Monte Carlo Inverse Folding

Tristan Cazenave and Thomas Fournier

LAMSADE, Université Paris-Dauphine, PSL, CNRS, France
Tristan.Cazenave@dauphine.psl.eu
thomas.fournier@dauphine.eu

Abstract. The RNA Inverse Folding problem comes from computational biology. The goal is to find a molecule that has a given folding. It is important for scientific fields such as bioengineering, pharmaceutical research, biochemistry, synthetic biology and RNA nanostructures. Nested Monte Carlo Search has given excellent results for this problem. We propose to adapt and evaluate different Monte Carlo Search algorithms for the RNA Inverse Folding problem.

1 Introduction

Monte Carlo Tree Search (MCTS) [16,24] originated in the field of computer Go and has been applied to many other problems [4].

Nested Monte Carlo Search (NMCS) [5] uses multiple levels of search, memorizing the best sequence of each level. It has been applied to many single-player games and optimization problems [31,25,23,2,6,29,30,3] and also to two-player games [7].

Nested Rollout Policy Adaptation (NRPA) [32] also uses multiple levels of search, memorizing the best sequences. It additionally learns a playout policy using the best sequences. It has also been applied to many problems [13,18,20,21,22,19,17,11] and games [8].

The RNA design problem also named the RNA Inverse Folding problem is computationally hard [1]. This problem is important for scientific fields such as bioengineering, pharmaceutical research, biochemistry, synthetic biology and RNA nanostructures [28]. NMCS has been successfully applied to the RNA Inverse Folding problem with the NEMO program [28]. As a follow-up to NEMO, we propose to investigate different Monte Carlo Search algorithms for this problem.

The paper is organized as follows. The second section describes the Inverse Folding problem, the NEMO program by Fernando Portela [28] and the domain knowledge used in NEMO. The third section describes the Monte Carlo Search algorithms we have used for solving Inverse Folding problems of the Eterna100 benchmark. We present a new algorithm performing well for this problem, the Generalized Nested Rollout Policy Adaptation (GNRPA) algorithm with restarts. The fourth sections details experimental results.

2   Inverse Folding

2.1   Presentation of the RNA inverse folding problem

An RNA strand is a molecule composed of a sequence of nucleotides. This strand folds back on itself to form what is called its secondary structure (See Figure 1). It is possible to find in a polynomial time the folded structure of a given sequence. However, the opposite, which is the RNA inverse folding problem, is much harder and is supposed to be NP-complete. This problem still resists algorithmic approaches that still fail to match the performance of human experts. This is partly due to the chaotic changes that can occur in the secondary structure because of a small change in the sequence (See difference between Figure 3 and 2.b).

These performances are evaluated on the Eterna100 benchmark which contains 100 RNA secondary structure puzzles of varying degrees of difficulty. A puzzle consists of a given structure under the dot-bracket notation. This notation defines a structure as a sequence of parentheses and points each representing a base. The matching parentheses symbolize the paired bases and the dots the unpaired ones. The puzzle is solved when a sequence of the four nucleotides A,U,G and C, folding according to the target structure, is found. In some puzzles, the value of certain bases is imposed.

Where human experts have managed to solve the 100 problems of the benchmark, no program has so far achieved such a score. The best score so far is 95/100 by NEMO, NEsted MOnte Carlo RNA Puzzle Solver [28].

2.2   NEMO

NEMO works by performing several iterations of NMCS-B, a slightly modified version of NMCS. Between each iteration of the NMCS-B, NEMO retains part of the best current solution. It identifies a part of the sequence on which to perform mutations using stochastic heuristics and restarts the NMCS-B on it. We model a candidate by a sequence. The bases assigned are represented by the corresponding letter (A/U/G/C) and the others by the letter N. This is how the NMCS-B identifies the bases on which it must work.

