Learning a Prior for Monte Carlo Search by Replaying Solutions to Combinatorial Problems

Tristan Cazenave

LAMSADE, Université Paris Dauphine - PSL, CNRS, Paris, France

Abstract. Monte Carlo Search gives excellent results in multiple difficult combinatorial problems. Using a prior to perform non uniform playouts during the search improves a lot the results compared to uniform playouts. Handmade heuristics tailored to the combinatorial problem are often used as priors. We propose a method to automatically compute a prior. It uses statistics on solved problems. It is a simple and general method that incurs no computational cost at playout time and that brings large performance gains. The method is applied to three difficult combinatorial problems: Latin Square Completion, Kakuro, and Inverse RNA Folding.

Keywords: Monte Carlo Tree Search · Combinatorial Problems · Learning Search Heuristics · Latin Square Completion · Kakuro · Inverse RNA Folding.

1 Introduction

Monte Carlo Tree Search (MCTS) has been successfully applied to many games and problems [4]. Combined with Deep Reinforcement Learning it has superhuman performances in two player complete information games such as Go and Chess [32].

Nested Monte Carlo Search (NMCS) [6] is an algorithm that works well for puzzles and combinatorial problems. It biases its playouts using lower level playouts. At level zero NMCS adopts a uniform random playout policy. Learning of playout strategies combined with NMCS has given good results on combinatorial problems [28]. Other applications of NMCS include Single Player General Game Playing [24], Cooperative Pathfinding [2], Software testing [26], heuristic Model-Checking [27], the Pancake problem [3], Games [10], the Inverse RNA Folding problem [25] and retrosynthesis [30].

Online learning of a playout policy in the context of nested searches has been further developed for puzzles and combinatorial problems with Nested Rollout Policy Adaptation (NRPA) [29]. NRPA has found new world records in Morpion Solitaire and crosswords puzzles. NRPA has been applied to multiple problems: the Traveling Salesman Problem with Time Windows (TSPTW) [11,13], 3D Packing with Object Orientation [15], the physical traveling salesman problem [16], the Multiple Sequence Alignment problem [17] or Logistics [14]. The principle of NRPA is to adapt the playout policy so as to reinforce the best sequence of moves found so far at each level.

The use of Gibbs sampling in Monte Carlo Tree Search dates back to the general game player Cadia Player and its MAST playout policy [19].

Monte Carlo Search for combinatorial problems can be much improved using a prior. A prior is a heuristic that is used in playouts to sample in a non uniform way. It favors some moves in the playout according to the heuristic. The use of a bias or the initialization of the weights to produce an initial non uniform policy have been used for multiple difficult problems: the Traveling Salesman Problem with Time Windows [28,13] with a distance based heuristic, the Vehicle Routing Problems [14,9] with again a distance based heuristic, the Inverse RNA Folding problem [25] with manually encoded heuristics on pairs of bases, the Pancake problem [3] with manually encoded heuristics, the Virtual Network Embedding problem [18] with a distance based heuristic again. In all these problems the manual prior improves much the performances of Monte Carlo Search.

We propose a method to automatically compute a prior. It uses statistics on solved problems. The method is simple, moreover it does not use computation time during sampling and it is general. It improves much on Monte Carlo Search without a prior for the problems that we tried. It also improves over manually defined priors.

We now give the outline of the paper. The second section describes Monte Carlo Search. The third section explains how to compute the prior. The fourth section gives experimental results for Latin Square Completion (LSC), Kakuro and Inverse RNA Folding.

2 Monte Carlo Search

This section presents the Generalized NRPA (GNRPA) [7] algorithm which is a generalization of the NRPA algorithm to the use of a prior.

The Nested Rollout Policy Adaptation (NRPA) [29] algorithm is an effective combination of NMCS and the online learning of a playout policy. NRPA holds world records for Morpion Solitaire and crosswords puzzles. It is different from learning a prior as GNRPA reinforces the policy for the instance whereas learning a prior is done once for all and is used for all instances.

In NRPA/GNRPA each move is associated to a weight stored in an array called the policy. The goal of these two algorithms is to learn these weights using the best sequences of moves found during the search. The weights are used in the softmax function to produce a playout policy that generates good sequences of moves.

