Parallel Nested Rollout Policy Adaptation

Andrzej Nagórko
Institute of Mathematics
University of Warsaw
Warsaw, Poland
amn@mimuw.edu.pl

***Abstract*—Nested Rollout Policy Adaptation (NRPA) is a Monte Carlo tree search algorithm that excels in the domain of single agent optimization problems. It was applied to a wide class of problems, including vehicle routing, DNA alignment, 3D packing, travelling salesman problem, combinatorial puzzles and more.**

We develop a parallel version of NRPA that replicates results of the sequential version. The parallelization allows us to run deeper calculations. The experimental data shows that depth of the calculation is a deciding factor in the result quality. Earlier parallelization attempts used parallel architecture to run wider, but not deeper, calculations.

We applied the parallel version to the Morpion Solitaire benchmark. To aid parallelization, we used a different best result replacement rule. Using the new rule, on a distributed architecture with 768 cores we obtained an average speedup over a single core computation by a factor 547.48. For the first time a level 6 NRPA computation with 80 iterations per level was finished. It was completed in less than 20 hours of wall-clock time. The different replacement rule was effective. In each run, the record 178-move sequence was found.

***Index Terms*—MCTS, NRPA, parallelization, Morpion Solitaire**

I. Introduction

Nested Rollout Policy Adaptation [18] is a Monte Carlo tree search algorithm. It beats more general Monte Carlo tree search algorithms in the domain of single agent optimization problems. It was applied to a wide class of problems, including vehicle routing [9, 7], DNA alignment [10, 11], 3D packing [8], travelling salesman problem [5, 12, 10] and more [3, 13, 16]. It outclasses MCTS in a benchmark problem of Morpion Solitaire [14, 1, 4, 18], which is notoriously hard for computers. The NRPA paper [18] was awarded the best paper award at IJCAI'11.

The NRPA algorithm is computationally expensive. To achieve record results in Morpion Solitaire on average over a week of single-core computation is required. The core of NRPA algorithm is sequential and seems to be non-parallelizable at the first glance. In [16] a parallel version was developed that uses parallelization to widen the search. However, the experimental data shows that depth of the calculation is a deciding factor in the result quality. In the present paper we introduce a parallel version of NRPA algorithm that is a direct parallelization of the sequential version. It is deterministic and computes the same result that the sequential version does. The parallel speedup allows us to run deeper calculations. In Morpion Solitaire, it allowed us to complete for the first time a level 6 run of NRPA with 80 iterations per level. The computation was completed in less than 20 hours.

To aid parallelization, we used a different best result replacement rule. Using the new rule, on a distributed architecture with 768 cores we obtained an average speedup over a single core computation by a factor 547.48 (measured by a wall-clock time). The different replacement rule was quite effective. In each run, the record 178-move sequence was found.

978-1-7281-1884-0/19/$31.00 ©2019 European Union

II. Optimization problem abstraction

We assume that the optimization problem supports the following three operations.

SIMULATE(P)	- given a policy P, generate a random playout;
ADAPT(P, B)	- given a policy P and a playout B, adapt P to increase the probability of generating B;
MAX(B_1, B_2)	- select a playout with higher reward.

We assume that the following conditions are satisfied.

1) ADAPT converges quickly: an ADAPT with the same playout B repeated few dozen times results in a policy that generates B with probability close to 1.
2) ADAPT is fast: we can compute hundreds of adapts per second.
3) SIMULATE may be much slower: in fact, if it is fast, we replace it with a short sequential NRPA run (that we call an atomic computation) to slow it down.
4) Reward is discrete and does not change often.

Note that condition 4) is a characteristic of more challenging optimization problems, such as Montezuma Revenge. For parallel NRPA, sparsity of rewards aids parallelization, as each improvement of the best sequence usually discards part of the calculations.

In the present paper we used Morpion Solitaire as a benchmark problem, as it was used by Rosin in the original NRPA paper [18]. The benchmark is described in Section V.

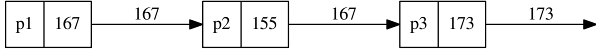
III. Rollout parallelization

A rollout is a reinforcement learning loop displayed as Algorithm 1. A rollout is a basic building block of the NRPA algorithm.