NEMO uses a level 1 NMCS for its NMCS-B. At level 1, the NMCS, in each state of the problem, will perform a certain number of playouts for each possible move. It then plays the move that led to the best playout and moves to the next state until it reaches a final state. In the context of NEMO, each state corresponds to a sequence, initially the candidate sequence. Each move consists in taking the first N in the sequence and assigning a value to it, working on paired bases first. When the base is paired in the target structure, it will assign a value to both bases of the pair simultaneously. Indeed only the three combinations AU, GC, and GU can be paired, so it is more convenient to consider them at the same time. To perform playouts, NEMO is also using heuristics with biased weights depending on the location in the target structure. In addition, unlike the classic NMCS, the NMCS-B retains the best playout achieved so far throughout the execution. A final state is found when the sequence is fully completed. The playouts are evaluated according to the function :
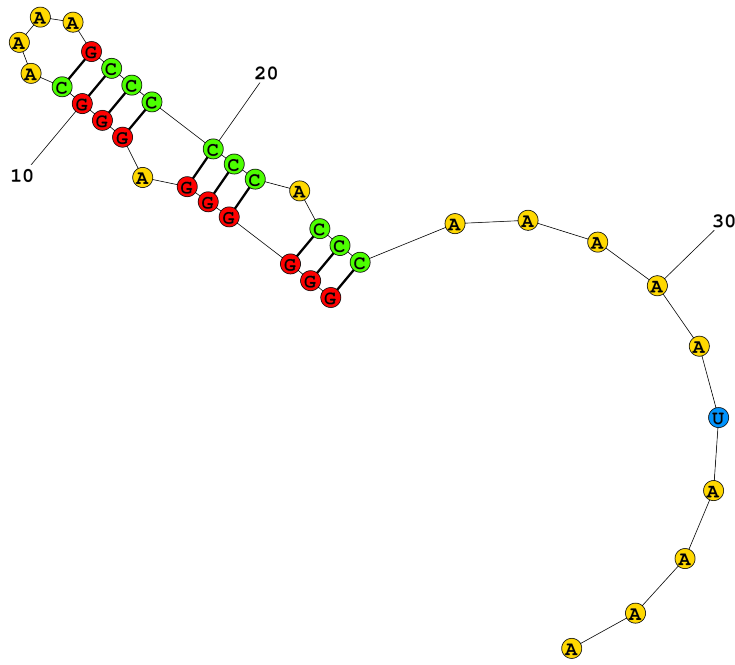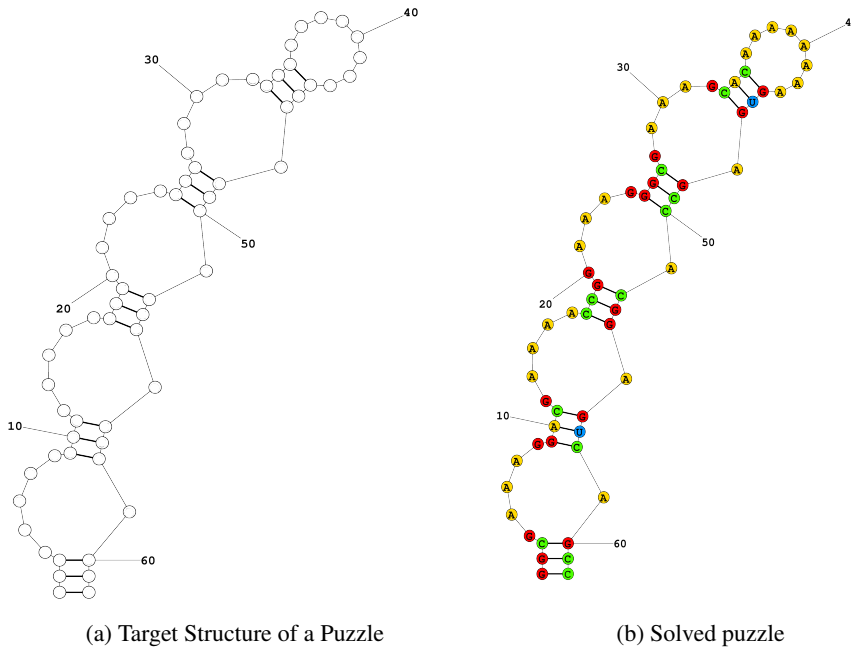
Fig. 1: Secondary Structure of a RNA strand

(a) Target Structure of a Puzzle   (b) Solved puzzle

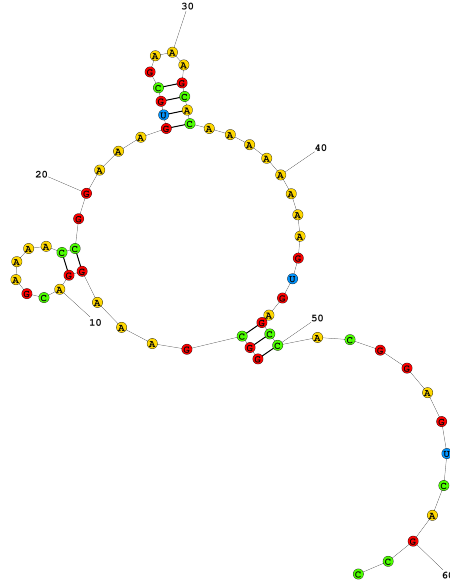Fig. 2: Exemple of an Eterna100 puzzle

Fig. 3: Sequence of figure 2.b with base 25 replaced by U

$$score = \begin{cases} \frac{K}{1+\Delta G} & \text{if } K > 0 \\ K(1 + \Delta G) & \text{else} \end{cases}$$

with $K = 1 - \frac{BPD}{2*NumTargetPairs}$

Where BPD is the number of different pairs between the secondary structure of the sequence and the target structure.