NRPA/GNRPA use nested search. In NRPA/GNRPA, each level takes a policy as input and returns a sequence and its associated score. At any level > 0, the algorithm makes numerous recursive calls to the lower level, adapting the policy each time with the best sequence of moves to date. The changes made to the policy do not affect the policy in higher levels. At level 0, NRPA/GNRPA return the sequence obtained by the playout function as well as its associated score.

The playout function sequentially constructs a random solution biased by the weights of the moves until it reaches a terminal state. At each step, the function performs Gibbs sampling, choosing the actions with a probability given by the softmax function.

Let w_m be the weight associated to a move m in the policy. In NRPA, the probability of choosing move m is defined by:

$$p_m = \frac{e^{w_m}}{\sum_k e^{w_k}}$$

where k goes through the set of possible moves, including m.

GNRPA [7] generalizes the way the probability is calculated using a bias β_m. The probability of choosing move m becomes:

$$p_m = \frac{e^{w_m+\beta_m}}{\sum_k e^{w_k+\beta_k}}$$

By taking $\beta_m = \beta_k = 0$, we find the formula for NRPA again wich corresponds to sampling without a prior.

In NRPA it is possible to initialize the weights according to a heuristic relevant to the problem to solve. In GNRPA, the policy initialization is replaced by the bias. It is sometimes more practical to use β_k biases than to initialize the weights as the codes for the moves can be different from the codes of the biases. The method we propose could also be applied without modification to NRPA with initialization of the weights by initializing the weight of move m with β_m the first time the weight is used.

The algorithm to perform playouts in GNRPA is given in algorithm 1. The main GNRPA algorithm is given in algorithm 3. GNRPA calls the adapt algorithm to modify the policy weights so as to reinforce the best sequence of the current level. The policy is passed by reference to the adapt algorithm which is given in algorithm 2.

The principle of the adapt function is to increase the weights of the moves of the best sequence of the level and to decrease the weights of all possible moves by an amount proportional to their probabilities of being played. $\delta_{bm} = 0$ when $b \neq m$ and $\delta_{bm} = 1$ when $b = m$.

```
1:  playout (policy)
2:     state ← root
3:     while true do
4:        if terminal(state) then
5:           # sequence(state) contains the moves played from the root to the state
6:           return (score (state), sequence(state))
7:        end if
8:        z ← 0
9:        for m ∈ possible moves for state do
10:          o[m] ← e^{policy[code(m)]+β_m} # code(m) is an integer representing move m
11:          z ← z + o[m]
12:       end for
13:       choose a move with probability o[move]/z
14:       play (state, move)
15:    end while
```

Algorithm 1: The playout algorithm

```
1:  adapt (policy, sequence)
2:    polp ← policy
3:    state ← root
4:    for b ∈ sequence do
5:      z ← 0
6:      for m ∈ possible moves for state do
7:        o[m] ← e^{policy[code(m)]+β_m}
8:        z ← z + o[m]
9:      end for
10:     for m ∈ possible moves for state do
11:       p_m ← o[m]/z
12:       polp[code(m)] ← polp[code(m)] − α(p_m − δ_bm)
13:     end for
14:     play (state, b)
15:   end for
16:   policy ← polp
```

Algorithm 2: The adapt algorithm

```
1:  GNRPA (level, policy)
2:    if level == 0 then
3:      return playout (policy)
4:    else
5:      bestScore ← −∞
6:      for N iterations do
7:        (score,new) ← GNRPA(level − 1, policy)
8:        if score ≥ bestScore then
9:          bestScore ← score
10:         seq ← new
11:       end if
12:       adapt (policy, seq)
13:     end for
14:     return (bestScore, seq)
15:   end if
```

Algorithm 3: The GNRPA algorithm.

3 Learning a Prior

This section presents the computation of the prior. The principle underlying the prior is to compute the frequency each move has been the move solving a problem. In order to compute it we generate many solved problems associated to their solutions, e.g. the sequence of moves that solves the problem from the starting state. It is usually hard for combinatorial problems to find a solution. However in some problems it is easy to generate problems associated to their solutions. The three problems we experimented with have this property that it is easy to generate problems and their associated solutions.

The principle for learning the prior is to replay the solution and to update the count for each possible move of each possible state of the solution. We also update the count of the moves that are part of the solution. We can then calculate for each move the frequency it has been the solution move, this is the number of times it has been in a solution divided by the number of times it has been a possible move.