Algorithm 1 A rollout - basic building block of NRPA

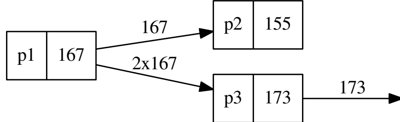
1: **function** ROLLOUT
2: $p_1 \leftarrow \mathbb{1}$ ▷ Initial policy (uniform random)
3: $b_0 \leftarrow \emptyset$ ▷ Best result (so far)
4: **for** $i \leftarrow 1$ to N **do**
5: $r_i \leftarrow$ SIMULATE(p_i)
6: $b_i \leftarrow \max(b_{i-1}, r_i)$
7: $p_{i+1} \leftarrow$ ADAPT(p_i, b_i)
8: **return** b_N

To show how to parallelize ROLLOUT, consider a sample run show below.



The diagram shows three iterations of the rollout loop. In the first iteration, SIMULATE on policy P$_1$ returns a result with value 167, which is stored in r_1 and b_1 (in the diagram we identify a sequence with its reward, for simplicity, so number 167 stands for sequence r_1). Then b_1 is used to ADAPT p_2. In the second iteration, SIMULATE returns a result with value 155, which is stored in r_2, but not in b_2, which stays equal to b_1. Then $b_2 = b_1$ is used to ADAPT p_3. In the third iteration, SIMULATE yields a result with value 173, which is stored as b_3 and b_3 is returned as the final value of the rollout.

We can observe that the calculation can be made faster if we compute simulations from p_2 and p_3 in parallel (after adapting p_3 from p_1 using sequence r_1 twice). This is possible because r_2 does not improve b_1. The following diagram shows a computation of the same rollout in two iterations, with two computations running in parallel.



An algorithm that implements this idea is displayed as Algorithm 2. Note that we do not know beforehand which calculations can be made in parallel, so some calculations are started and then have to be discarded, when we find out that an earlier simulation improved the best sequence.

IV. NESTED ROLLOUT PARALLELIZATION

The NRPA algorithm uses nested rollouts. At level $L > 0$, NRPA replaces a call to SIMULATE with a recursive call to NRPA at level $L-1$. The lower level NRPA calculation uses its parent's policy, instead of a uniform random one. The NRPA algorithm is displayed as Algorithm 3. Note that $L = 1$ NRPA is equivalent to the ROLLOUT function described in the last section.

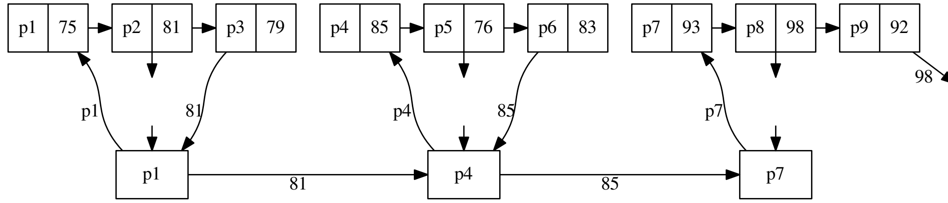
Figure 1a shows a sample run of level 2 NRPA (sequential version that is displayed as Algorithm 3). Note that as before, this process can be parallelized at the lowest level $L = 1$ of simple rollouts, using strategy described in Section III. But we can also parallelize the computation at the higher level $L = 2$. The idea is shown in Figure 1b. Note that we benefit greatly from $L = 2$ parallelization: a sequential version of the algorithm runs in 9 iterations; if we only parallelized at $L = 1$, it would run in 8 time steps; the fully parallel version finishes in 6 iterations. The full algorithm is displayed as in the Appendix Algorithm 4.

Algorithm 2 Parallelization of a rollout with N iterations and K workers. It uses a client-server architecture with message passing and is suitable to use with the MPI library.