$NumTargetPairs$ is the number of pairs in the target structure.

ΔG is the difference between the Minimum Free Energy of the secondary structure and the free energy that the sequence would have in the target structure.

The objective is to maximize the score function until a value of 1 is obtained, meaning that the problem is solved. Indeed, we use a model based on Minimum Free Energy (MFE) predictions. For each pair composed of a sequence and a secondary structure of the same size we can compute the Gibbs-Free Energy. The predicted secondary structure of a sequence is the one minimizing the Gibbs-Free Energy among all structures of this length. Thus we try to minimize the positive values ΔG and BPD until they are zero, i.e. score=1. Due to the large number of possible matches, BPD is for most sequences very high especially for long sequences and K is therefore negative. However, interesting sequences whose secondary structure is close to the target structure thus have scores between 0 and 1.

2.3 Domain Knowledge and Heuristics

The heuristics used for the sampling of the NMCS-B are based on domain knowledge and personal experience and are chosen without computational optimization. These are crucial as they increase the average number of puzzles solved in a run from 88 to 92.1 and is on average twice as fast.

The paired bases are generally chosen according to the same rule. With the exception of adjacent stacks in multi-loops, the closing pairs of the left-most and rightmost stacks are chosen with the weights given in the following table. The notion of right and left in this case is defined from the point of view of the inside of the loop.

Paired Bases	GC/CG	AU/UA	GU/UG
General Case	60%	33%	7%
Left-Most in Junction	82%	11%	7%
Right-Most in Junction	37%	56%	7%

Various rules are applied for unpaired bases. The weights used to choose between A/U/G/C in the general case are 93%, 1%, 5% and 1%. Mismatches are treated differently depending on the case.

Since NEMO first assigns a value to the paired bases, the weights for the bases with a paired mismatch depend on the value of this mismatch.

Mismatch with a Paired Base	A	U	G	C
Mismatch is a paired A	63%	0%	25%	12%
Mismatch is a paired U	0%	55%	9%	36%
Mismatch is a paired G	25%	12%	63%	0%
Mismatch is a paired C	55%	36%	0%	9%

Furthermore, in internal loops, the weights also depend on the mismatch value if it has already been assigned, otherwise a more general rule applies.

Mismatch in Internal Loops	A	U	G	C
Mismatch is not assigned	18%	4%	74%	4%
Mismatch is A	44%	0%	44%	12%
Mismatch is U	0%	67%	11%	22%
Mismatch is G	67%	11%	22%	0%
Mismatch is C	66%	17%	0%	17%

Finally, the mismatch in junctions and external loops are drawn according to the distribution 97%, 1%, 1% and 1%.

Much stronger and more deterministic rules are applied with high probabilities (more than 80%) in specific cases, especially for triloops and internal loops. For both 1-1 and 2-2 internal loops for instance, only one mismatch pair is possible, and there are only three possibilities in the general case. This is part of a process of reproducing a "boosting" strategy. Depending on the type of loop, certain combinations of nucleotides at specific locations called "boosting points", especially terminal mismatches, can be used to reduce the energy of the structure. However, the most difficult puzzles may require

less conventional solutions, hence the need not to apply these rules 100% of the time. Therefore, in the use we will make of this heuristic we will not apply these last rules and when we mention the weights of the NEMO heuristic we refer to the previously mentioned values.

In addition, between iterations of NMCS-B, if it hasn't solved the problem NEMO keeps part of the best current solution to restart the algorithm on. The set of bases that are not kept contains those that do not fold correctly, their neighborhood and randomly selected bases. This principle, i.e. keeping part of the best current solution, has not been applied to the presented algorithms.