1: Replay ($state$, $sequence$)
2: **for** $b \in sequence$ **do**
3: $count[code(b)] \leftarrow count[code(b)] + 1$
4: **for** $m \in$ possible moves for $state$ **do**
5: $nb[code(m)] \leftarrow nb[code(m)] + 1$
6: **end for**
7: play ($state$, b)
8: **end for**

Algorithm 4: The Replay algorithm

Algorithm 4 details how to compute the $count$ and nb arrays given an initial state and the solution to the problem given as a sequence of moves. The nb array memorizes the number of times a move has been possible and the $count$ array memorizes how many times it was part of a solution. The Replay function is called for each solved problem of the training dataset.

We then define the bias β_m as:

$$\beta_m = \tau * log(\frac{count[code(m)]}{nb[code(m)]})$$

where τ is called the temperature of the bias.

The default sampling policy with a prior plays a move m with probability:

$$p_m = \frac{e^{\beta_m}}{\Sigma_k e^{\beta_k}}$$

4 Experimental Results

This section details the computation of the prior for three difficult combinatorial problems: Latin Square Completion, Kakuro and Inverse RNA Folding. It also compares

sampling with the computed prior to sampling without a prior. It also compares NRPA to GNRPA with the computed prior.

Table 1: Number of LSC problems of size 20 in the transition phase solved by different algorithms out of 100 problems. The number of playouts ranges from 1,024 playouts to 131,072 playouts. The temperature of the Dual prior is set to $\tau = 4$ which is the temperature that gave the best results. Sampling with the Dual prior solves more problems than uniform sampling. GNRPA with the Dual prior is better than NRPA and sampling.

Algorithm	1,024	2,048	4,096	8,192	16,384	32,768	65,536	131,072
Sampling	2	5	10	16	26	36	49	61
Sampling Dual prior	12	24	34	48	70	80	89	95
NRPA	8	16	25	35	48	61	70	80
GNRPA Dual prior	26	39	54	67	83	91	95	98

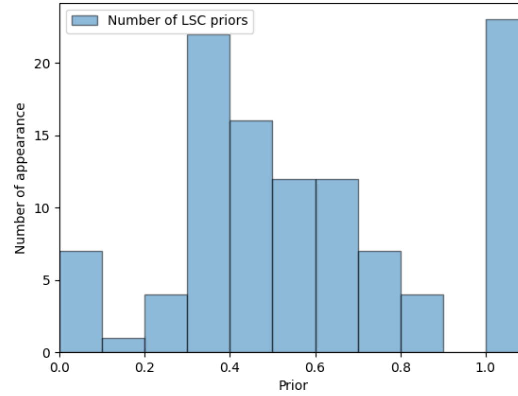

Fig. 1: The distribution of the priors for LSC. The priors associated to codes that have never been seen during replay (e.g. nb [code] = 0) have been removed.

4.1 Latin Square Completion

A Latin Square of order n is a $n \times n$ grid filled with numbers from 1 to n such that the same number does not appear more than once in each row and each column. A partial Latin Square is a Latin Square with some empty cells. The Latin Square Completion problem (LSC) consists in completing a partial Latin Square so as to form a complete Latin Square. Latin Square Completion is a NP-complete problem [12].

The LSC problem has a phase transition. When a grid has a lot of empty cells or only few empty cells, the completion is very easy. When the percentage of empty cells is close to 42% the problem becomes hard. Figure 2 gives the median number of random playouts required to solve LSC problems according to their percentage of empty cells. We can observe the peak in number of random playouts at 42% of empty cells.

It can also be observed in Figure 2 that generating Latin Squares from the empty grid is extremely easy. The first three random playouts usually generate a valid Latin Square. Therefore generating difficult LSC problems and their associated solution is also extremely easy. First generate a valid Latin Square, memorize it as a solution and then randomly remove 42% of the cells so as to have a difficult LSC problem associated to its solution.