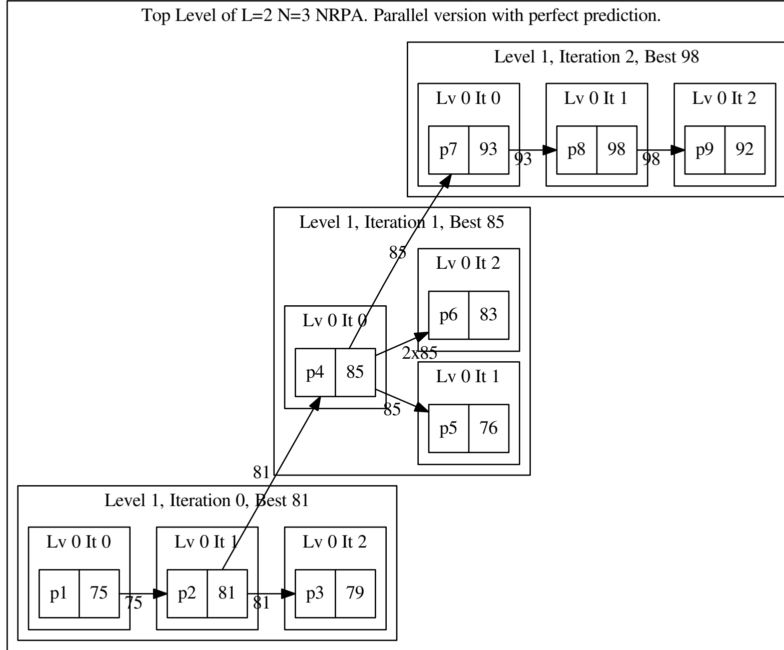
Ensure: $r_i =$ SIMULATE(p_i) ▷ Result r_i is generated by p_i
$b_i = \max(b_{i-1}, r_i)$ ▷ b_i is best up to ith iteration
$p_i =$ ADAPT(p_{i-1}, b_{i-1}) ▷ p_i is adapted with b_{i-1}

1: **procedure** SERVER
2: $p_1 \leftarrow \mathbb{1}$ ▷ Initial policy (uniform random)
3: $b_0 \leftarrow \emptyset$ ▷ Best result at iteration 0 is empty
4: pending $\leftarrow \{1, 2, \ldots, N\}$ ▷ Iterations that are neither finished nor running
5: workers $\leftarrow \{1, 2, \ldots, K\}$
6: finished $\leftarrow \emptyset$
7: **while** #finished $< N$ **do**
8: **while** #workers > 0 and #pending > 0 **do**
9: $i \leftarrow \min(\text{pending})$, pending $\leftarrow$ pending$\setminus\{i\}$
10: $w \leftarrow \min(\text{workers})$, workers $\leftarrow$ workers$\setminus\{w\}$
11: $b_i \leftarrow b_{i-1}$ ▷ Parallelization is achieved here
12: **if** $i > 1$ **then**
13: $p_i \leftarrow$ ADAPT(p_{i-1}, b_{i-1})
14: job$_i$ = UNIQUEID()
15: SEND(target=w, (i=i, id=job$_i$, policy=p_i, seed=seed$_i$))
16: (i, w, id, result) $\leftarrow$ RECEIVE() ▷ Blocking call
17: workers $\leftarrow$ workers $\cup \{w\}$
18: **if** job$_i$ = id **then** ▷ Job was not discarded
19: $r_i \leftarrow$ result
20: finished $\leftarrow$ finished $\cup \{i\}$
21: **if** $r_i > b_i$ **then** ▷ Discard everything after i if we get new b_i
22: $b_i \leftarrow r_i$ ▷ $b_i \leftarrow \max(b_{i-1}, r_i)$
23: **for** $j \leftarrow i+1, i+2, \ldots, N$ **do**
24: delete p_j, b_j, r_j, job$_j$
25: finished $\leftarrow$ finished $\setminus \{j\}$
26: pending $\leftarrow$ pending $\cup \{j\}$
27: **return** b_N

1: **procedure** WORKER(id) ▷ id $\in \{1, 2, \ldots, K\}$
2: **loop**
3: $(i, \text{id}, p, \text{seed}) \leftarrow$ RECEIVE() ▷ Blocking call
4: result $\leftarrow$ SIMULATE(p, seed) ▷ Use same random seed on recalculation
5: SEND(target=SERVER, $(i, w, \text{id}, \text{result})$)



(a) A sample run of $L = 2, N = 3$ sequential NRPA (Algorithm 3). Bottom row shows $L = 2$ rollout. At iteration $i = 1$ we start with uniform random policy p_1, which is passed to a recursive NRPA call at level $L = 1$. At $L = 1$ a rollout is performed (top row, left side) and a result with value 81 is passed back to level 2 rollout. Policy p_4 is adapted from p_1 using result with value 81. Policy p_4 is passed to another recursive NRPA call at $L = 1$; it yields a result with value 85. It is used to adapt policy p_7 from policy p_4. The final recursive call yields result with value 98.