3   Monte Carlo Search

3.1   Presentation of the NRPA algorithm

The Nested Rollout Policy Adaptation (NRPA) algorithm is a Monte Carlo Tree Search based algorithm with adaptive rollout policy during execution. It is a recursive algorithm. At level 0 it generates a playout according to the current policy. At level n, it calls for a given number of iterations the n-1 level of the algorithm, adapting the policy each time with the best solution so far. NRPA is given in algorithms 1, 2 and 3. The policy consists of a set of weights, one for each possible coding value. The adaptation is made through gradient ascent. Given a rollout to adapt the policy on, we increase the weights of the moves chosen during this playout in proportion to their values and decreasing that of the other possible moves in each state. The NRPA applies this gradient ascent on the best current solution of the level after each iteration.

Algorithm 1 The playout algorithm

1: playout ($state$, $policy$)
2:      $sequence \leftarrow []$
3:      **while** true **do**
4:          **if** $state$ is terminal **then**
5:              **return** (score ($state$), $sequence$)
6:          **end if**
7:          $z \leftarrow 0.0$
8:          **for** m in possible moves for $state$ **do**
9:              $z \leftarrow z + \exp (policy\ [\text{code}(m)])$
10:         **end for**
11:         choose a $move$ with probability $\frac{exp(policy[code(move)])}{z}$
12:         $state \leftarrow$ play ($state$, $move$)
13:         $sequence \leftarrow sequence + move$
14:     **end while**

Algorithm 2 The Adapt algorithm

1: Adapt ($policy$, $sequence$)
2:      $polp \leftarrow policy$
3:      $state \leftarrow root$
4:      **for** $move$ in $sequence$ **do**
5:          $polp$ [code($move$)] $\leftarrow polp$ [code($move$)] $+ \alpha$
6:          $z \leftarrow 0.0$
7:          **for** m in possible moves for $state$ **do**
8:              $z \leftarrow z +$ exp ($policy$ [code(m)])
9:          **end for**
10:         **for** m in possible moves for $state$ **do**
11:             $polp$ [code(m)] $\leftarrow polp$ [code(m)] $- \alpha * \frac{exp(policy[code(m)])}{z}$
12:         **end for**
13:         $state \leftarrow$ play ($state$, $move$)
14:     **end for**
15:     $policy \leftarrow polp$

Algorithm 3 The NRPA algorithm.

1: NRPA ($level$, $policy$)
2:      **if** level == 0 **then**
3:          **return** playout (root, $policy$)
4:      **else**
5:          $bestScore \leftarrow -\infty$
6:          **for** N iterations **do**
7:              (result,new) $\leftarrow$ NRPA($level - 1$, $policy$)
8:              **if** result $\geq$ bestScore **then**
9:                  bestScore $\leftarrow$ result
10:                 seq $\leftarrow$ new
11:             **end if**
12:             policy $\leftarrow$ Adapt (policy, seq)
13:         **end for**
14:         **return** (bestScore, seq)
15:     **end if**

3.2   Application of NRPA to Inverse Folding

As part of the Inverse Folding problem, one solution consists in a chain of bases. A playout is made by running through the targeted structure of the chain, each move consists in assigning a value to the missing links. We distinguish between two cases, the unpaired bases in the target structure have four possible moves, one for each nucleic base (A/U/G/C). The paired bases have 6 possible moves, one for each possible ordered combination with their pair (GC/CG/AU/...). Each move is therefore defined by its position in the chain and whether it is a pair or not. Thus, there is a fixed number of moves, which are always ordered in the same way. Solutions are evaluated with the same score function as in the NEMO algorithm which is a combination of the fitness of the chain with the target structure and the difference between the target structure and the folded chain structure.

3.3   Generalized NRPA

Let w_{ib} be the weight associated to move b at index i in the sequence. In NRPA the probability of choosing move b at index i is:

$$p_{ib} = \frac{e^{w_{ib}}}{\Sigma_k e^{w_{ik}}}$$

We propose to try Generalized NRPA (GNRPA) [9] for Inverse Folding and to replace it with:

$$p_{ib} = \frac{e^{w_{ib}+\beta_{ib}}}{\Sigma_k e^{w_{ik}+\beta_{ik}}}$$

where we use for β_{ib} the logarithm of the probabilities used in NEMO.

3.4   Stabilized GNRPA

Stabilized NRPA [12] is a simple improvement of NRPA. The principle is to play P playouts at level 1 before each call to the adapt function. The number of calls to the adapt function as level 1 is still N, the number of iteration of upper levels. So at level 1, $P \times N$ playouts are performed. The same principle is used for stabilized GNRPA.