Here is an example of a difficult LSC problem of size 20 generated with this method:

```
 3             11 15  4  8 13 17 14  2    12 10  6
13    18 17 14 16  8 19  7              1 11       20    12
20        8    17 12  1 19 10  3  6    16 15          14  4
             3  4           9 14    15 11        5  7 19  1
      15 18     9     3  4  1  5  2       13    12  8 10
      17          19    14              3  7     4 16  6 20
16 13  4 11       10  9 17              7 14     3 15  5
    2     1          15 18     6 16  5 12 17 20 19 14 13 11
 9 20        4     3 11 12  8       17  6    18    19
11 10 20  6 13           5  3  1        9  4 14 18 12     7
    8    19          14 10        7 13 18     5     3 17 15
   12 11  5  6       13       19  4 14 10 20     9  1    18
19 15  9 20 10     5     3    18     4       17    11  2 13
18    14  9 16  5     6 11    13 17     2     3     4
   14  6 15     3  4 18 16  2    11           9  7 17    19
 4  7  5  3    12          19     9 16 20 18    17
    3    10     1    16    12  7       17       20  5 18
    6 12    15  2     7 13        5    19  3        9 11  8
        7 14 17     1 20       15 13          16       12
   18    12     8     5       16  3 11    19        6    17
```

LSC and related problems appear in a variety of practical applications such as scheduling, optical routing, error correcting codes as well as combinatorial design [20].

We model the LSC problem as a Constraint Satisfaction Problem. We use Forward Checking associated to channeling constraints. If a value appears only once in a column or in a row it is directly assigned. If it is not the case, the variable with the smallest number of possible values is chosen and a possible value is randomly assigned according to the policy. A state is terminal if the Latin Square is complete or if one of the variables is not assigned and has an empty domain. In this case the score of a playout is the opposite of the number of remaining variables.

The code associated to a move contains the number of times the value is present in the same column and the number of times it is present in the same row. We call the prior associated to this code the Dual prior. Note that it is a very simple code and that it could probably be refined. The bias for GNRPA using this code is:

$$\beta_m = \tau * log(\frac{count[code(m)]}{nb[code(m)]})$$

Figure 1 gives the distribution of the priors for this code and LSC problems of size 20. The priors were computed using 10,000 solved problems generated randomly in the transition phase. We can observe that the priors have varied values.

Table 1 gives the evolution of the number of problems solved by different algorithms with doubling numbers of playouts. Sampling with the Dual prior is much better than sampling without the prior. GNRPA with the Dual prior is much better than NRPA. The computation time of the Dual prior during the playouts is negligible.

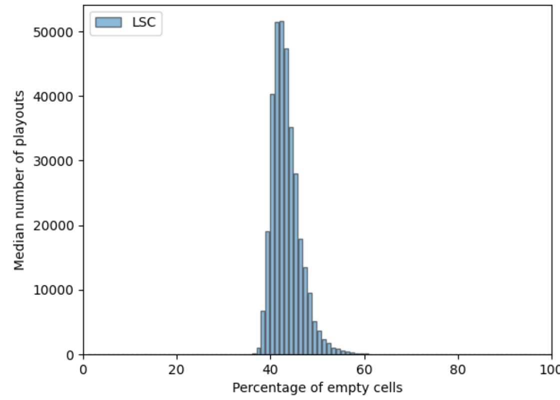

Fig. 2: The median number of random playouts required to solve LSC instances of size 20 with x% of empty cells. The phase transition happens at 42% of empty cells. All further experiments will use Latin Squares of size 20 with 42% of empty cells. The median for each percentage was calculated solving 1,000 problems.

4.2 Kakuro

A Kakuro puzzle is played on a rectangular grid. The objective is to fill numbers into the blank cells, according to the following rules:

- A sum is associated with every horizontal or vertical sequence of blank cells.
- Each horizontal (respectively vertical) sequence has a cell left of (respectively above) its first cell, and that cell contains the sum that is associated with the sequence.
- In each horizontal/vertical sequence of cells, every number may occur at most once.
- The sum of the numbers of a sequence must equal the number that is denoted in the corresponding hint.

Fig. 3: The distribution of the priors for Kakuro. The y-axis gives the number of priors in each range of values. For example there are 15,410 priors that have the value 1.0 and 20,353 priors that have a value between 0.0 and 0.1. The priors associated to codes that have never been seen during replay (e.g. nb [code] = 0) have been removed. We can observe the peak at 0.0 which mainly corresponds to the numbers that are impossible given the row and the column sums. We can also observe the smaller peak at 1.0 which corresponds to the numbers that are forced. Note that apart from these two cases there are many cases where the prior is between 0.0 and 1.0 which does not correspond to a hard constraint.

Table 2: Number of Kakuro problems of size 10, with 11 possible values, solved by different algorithms out of 100 problems and for various numbers of playouts. The temperature of the prior is set to $\tau = 4$. Using the prior usually solves the problem in 1 playout.