(b) A parallelization of a sample run of $L = 2$ of NRPA discussed above (Algorithm 4). Policy p_4 can be adapted straight from p_2, before first $L = 1$ run is completed, as sequence simulated by p_3 does not improve the best result. Likewise, simulations from policies p_5, p_6 and p_7 can be made in parallel.

Fig. 1. The idea of parallelization of the NRPA algorithm.

Algorithm 3 NRPA algorithm with N iterations and L levels, sequential version from Rosin's paper [18]. The parallel version is displayed as Algorithm 4.

1: **procedure** NRPA(L, p)
2:     **if** $L = 0$ **then**
3:         **return** SIMULATE(p)
4:     **else**
5:         $p_1 \leftarrow p$ ▷ Initial policy (passed by parent)
6:         $b_0 \leftarrow \emptyset$ ▷ Best result (so far)
7:         **for** $i \leftarrow 1$ to N **do**
8:             $r_i \leftarrow$ NRPA$(L-1, p_i)$
9:             $b_i \leftarrow \max(b_{i-1}, r_i)$
10:             $p_{i+1} \leftarrow$ ADAPT(p_i, b_i)
11:         **return** b_N

V. The Experiments - Morpion Solitaire

A. Morpion Solitaire

The Morpion Solitaire is a paper-and-pencil single-player game played on a square grid with the initial configuration of 36 dots depicted in Figure 2. In each move the player puts a dot on an unused grid position and draws a line that consists of four consecutive segments passing through the dot. The line must be horizontal, vertical or diagonal and no segment may be drawn twice, i.e. the moves have to be segment-disjoint. The goal is to find the longest possible sequence of moves. It was proved in [6] that the problem is finite. The best known upper bound on the length of the sequence is 485 [15] and is far away from the best known sequence of length 178.

The problem of finding the longest sequence of moves in the Morpion Solitaire is notoriously hard for computers.

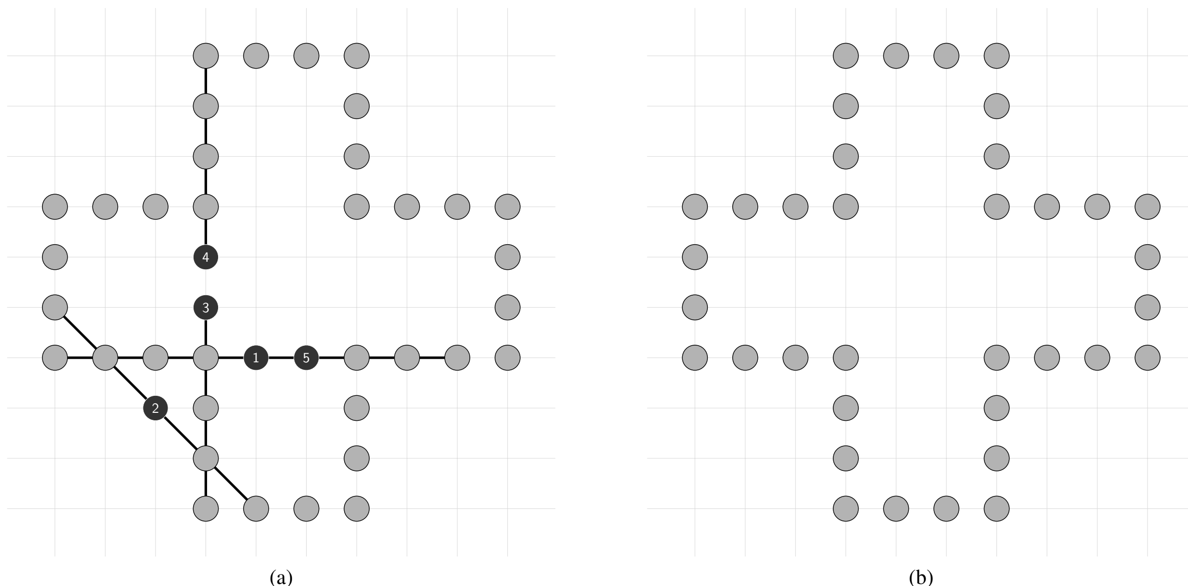


Fig. 2. The initial position of the Morpion Solitaire is depicted on the right. On the left a sequence of 5 moves from the initial grid is played.