3.5   Beam GNRPA

Beam NRPA has already been applied successfully to the TSPTW and to Morpion Solitaire [14]. The best results were obtained using a beam at level 1. Similarly to Stabilized GNRPA, at level 1, $B \times N$ playouts are performed for a beam of size B. However the algorithm is different from Stabilized GNRPA since it memorizes B best sequences and B policies and plays the B playouts with different policies.

As Stabilized GNRPA it is embarrassingly parallel at level 1 and can be very efficient on a parallel machine.

When using Beam NRPA/GNRPA it can be beneficial to ensure the diversity of the beam [17]. The diversity criterion we have used is to only keep in the beam sequences that have different scores. It is simple and efficient as it ensures diversity while keeping enough sequences.

3.6 Coding Moves

Moves in the policy are represented by their encoding values, hence their importance. The natural way to code moves for Inverse Folding is to use the index of the base or of the pair of base in the string (m.index) and the index of the base in the list of bases or of the pair of bases in the list of pairs of bases (m.number). The formula is then:

$$code(m) = m.index + M \times m.number$$

For example if the move m is to put the fourth base at index 10 the code is $10 + 2000 * 4 = 8010$, provided strings always have less than 2000 characters and therefore $M = 2000$.

It may be interesting to include the previously chosen bases in the code of a move, for example if a base has meaning only if following another base. We call the history of a code the number of bases in the history included in the code. The previous formula holds for a code history of 0. The code for a code history of 1 is:

$$code(m) = m.index + M \times m.number + M \times 6 \times previousMove.number$$

Six is the maximum value for a move number, the maximum number of legal moves is 6. The code for a code history of 2 includes the two previous moves in the code.

3.7 Start Learning

In order to wait for better sequences before learning it is possible to delay learning only after a given number of sequences have been found [7]. For this we call the adapt method only after a certain number of iterations as in line 19 of algorithm 4.

3.8 Zobrist Hashing

Each state is associated to a different hash code. This is done with Zobrist Hashing. Each move at each index is associated to a random number. The hash code of a state is the XOR of all the random numbers corresponding to the moves that have been played to reach this state. Zobrist hashing is used in games to build a transposition table. We will use it in the UCT variants. Another use of Zobrist Hashing is to detect playouts that have already been evaluated. As most of the time in Inverse Folding is spent scoring the playouts, it is advisable to avoid reevaluating an already evaluated playouts. This is done with a score hash table that contains the hash of the terminal states already encountered associated to their computed scores. If a terminal state is met again the score need not be recomputed it can just be sent back from the table. There are variations on the number of playouts already evaluated according to the sequence, some sequences have very few while others have a lot.

3.9   Restarts

There can be large variations on the solving times of some problems. Sometimes the search algorithm takes a wrong direction and stay stuck on a suboptimal sequence without making any progress. A way to deal with this behavior is to periodically stop and restart the search. For some difficult problems however a long search is required to find the solution. There are multiple ways to use restarts. The algorithm can double the search time at each restart for example. We call this method iterative doubling. We have observed that a level 2 search is able to solve many problems, a restart strategy can also be to repeatedly call GNRPA at level 2 until thinking time is elapsed. Another way to deal with search being stuck is to stop a level when the best sequence has not changed for a given number of recursive calls.

It is difficult to set a static restart strategy for all problems. Long sequences are much more difficult than short ones and the progress on long sequences is slower. In order to cope with this property we use a restart threshold. It is set to the length of the sequence divided by 5. When using this restart strategy there is no limit on the length of a level.

Algorithm 4 gives the GNRPA algorithm with restarts.

Algorithm 4 The GNRPA algorithm with restarts.

```
 1: GNRPA (level, policy)
 2:    if level == 0 then
 3:        return  playout (root, policy)
 4:    else
 5:        bestScore ← −∞
 6:        last ← 0
 7:        for i in range(∞) do
 8:            (result,new) ← GNRPA(level − 1, policy)
 9:            if result ≥ bestScore then
10:                if result > bestScore then
11:                    last ← i
12:                end if
13:                bestScore ← result
14:                seq ← new
15:            end if
16:            if i − last ≥ limitRestart then
17:                return  (bestScore, seq)
18:            end if
19:            if i ≥ startLearning then
20:                policy ← Adapt (policy, seq)
21:            end if
22:        end for
23:    end if
```

3.10   Parallelization

The multiple playouts of stabilized NRPA and the loop over the elements of the beam are embarrassingly parallel. We simply parallelized with OpenMP a common loop including the beam and the stabilized playouts at level 1. If we have 4 stabilized playouts and a beam of 8, the 32 resulting playouts are played in parallel. This kind of parallelization is a kind of leaf parallelization [10,15].