Algorithm	1,024	2,048	4,096	8,192	16,384	32,768	65,536	131,072
Sampling	0	0	0	0	0	0	0	0
Sampling Prior	100	100	100	100	100	100	100	100
NRPA	0	0	0	23	35	65	86	98
GNRPA Prior	100	100	100	100	100	100	100	100

Kakuro is hard [31]. The most difficult Kakuro problems are the empty problems with only the sum of the columns and of the rows already given [5].

The generation of a Kakuro problem and its solution is almost as easy as the generation of a LSC problem. First generate a valid square with sampling. A single playout is usually enough. Then calculate the sums for each row and for each column. Then remove all the values and keep the generated valid square as the solution to the problem.

Here is an example of a solved Kakuro problem of size 10 with values ranging from 1 to 11 generated with our method:

```
    65 60 58 62 59 59 62 60 56 55
55   3  5  4  1 10  8  2  9  6  7
62   9 11 10  5  3  6  7  1  8  2
60   8  2  5 10  9  4 11  3  7  1
56   2  4  9  8  1  5  3  7 11  6
58   4  7  3  6  2 10  1 11  5  9
60   7  1  2  3  8 11  5 10  9  4
59   5  3  6 11  4  1  9  8  2 10
62  11  9  7  2  6  3 10  5  1  8
65   6 10 11  7  5  9  8  2  4  3
59  10  8  1  9 11  2  6  4  3  5
```

We model Kakuro as a Constraint Satisfaction Problem. We use Forward Checking but we do not use channeling constraints. In a playout we choose the variable with the least number of possible values and we assign a value according to the policy (which is uniform in the case of sampling and which uses the softmax of the biases in the case of the prior policy). When a variable has an empty domain the playout is stopped and the score is returned. The score is the opposite of the number of remaining unassigned variables when the Kakuro is not complete and the number of rows and columns that sum to the hint when all variables are assigned.

The code for a move contains the number of times the value appears in the same row, the number of times it appears in the same column, the remaining sum to reach in the row and the remaining sum to reach in the column.

Figure 3 gives the numbers of appearance of the Kakuro priors. It was calculated using 10,000 solved problems randomly generated. With the priors equal to 0.0 or 1.0, it rediscovers the hard constraints manually programmed in specialized Kakuro solvers [33] that compute the impossible values for a given sum. However our prior is more precise than that since it takes into account the remaining row/column sums as well as the number of appearances of the value in the same row/column. There are many priors different from 0.0 and 1.0 that model something different from the hard constraints and that capture some probabilistic properties of the values to assign.

Table 2 gives the results for Monte Carlo Search with and without the prior. Using the prior both sampling and GNRPA usually find the solution at the first playout whereas without the prior both sampling and NRPA take much more time.

4.3 Inverse RNA Folding

The design of RNA molecules with specific properties is an important topic for health related research. For example, many viruses rely on RNAs to infect and replicate inside a host: this is the case for coronaviruses [23] and Dengue viruses. Understanding viral RNAs is essential for the scientific community to develop novel drugs in response to pandemics like COVID-19 [21].

RNA molecules are long molecules composed of four possible nucleotides. Molecules can be represented as strings composed of the four characters 'A', 'C', 'G', 'U'. For RNA molecules of length N, the size of the state space of possible strings is exponential in N. It can be very large for long molecules. The molecules of the Eterna100 benchmark we use can have hundreds of nucleotides. The sequence of nucleotides folds back on itself to form its secondary structure. It is possible to find in a polynomial time the folded structure of a given sequence. However, the opposite which is to find a sequence that folds into a predefined structure, that is the Inverse RNA Folding problem, is hard [1].

The state space is the set of all sequences that are consistent with the secondary structure given as input. The secondary structure is a sequence of characters. The possible characters are '.', '(' and ')'. For each '.' in the input sequence there are four possible characters in the nucleotide sequence: 'A', 'C', 'G' and 'U'. Each '(' character is associated to the ')' character that closes the expression it has opened (e.g. when the same number of '(' and ')' are in between the two). Six pairs of characters are possible to replace the '(' and the corresponding ')': 'CG', 'GC', 'GU', 'UG', 'AU' and 'UA'. When a nucleotide sequence is complete, the ViennaRNA package [22] is used to fold the sequence and verify if it folds into the target structure.

We evaluate different Monte Carlo Search algorithms 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 brackets and dots each representing a base. The matching brackets symbolize the paired bases and the dots the unpaired ones. The puzzle is solved when a sequence of the four nucleotides 'A', 'C', 'G' and 'U', folds according to the target structure. In some puzzles, the value of certain bases is imposed.