For 34 years the longest known sequence was one of 170 moves discovered by hand by Bruneau in 1976. Despite considerable computational effort, until 2010 the computer generated sequences were much shorter. In 2010 the human record was broken, when Rosin [18] obtained the current world record of 178 moves using the NRPA algorithm. The webpage [4] maintained by Boyer contains an extensive and up-to-date information about records in all Morpion Solitaire variants.

B. Policy, adapt and max functions

We used a weight-table based policy and gradient-ascent adapt, faithfully following Rosin's paper [18].

The standard NRPA algorithm always replaces the best sequence if the new one that was found has greater *or equal* length. For greater parallel efficiency, we want to make as little replacements as possible. We found that sequences of equal length often differ by just few moves and replacement by a similar sequence does not aid exploration, while it ruins parallelization. Therefore we ran our experiments with different replacement strategy: we replaced a sequence with a sequence of equal length only if the new sequence had more than 30% different moves than the old one. Note that during atomic computations (a short run of sequential NRPA done at the worker nodes) we used the usual replacement rule. Hence our rule was used on top two or three levels of the computation.

C. Order of computation

We used the following heuristic as the SELECT function in Algorithm 4. We guesstimated a probability that a simulation will beat a sequence of the given length (a probability table was used; probability that a sequence of length 130 will be beaten was set to 10%; to 20% for length 140 and so on). Then for each leaf node SELECT function calculated probability that the search will not change its policy and a node with greatest probability was selected for computation.

D. Atomic calculations

The algorithm was ran on a distributed architecture using Intel's MPI library (2018 version) for inter-process communication. The server code was implemented in Python with C++ extensions and worker code mostly in C++. Because of communication overhead, workers instead of calculating a single SIMULATE call calculated a $L = 2$ or $L = 3$ NRPA run (an *atomic calculation*). For $L = 6, N = 80$ runs a single worker task was $L = 3, N = 80$ NRPA call. In these runs the server had to distribute $80^3 = 512000$ atomic calculations among it workers (this number does not include restarted calculations). A single worker calculated at an approximate speed of 20000 simulations per second. An atomic calculation with $L = 3, N = 80$ took approximately 25 seconds.

E. Implementation

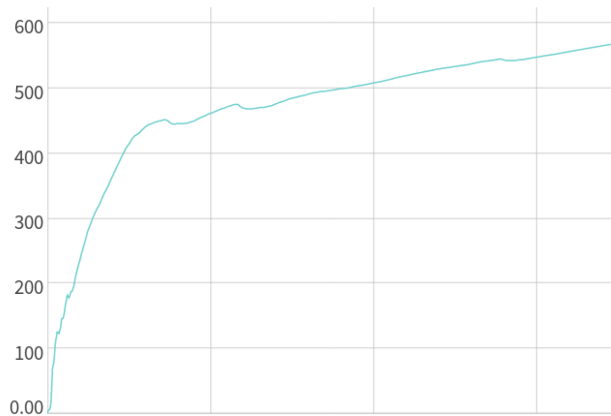
The full implementation is published in a repository

https://github.com/amn/parallel-nrpa/

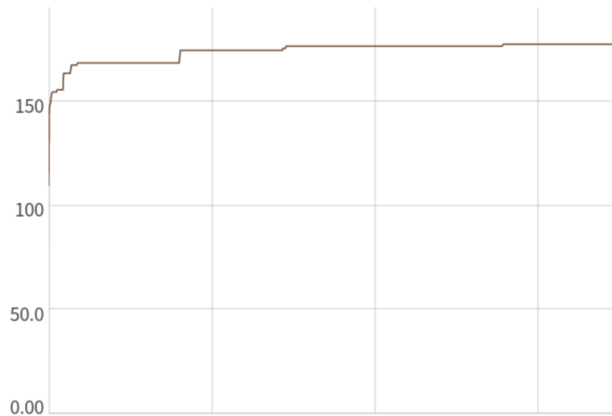
F. Experimental results

We ran 10 calculations with $N = 80, L = 6$. Every calculation found the record sequence of length 178 (it was always equal up to symmetry to the record sequence found by Rosin, see Figure 4). The calculations were ran on 768 cores (with three levels of atomic computations) and the average parallel speedup was 547.48 (71% efficiency).