We also experiment with root parallelization [10,15]. The principle is to perform multiple independent GNRPA in parallel and to stop as soon as one has found a solution or when the allocated time is elapsed.

Leaf parallelization is more difficult to scale than root parallelization. For the same wall clock time leaf parallelization runs more iterations for a single policy than root parallelization which optimizes many more different policies but with less iterations. On the other hand root parallelization scales very well and has a built-in restart strategy. Root parallelization works well for problems that converge relatively rapidly on a suboptimal solution as they benefit from restarts, while leaf parallelization works better for problems that converge slowly but steadily towards the best solutions.

Table 1: Number of problems solved, out of 100 problems, using different parameters.

Level	α	N	β_{ib}	P	Beam	H	Solved
1	1.0	100	no	1	1.1.1	0	3
1	1.0	100	yes	1	1.1.1	0	30
1	1.0	100	yes	1	1.1.1	1	32
1	1.0	100	yes	1	4.1.1	1	42
1	1.0	100	yes	1	8.1.1	1	53
1	1.0	100	yes	1	16.1.1	1	54
1	1.0	100	yes	4	8.1.1	1	69
1	1.0	100	yes	4	8.1.1	2	69
2	1.0	100	no	1	1.1.1	0	49
2	1.0	100	yes	1	1.1.1	0	73
2	1.0	100	yes	1	1.1.1	1	75
2	1.0	100	yes	2	1.1.1	0	73
2	1.0	100	yes	3	1.1.1	0	74
2	1.0	100	yes	4	1.1.1	0	80
2	1.0	100	yes	5	1.1.1	0	77
2	1.0	100	yes	6	1.1.1	0	75
2	1.0	100	yes	7	1.1.1	0	80
2	1.0	100	yes	8	1.1.1	0	79
2	1.0	100	yes	9	1.1.1	0	81
2	1.0	100	yes	10	1.1.1	0	80
2	1.0	100	yes	4	8.1.1	1	85
3	1.0	100	yes	1	1.1.1	0	85

Table 2: Parallelization efficiency.

Algorithm	1	2	4	6	8	12
GNRPA(level=1,N=100,P=4,Beam=8)	11.916	6.889	4.526	3.657	3.169	3.359

Table 3: Number of problems solved by GNRPA using different parameters and a fixed time limit.

β_{ib}	P	Beam	//	R	N	Start	H	1m	2m	4m	8m	16m	32m	64m
no	1.1	1.1	n	n	100.100	0.0	0	30	33	41	50	61	67	69
yes	1.1	1.1	n	n	100.100	0.0	0	58	64	68	72	74	75	79
yes	4.1	4.1	n	n	100.100	0.0	0	71	75	78	79	81	83	84
yes	4.1	8.1	n	n	100.100	0.0	0	75	75	79	80	81	83	84
yes	4.1	8.1	n	n	100.100	0.0	1	75	78	80	82	83	85	87
yes	4.1	8.1	n	n	100.100	4.4	1	76	80	81	82	84	85	87
yes	4.1	8.1	y	n	100.100	4.4	1	78	84	83	86	87	87	88
yes	4.1	8.1	y	3	$\infty.\infty$	4.4	1	80	84	85	85	88	89	92

Table 4: Number of problems solved by different algorithms and a fixed time limit.

Algorithm	1m	2m	4m	8m	16m	32m	64m
UCT(0.4)	47	49	50	54	56	57	62
Nested(1)	59	62	64	65	67	71	73
Nested(2)	55	59	61	66	66	71	73
Diversity GNRPA(5.1)	72	73	73	77	78	78	80

Table 5: Number of problems solved with different options.

Correction	Order	1m	2m	4m	8m	16m
no	no	84	87	85	85	87
yes	no	80	86	85	86	87
no	yes	81	84	86	85	87
yes	yes	80	82	84	88	87

Table 6: Number of problems solved with root parallel GNRPA.

Process	1m	2m	4m	8m	16m	32m	64m
20	82	84	85	86	87	88	89

4   Experimental Results

We now detail experiments with the different Monte Carlo Search algorithms on the Eterna100 benchmark.

Table 1 gives the number of problems solved with different parameters for the GNRPA algorithm. We can see that at level 1 using GNRPA instead of NRPA enables to solve 30 problems instead of 3. Similarly at level 2 it solves 73 problems instead of 49. Using Stabilized GNRPA with P=4 and Beam GNRPA with a beam of 8 at level 1 also improves quite much the number of problems solved at levels 1 and 2. P and H take several values for the number of threads and beams, respectively, used at each level. Interestingly for level 2 it is 32 times slower that a regular level 2 search and solves 85 problems, when a search at level 3 is 100 times slower and still solves 85 problems.

