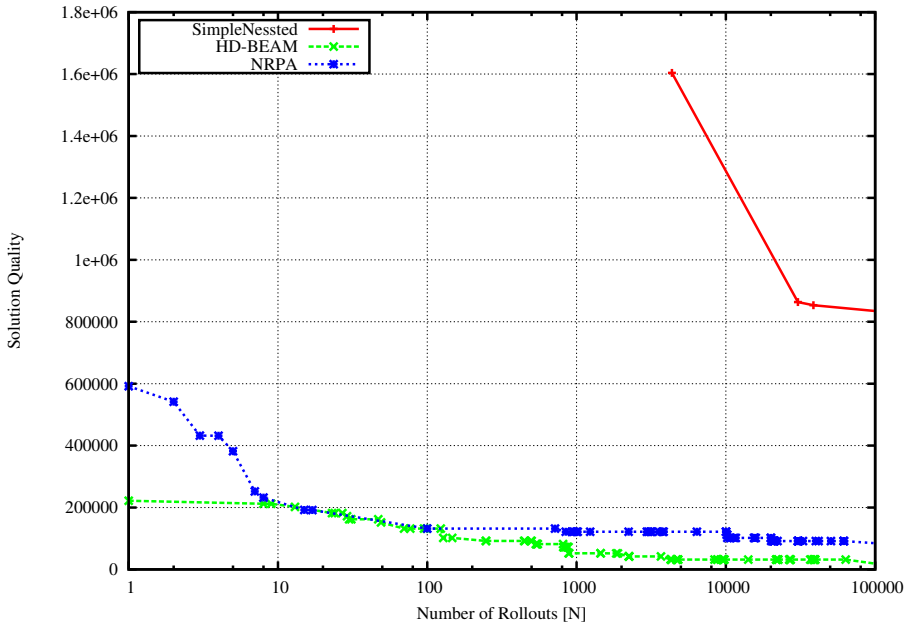

Fig. 9. Comparing the learning in VRP of Nested MCS, NRPA, and Diversity-NRPA.

5. T. Cazenave. Monte-Carlo beam search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):68–72, 2012.
6. T. Cazenave and F. Teytaud. Beam nested rollout policy adaptation. In *ECAI-Workshop on Computer Games*, pages 1–12, 2012.
7. S. Edelkamp and M. Gath. Monte-Carlo tree search for 3d packing with object orientation. In *KI*, 2014.
8. S. Edelkamp and M. Gath. Pickup-and-delivery problems with time windows and capacity constraints using nested Monte-Carlo search. In *ICAART*, 2014.
9. S. Edelkamp, M. Gath, T. Cazenave, and F. Teytaud. Algorithm and knowledge engineering for the TSPTW problem. In *IEEE SSCI*, 2013.
10. M. Gath, O. Herzog, and S. Edelkamp. Agent-based planning and control for groupage traffic. In *IEEE-CEWIT*, 2013.
11. S.-C. Huang, B. Arneson, R. B. Hayward, M. Mueller, and J. Pawlewicz. Mohex 2.0: A pattern-based MCTS Hex player. In *Computers and Games*, pages 60–71, 2013.
12. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293, 2006.
13. A. Palombo, R. Stern, R. Puzis, A. Felner, S. Kiesel, and W. Ruml. 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.*, pages 96–104, 2015.
14. C. D. Rosin. Nested rollout policy adaptation for Monte-Carlo tree search. In *IJCAI*, pages 649–654, 2011.