We ran 20 calculations with $N = 70, L = 6$. The record sequence of 178 moves was found 10 times, a sequence with 177 moves was found 9 times and a sequence of length 172



(a) Parallel speedup over iteration. Efficiency increases when the best sequence stabilizes.



(b) Best sequence over iteration. Sequence of length 178 was found near the end of 20-hour run.

Fig. 3. Statistics from a $L = 6$, $N = 80$ NRPA run.

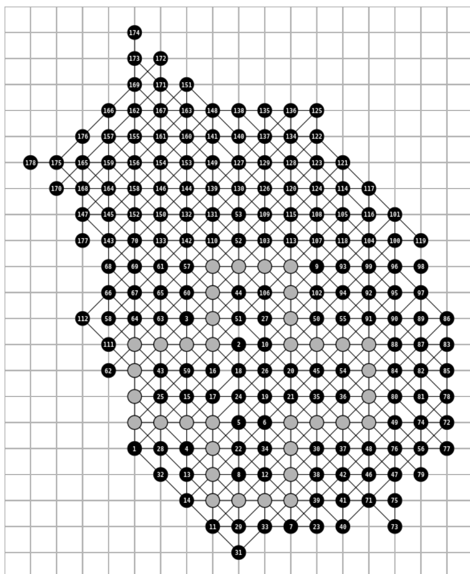


Fig. 4. A 178 moves sequence found by $L = 6$, $N = 80$ run, identical (up to symmetry) to Rosin's record sequence.

was found once. The average parallel speedup on 576 cores was 402.89 (70% efficiency) The average parallel speedup on 376 cores was 288.68 (75% efficiency).

VI. Further work

The parallelization of NRPA is achieved at the top-level of the algorithm. We do not modify the policy and treat it as a black-box component (as described Section II). The algorithm developed in this paper and the speedup obtained by the parallelization allows for a replacement of a computationally inexpensive weight-table based policy (used by the original NRPA) by a much more computationally expensive neural networks. Such an approach was previously successfully applied to MCTS algorithm in the Alpha Go Zero project [2] to create world's best Go player. In a similar fashion, parallelized NRPA with neural networks can be applied to a variety of single-agent problems, such as challenging instances of Atari games, such as Sokoban [17].

Sources

The full source code for experiments reported in the paper is published in a repository

https://github.com/amn/parallel-nrpa/

Acknowledgment

This research would not be possible without a support from the PL-Grid Infrastructure. In particular, we used extensively the Prometheus supercomputer, located in the Academic Computer Center Cyfronet in the AGH University of Science and Technology in Kraków, Poland (4320 Intel Xeon E5-2680 v3 processors, each with 12 cores).

References

[1] Haruhiko Akiyama. *A computer generated Morpion 5T grid consisting of 145 moves*. 2010. URL: http://www.morpionsolitaire.com/English/RecordsGrids5T.htm.

[2] Silver D. et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (2017), pp. 354–.

[3] Hendrik Baier and Mark HM Winands. "Nested Monte-Carlo Tree Search for Online Planning in Large MDPs." In: *ECAI*. Vol. 242. 2012, pp. 109–114.

[4] Christian Boyer. *Morpion Solitaire*. 2015. URL: http://www.morpionsolitaire.com.

[5] Tristan Cazenave and Fabien Teytaud. "Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows". In: *In LION*. 2011, pp. 42–54.

[6] Erik D. Demaine et al. "Morpion Solitaire". In: *Theory Comput. Syst.* 39.3 (2006), pp. 439–453.

[7] Stefan Edelkamp and Max Gath. “Solving single vehicle pickup and delivery problems with time windows and capacity constraints using nested monte-carlo search”. In: *Proceedings of the 6th International Conference on Agents and Artificial Intelligence-Volume 1*. SCITEPRESS-Science and Technology Publications, Lda. 2014, pp. 22–33.

[8] Stefan Edelkamp, Max Gath, and Moritz Rohde. “Monte-Carlo tree search for 3D packing with object orientation”. In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer. 2014, pp. 285–296.

[9] Stefan Edelkamp, Denis Golubev, and Christoph Greulich. “Solving the Physical Vehicle Routing Problem for Improved Multi-robot Freespace Navigation”. In: *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*. Springer. 2016, pp. 155–161.