Table 2 gives the median time over 3 runs of a level 1 search with different numbers of threads on problem 64 which is difficult. The algorithm is GNRPA with $P = 4$, a beam of 8 and $N = 100$. We can see that using 8 threads gives the best results. We optimized memory in order to avoid cache misses but we were not able to have better results with more threads.

Table 3 gives the number of problems solved within a fixed time limit for different algorithms. The time limits range from 1 minute to 64 minutes per problem. The parallel algorithm runs on a multicore machine. The results of the parallel program are given in the last line of table 3. The times used to stop the parallel program are the wall clock times. The last line is leaf parallel GNRPA with restarts and gives the best results within 1 hour of wall clock time, solving 92 of the 100 problems in one run. Start is 4 meaning that it starts learning after 4 playouts. H is 1 meaning the code include the previous move, R is 3 meaning that the restart threshold is set to the length of the string divided by 3.

The problems solved by different runs of 1 hour we made are not always the same. Some hard problems are solved only in some runs. So the limit of 92 solved problems is not the limit of the algorithm. The problems that were never solved in 1 hour are problems 100, 99, 97, 91, 90, 78. With a two hours limit, problem 90 is solved thus reaching 95 solved problems, the same number of solved problems as NEMO. However,

it is interesting to note that GNRPA solves problem 94 while NEMO solves problem 91.

For the sake of completeness we also tested other popular Monte Carlo Search algorithms. The results are given in table 4 for UCT [24], Nested Monte Carlo Search [5,28] and Diversity NRPA [17]. The UCT constant is set to 0.4, NMCS is tested for repeated calls to level 1 and level 2. Diversity GNRPA [17] is called with a set of 5 sequences at level 1 and 1 sequence at level 2. The improved GNRPA algorithm gives better results than these algorithms.

Table 5 gives the results with time of the best parallel algorithm using different options. The Correction option is to fix a discrepancy between the NEMO paper and the NEMO code in the heuristic. The Order option is to order moves such as NEMO or to use the order of the string. Given the results in the table the two options do not seem to matter much.

Table 6 gives the number of problems solved with time using the root parallel GN-RPA algorithm with 20 process. The results are slightly worse than when using leaf parallelization with 8 threads. It is due to problems that converge slowly and do not benefit from restarts and where leaf parallelization enables to improve during much longer than root parallelization the best policy. Given enough resources the best algorithm might be the combination of root and leaf parallelization as in [27]. The parallelization of NRPA proposed by Nagorko [26] is also appealing.

5   Conclusion

We experimented with various Monte Carlo Search algorithms for the Inverse Folding problem. We have used very limited domain knowledge, essentially using a small part of the NEMO heuristics for the bias. By applying general Monte Carlo Search heuristics we were able to solve as many problems as NEMO in comparable times.

Acknowledgment

References

1. Édouard Bonnet, Rzążewski, P., Sikora, F.: Designing RNA secondary structures is hard. Journal of Computational Biology **27**(3) (2020)
2. Bouzy, B.: Monte-carlo fork search for cooperative path-finding. In: Computer Games - Workshop on Computer Games, CGW 2013, Held in Conjunction with the 23rd International Conference on Artificial Intelligence, IJCAI 2013, Beijing, China, August 3, 2013, Revised Selected Papers. pp. 1–15 (2013)

3. Bouzy, B.: Burnt pancake problem: New lower bounds on the diameter and new experimental optimality ratios. In: Proceedings of the Ninth Annual Symposium on Combinatorial Search, SOCS 2016, Tarrytown, NY, USA, July 6-8, 2016. pp. 119–120 (2016)

4. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., Colton, S.: A survey of Monte Carlo tree search methods. IEEE Transactions on Computational Intelligence and AI in Games **4**(1), 1–43 (Mar 2012). https://doi.org/10.1109/TCIAIG.2012.2186810

5. Cazenave, T.: Nested Monte-Carlo Search. In: Boutilier, C. (ed.) IJCAI. pp. 456–461 (2009)

6. Cazenave, T.: Monte-carlo expression discovery. International Journal on Artificial Intelligence Tools **22**(1) (2013)

7. Cazenave, T.: Nested rollout policy adaptation with selective policies. In: Computer Games, CGW at IJCAI 2016, pp. 44–56. Springer (2016)