Human experts have solved the 100 problems of the benchmark. No program has solved all problems. The best score so far for a program is 95/100 by NEMO, NEsted MOnte Carlo RNA Puzzle Solver [25] and by GNRPA using the NEMO prior [8].

To compute the prior we use the Rfam database [21]. Rfam is the database of non-coding RNA families. We use the 85,232 RNA sequences from Rfam associated to their target folding. The NGRAM prior consists in statistics on the occurrence of two following moves.

On the contrary of the hand crafted heuristics of NEMO, the NGRAM prior has been learned on the Rfam database which is separated from the Eterna100 benchmark. The computation of the NGRAM prior on the Rfam database is a more general and simple way to create priors and it is not specific to the Eterna100 benchmark.

Algorithm 5 gives the function used to compute the NGRAM prior, t is the set of target structures and s is the set of RNA sequences that fold in the target structures. The

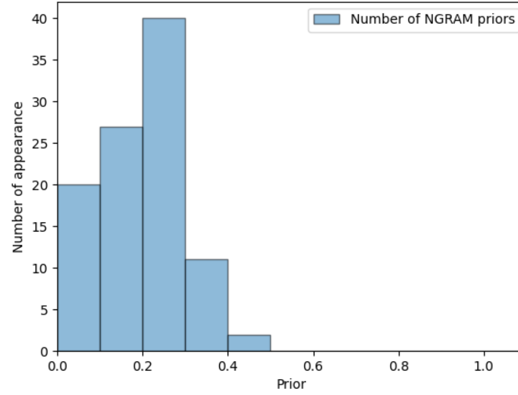

Fig. 4: The distribution of the priors for Inverse RNA Folding. The y-axis gives the number of priors in each range of values. There are 6 possible moves for a '(' and 4 possible moves for a '.' in the target structure. This makes 10 possibilities for the previous move in the NGRAM and again 10 possibilities for the current move. Therefore there are 100 different priors. On the contrary of LSC and Kakuro the distribution of the priors is mainly on small values. The smallest prior is equal to 0.010083 and the greatest prior is equal to 0.437825.

Table 3: Number of Eterna100 problems solved by different algorithms and various search time limits in seconds. GNRPA is much better than NRPA. The NGRAM prior is better than the NEMO prior. The temperature for the NGRAM prior is $\tau = 6$. Sampling with the NGRAM prior is better than sampling with the NEMO prior. Sampling with a prior is much better than uniform sampling.

Algorithm	32s	64s	128s	256s	512s	1,024s	2,048s	4,096s
Sampling	11	11	11	12	14	16	16	17
Sampling NEMO prior	51	55	57	60	61	61	62	64
Sampling NGRAM prior	57	65	68	69	69	69	69	69
NRPA	28	33	41	48	57	59	61	65
GNRPA NEMO prior	68	69	74	77	78	79	81	81
GNRPA NGRAM prior	70	75	78	79	80	81	82	85

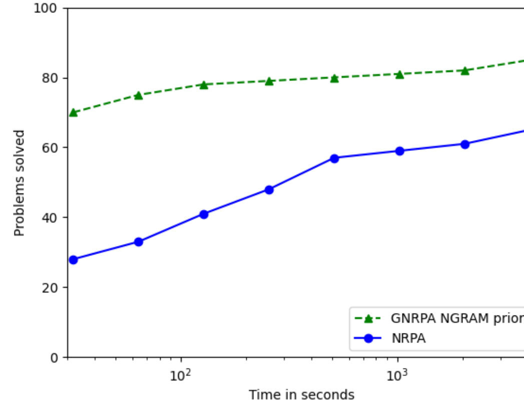

Fig. 5: The evolution with the logarithm of the search time of the number of Eterna100 problems solved by NRPA and GNRPA NGRAM prior.