[10] Stefan Edelkamp and Christoph Greulich. “Solving physical traveling salesman problems with policy adaptation”. In: *2014 IEEE Conference on Computational Intelligence and Games*. IEEE. 2014, pp. 1–8.

[11] Stefan Edelkamp and Zhihao Tang. “Monte-carlo tree search for the multiple sequence alignment problem”. In: *Eighth Annual Symposium on Combinatorial Search*. 2015.

[12] Stefan Edelkamp et al. “Algorithm and knowledge engineering for the TSPTW problem”. In: *2013 IEEE Symposium on Computational Intelligence in Scheduling (CISched)*. IEEE. 2013, pp. 44–51.

[13] Shalom Eliahou et al. “Investigating Monte-Carlo methods on the weak Schur problem”. In: *European Conference on Evolutionary Computation in Combinatorial Optimization*. Springer. 2013, pp. 191–201.

[14] Heikki Hyyrö and Timo Poranen. “New Heuristics for Morpion Solitaire”. In: *preprint*. 2007.

[15] Henryk Michalewski, Andrzej Nagórko, and Jakub Pawlewicz. “485 - A new upper bound for Morpion Solitaire”. In: *CGW@IJCAI*. 2015.

[16] Cazenave T. Negrevergne B. “Distributed Nested Rollout Policy for SameGame”. In: *Computer Games. CGW 2017.* Vol. 818. 2018.

[17] Sébastien Racanière et al. “Imagination-augmented agents for deep reinforcement learning”. In: *Advances in neural information processing systems*. 2017, pp. 5690–5701.

[18] Christopher D. Rosin. “Nested Rollout Policy Adaptation for Monte Carlo Tree Search.” In: *IJCAI*. 2011, pp. 649–654.

Algorithm 4 Parallelization of NRPA with N iterations, L levels and K workers

1: **procedure** SERVER
2: $p_1 \leftarrow \mathbb{1}$ ▷ Root node policy
3: $b_1 \leftarrow \emptyset$
4: pending $\leftarrow$ LEAVES(TREE(L levels, degree N))
5: workers $\leftarrow \{1, 2, \ldots, K\}$, finished $\leftarrow \emptyset$
6: **while** #finished $< N^L$ **do**
7: **while** #workers > 0 and #pending > 0 **do**
8: $i \leftarrow$ SELECT(pending), pending $\leftarrow$ pending $\setminus \{i\}$
9: $w \leftarrow$ min(workers), workers $\leftarrow$ workers $\setminus \{w\}$
10: SELECTPOLICY(i)
11: job_i = UNIQUEID()
12: SEND(target=w, (i=i, id=job_i, policy=p_i, seed=seed_i))
13: (i, w, id, result) $\leftarrow$ RECEIVE() ▷ Blocking call
14: workers $\leftarrow$ workers $\cup \{w\}$
15: **if** job_i = id **then** ▷ Job was not discarded
16: $r_i \leftarrow$ result
17: finished $\leftarrow$ finished $\cup \{i\}$
18: **while** RESULT(i)$> b_i$ **do**
19: $b_i \leftarrow$RESULT(i)
20: **for** each sibling $j > k$ **do** DISCARD(j)
21: **if** $i = 1$ **then** break ▷ Root node
22: **else** i = PARENT(i)
23: **return** b_1
24: **function** SELECTPOLICY(i)
25: **if** p_i is not set **then**
26: **if** i is first in node **then**
27: $j \leftarrow$PARENT(i)
28: SETPOLICY(j)
29: $b_i \leftarrow \emptyset$
30: $p_i \leftarrow p_j$ ▷ Copy parent's policy
31: **else**
32: SETPOLICY($i - 1$)
33: $b_i \leftarrow b_{i-1}$
34: $p_i \leftarrow$ADAPT(p_j, b_{i-1}) ▷ Adapt sibling's policy
35: **function** DISCARD(j)
36: delete p_j, b_j
37: **if** j is leaf **then**
38: delete r_j, job_j
39: finished $\leftarrow$ finished $\setminus \{j\}$
40: pending $\leftarrow$ pending $\cup \{j\}$
41: **else**
42: **for** each child k **do** DISCARD(k)
43: **function** RESULT(i)
44: **return** $\max\{r_j \colon j$ is consistent leaf below $i\}$