8. Cazenave, T.: Playout policy adaptation with move features. Theoretical Computer Science **644**, 43–52 (2016)

9. Cazenave, T.: Generalized nested rollout policy adaptation. In: Monte Search at IJCAI (2020)

10. Cazenave, T., Jouandeau, N.: On the Parallelization of UCT. In: Computer Games Workshop. Amsterdam, Netherlands (Jun 2007), https://hal.archives-ouvertes.fr/hal-02310186

11. Cazenave, T., Lucas, J.Y., Kim, H., Triboulet, T.: Monte carlo vehicle routing. In: ATT at ECAI (2020)

12. Cazenave, T., Sevestre, J.B., Toulemont, M.: Stabilized nested rollout policy adaptation. In: Monte Search at IJCAI (2020)

13. Cazenave, T., Teytaud, F.: Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows. In: Learning and Intelligent Optimization - 6th International Conference, LION 6, Paris, France, January 16-20, 2012, Revised Selected Papers. pp. 42–54 (2012)

14. Cazenave, T., Teytaud, F.: Beam nested rollout policy adaptation. In: Computer Games Workshop, ECAI 2012. pp. 1–12 (2012)

15. Chaslot, G.M.B., Winands, M.H., van Den Herik, H.J.: Parallel monte-carlo tree search. In: International Conference on Computers and Games. pp. 60–71. Springer (2008)

16. Coulom, R.: Efficient selectivity and backup operators in Monte-Carlo tree search. In: van den Herik, H.J., Ciancarini, P., Donkers, H.H.L.M. (eds.) Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers. Lecture Notes in Computer Science, vol. 4630, pp. 72–83. Springer (2006)

17. Edelkamp, S., Cazenave, T.: Improved diversity in nested rollout policy adaptation. In: KI 2016: Advances in Artificial Intelligence - 39th Annual German Conference on AI, Klagenfurt, Austria, September 26-30, 2016, Proceedings. pp. 43–55 (2016)

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

19. Edelkamp, S., Gath, M., Greulich, C., Humann, M., Herzog, O., Lawo, M.: Monte-carlo tree search for logistics. In: Commercial Transport, pp. 427–440. Springer International Publishing (2016)

20. Edelkamp, S., Gath, M., Rohde, M.: Monte-carlo tree search for 3d packing with object orientation. In: KI 2014: Advances in Artificial Intelligence, pp. 285–296. Springer International Publishing (2014)

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

22. Edelkamp, S., Tang, Z.: Monte-carlo tree search for the multiple sequence alignment problem. In: Eighth Annual Symposium on Combinatorial Search (2015)

23. Kinny, D.: A new approach to the snake-in-the-box problem. In: ECAI. vol. 242, pp. 462–467 (2012)
24. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: 17th European Conference on Machine Learning (ECML'06). LNCS, vol. 4212, pp. 282–293. Springer (2006)
25. Méhat, J., Cazenave, T.: Combining UCT and Nested Monte Carlo Search for single-player general game playing. IEEE Transactions on Computational Intelligence and AI in Games **2**(4), 271–277 (2010)
26. Nagorko, A.: Parallel nested rollout policy adaptation. In: IEEE Conference on Games (2019)
27. Negrevergne, B., Cazenave, T.: Distributed nested rollout policy for samegame. In: Workshop on Computer Games. pp. 108–120. Springer (2017)
28. Portela, F.: An unexpectedly effective monte carlo technique for the rna inverse folding problem. BioRxiv p. 345587 (2018)
29. Poulding, S.M., Feldt, R.: Generating structured test data with specific properties using nested monte-carlo search. In: Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014. pp. 1279–1286 (2014)
30. Poulding, S.M., Feldt, R.: Heuristic model checking using a monte-carlo tree search algorithm. In: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015. pp. 1359–1366 (2015)
31. Rimmel, A., Teytaud, F., Cazenave, T.: Optimization of the Nested Monte-Carlo algorithm on the traveling salesman problem with time windows. In: Applications of Evolutionary Computation - EvoApplications 2011: EvoCOMNET, EvoFIN, EvoHOT, EvoMUSART, EvoSTIM, and EvoTRANSLOG, Torino, Italy, April 27-29, 2011, Proceedings, Part II. Lecture Notes in Computer Science, vol. 6625, pp. 501–510. Springer (2011)
32. Rosin, C.D.: Nested rollout policy adaptation for Monte Carlo Tree Search. In: IJCAI. pp. 649–654 (2011)