1: Replay (t, s)
2: **for** $i \in [0..len(s)]$ **do**
3: **for** $j \in [0..len(s[i]) - 1]$ **do**
4: $n \leftarrow code(t[i][j], s[i][j], t[i][j+1], s[i][j+1])$
5: $count[n] \leftarrow count[n] + 1$
6: **for** $m \in moves(t[i][j+1])$ **do**
7: $n \leftarrow code(t[i][j], s[i][j], t[i][j+1], m)$
8: $nb[n] \leftarrow nb[n] + 1$
9: **end for**
10: **end for**
11: **end for**

Algorithm 5: The algorithm to count the NGRAMs. It takes as arguments the target structures t and the corresponding solutions s as sequences of moves. It counts the number of times two following characters in the target structure and the corresponding two moves happen in the Rfam database. It also counts the number of appearance for all possible moves.

output of the algorithm are the $count$ and nb arrays that are used to calculate the prior of a move.

The code to calculate the statistics computes $count[code(t_p, m_p, t, k)]$ the number of times an NGRAM coded as $code(t_p, m_p, t, k)$ appears in the solution sequences of the Rfam database. We only compute the NGRAMs of size one, containing m the move to play, m_p the previous move, t the target folding character and t_p the previous target folding character. Figure 4 gives the distribution of the priors.

We define the bias β_m as:

$$\beta_m = \tau * log(\frac{count[code(t_p, m_p, t, m)]}{nb[code(t_p, m_p, t, m)]})$$

The score of a sequence of nucleotide is computed the same way as NEMO [25] using the ViennaRNA package [22].

Table 3 gives the evolution of the number of problems solved with time for different Monte Carlo Search algorithms. GNRPA with the NGRAM prior gives the best results. Note that the NEMO prior we used is a subset of the priors used in NEMO. It uses the heuristic functions on the pairs of bases. The pairs of bases heuristics are the main components of the NEMO prior. The same subset of heuristics were already used with GNRPA [8], equaling the 95/100 score of NEMO. This score was reached using various optimizations of GNRPA when we use a standard GNRPA in our paper. It explains why we only reach 85 solved problems and why the NEMO prior only reaches 81 solved problems.

Figure 5 gives a graphical comparison of NRPA and GNRPA NGRAM prior for the Eterna100 problems. The values for the numbers of solved problems are the same as in the Table 3. The time scale is logarithmic.

5 Conclusion

Calculating statistics about moves in solved combinatorial problems enables to create a prior for Monte Carlo Search. This prior is easy to compute and has a negligible computation time during sampling. It is a large improvement of Monte Carlo search for three difficult combinatorial problems: Latin Square Completion, Kakuro and Inverse RNA Folding. The method is general and can easily be applied to other difficult combinatorial problems.

As future works, the method could be improved for the combinatorial problems we tried simply using more elaborate codes for the moves. We could bias the policy according to other properties of the moves and of the states than the simple ones we used. The method should also be tried on other difficult combinatorial problems in order to evaluate the gains of using it. The problems we tried are decision problems, it would be interesting to also try optimization problems. The generation of the solved problems would be more time consuming for optimization problems but it would only be done once before the use of the prior in Monte Carlo Search. The sampling time with the prior would be similar to the sampling time without the prior but the scores obtained sampling with the prior could be much better than without the prior.

References

1. Edouard Bonnet, Paweł Rzążewski, and Florian Sikora. Designing RNA secondary structures is hard. *Journal of Computational Biology*, 27(3), 2020.
2. Bruno Bouzy. Monte-carlo fork search for cooperative path-finding. In *Computer Games Workshop at IJCAI*, pages 1–15, 2013.
3. Bruno Bouzy. Burnt pancake problem: New lower bounds on the diameter and new experimental optimality ratios. In *SOCS*, pages 119–120, 2016.
4. Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.
5. Tristan Cazenave. Monte-Carlo Kakuro. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers*, volume 6048 of *Lecture Notes in Computer Science*, pages 45–54. Springer, 2009.
6. Tristan Cazenave. Nested Monte-Carlo Search. In Craig Boutilier, editor, *IJCAI*, pages 456–461, 2009.
7. Tristan Cazenave. Generalized nested rollout policy adaptation. In *Monte Carlo Search at IJCAI*, 2020.
8. Tristan Cazenave and Thomas Fournier. Monte Carlo inverse folding. In *Monte Carlo Search at IJCAI*, 2020.
9. Tristan Cazenave, Jean-Yves Lucas, Thomas Triboulet, and Hyoseok Kim. Policy adaptation for vehicle routing. *Ai Communications*, 34(1):21–35, 2021.
10. Tristan Cazenave, Abdallah Saffidine, Michael John Schofield, and Michael Thielscher. Nested monte carlo search for two-player games. In *AAAI*, pages 687–693, 2016.
11. Tristan Cazenave and Fabien Teytaud. 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*, pages 42–54, 2012.
12. Charles J. Colbourn. The complexity of completing partial latin squares. *Discrete Applied Mathematics*, 8(1):25–30, 1984.
13. Stefan Edelkamp, Max Gath, Tristan Cazenave, and Fabien Teytaud. Algorithm and knowledge engineering for the tsptw problem. In *Computational Intelligence in Scheduling (SCIS), 2013 IEEE Symposium on*, pages 44–51. IEEE, 2013.
14. Stefan Edelkamp, Max Gath, Christoph Greulich, Malte Humann, Otthein Herzog, and Michael Lawo. Monte-Carlo tree search for logistics. In *Commercial Transport*, pages 427–440. Springer International Publishing, 2016.
15. Stefan Edelkamp, Max Gath, and Moritz Rohde. Monte-Carlo tree search for 3d packing with object orientation. In *KI 2014: Advances in Artificial Intelligence*, pages 285–296. Springer International Publishing, 2014.
16. Stefan Edelkamp and Christoph Greulich. Solving physical traveling salesman problems with policy adaptation. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.
17. Stefan Edelkamp and Zhihao Tang. Monte-Carlo tree search for the multiple sequence alignment problem. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015*, pages 9–17. AAAI Press, 2015.
18. Maxime Elkael, Massinissa Ait Aba, Andrea Araldo, Hind Castel-Taleb, and Badii Jouaber. Monkey business: Reinforcement learning meets neighborhood search for virtual network embedding. *Computer Networks*, 216:109204, 2022.

19. Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, volume 8, pages 259–264, 2008.
20. Yan Jin and Jin-Kao Hao. Solving the latin square completion problem by memetic graph coloring. *IEEE Transactions on Evolutionary Computation*, 23(6):1015–1028, 2019.
21. Ioanna Kalvari, Eric P Nawrocki, Nancy Ontiveros-Palacios, Joanna Argasinska, Kevin Lamkiewicz, Manja Marz, Sam Griffiths-Jones, Claire Toffano-Nioche, Daniel Gautheret, Zasha Weinberg, et al. Rfam 14: expanded coverage of metagenomic, viral and microrna families. *Nucleic Acids Research*, 49(D1):D192–D200, 2021.
22. Ronny Lorenz, Stephan H Bernhart, Christian Höner zu Siederdissen, Hakim Tafer, Christoph Flamm, Peter F Stadler, and Ivo L Hofacker. Viennarna package 2.0. *Algorithms for molecular biology*, 6:1–14, 2011.
23. Ramakanth Madhugiri, Nadja Karl, Daniel Petersen, Kevin Lamkiewicz, Markus Fricke, Ulrike Wend, Robina Scheuer, Manja Marz, and John Ziebuhr. Structural and functional conservation of cis-acting rna elements in coronavirus 5'-terminal genome regions. *Virology*, 517:44–55, 2018.
24. Jean Méhat and Tristan Cazenave. 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.
25. Fernando Portela. An unexpectedly effective Monte Carlo technique for the RNA inverse folding problem. *BioRxiv*, page 345587, 2018.
26. Simon M. Poulding and Robert Feldt. Generating structured test data with specific properties using nested Monte-Carlo search. In *GECCO*, pages 1279–1286, 2014.
27. Simon M. Poulding and Robert Feldt. Heuristic model checking using a Monte-Carlo tree search algorithm. In *GECCO*, pages 1359–1366, 2015.
28. Arpad Rimmel, Fabien Teytaud, and Tristan Cazenave. Optimization of the Nested Monte-Carlo algorithm on the traveling salesman problem with time windows. In *EvoApplications*, volume 6625 of *LNCS*, pages 501–510. Springer, 2011.
29. Christopher D. Rosin. Nested rollout policy adaptation for Monte Carlo Tree Search. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 649–654, 2011.
30. Milo Roucairol and Tristan Cazenave. Comparing search algorithms on the retrosynthesis problem. In *AI to Accelerate Science and Engineering at AAAI 2023*. 2023.
31. Oliver Ruepp and Markus Holzer. The computational complexity of the kakuro puzzle, revisited. In *International Conference on Fun with Algorithms*, pages 319–330. Springer, 2010.
32. David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.
33. Helmut Simonis. Kakuro as a constraint problem. *Proc. seventh Int. Works. on Constraint Modelling and Reformulation*, 2008.