Monte Carlo Graph Coloring

Tristan Cazenave, Benjamin Negrevergne, and Florian Sikora

Université Paris-Dauphine, PSL University, CNRS, LAMSADE, 75016 Paris, France
{Tristan.Cazenave,Benjamin.Negrevergne,Florian.Sikora}@dauphine.fr

Abstract. Graph Coloring is probably one of the most studied and famous problem in graph algorithms. Exact methods fail to solve instances with more than few hundred vertices, therefore, a large number of heuristics have been proposed. Nested Monte Carlo Search (NMCS) and Nested Rollout Policy Adaptation (NRPA) are Monte Carlo search algorithms for single player games. Surprisingly, few work has been dedicated to evaluating Monte Carlo search algorithms to combinatorial graph problems. In this paper we expose how to efficiently apply Monte Carlo search to Graph Coloring and compare this approach to existing ones.

1 Introduction

Given a graph G, a proper coloration of G consists in assigning a color to each vertex of the graph such that no adjacent vertices receive the same color. The chromatic number $\chi(G)$ of G is the minimum number of colors required to have a proper coloration for G. Determining the chromatic number of a graph is probably one of the most studied topics in graph algorithms and discrete mathematics. It has many applications, including scheduling, timetabling, or communication networks (see references in [24]). Unfortunately, identifying the chromatic number is notoriously hard to solve: it is already NP-hard even if the question is to decide if the graph can be colored with 3 colors, and it is essentially completely not-approximable [30].

To cope with this difficulty, the research community has tried a variety of different approaches: mathematical programming [24], exact moderately exponential algorithms [1], approximation algorithms on special graph classes [8], algorithms of parameterized complexity for structural parameters and data reduction [22,4,25,23], heuristics, meta-heuristics, etc. In practice exact methods generally fail to color graphs with more than few hundred vertices [24], so a large number of publications on graph coloring algorithms focus on the design and improvement of heuristics approaches.

Early heuristics for graph coloring were often based on pure local search strategies such as *TabuSearch* [19]. Nowadays, most efficient modern algorithms are still based a local search strategy, but they combine it with sophisticated exploration techniques to escape local minima (e.g. *Variable Neighborhood Search* [26] and *Variable Space Search* [20]). Building on a the idea of combining local search with more exploratory search procedures, Fleurent and Ferland have proposed to use the framework of hybrid algorithms which combine a local search operator with a population based algorithm [16]. This idea has inspired a lot of research in the field (see for example [17]),

and ultimately led to the state-of-the-art algorithm HEAD [27] (*Hybrid Evolutionary Algorithm in Duet*).

In comparison with hybrid algorithms based on local search, very little work has been dedicated to evaluating the performance of Monte-Carlo for graph coloring problem (except [9]). This is probably because the idea of discovering highly constrained solutions through random sampling seems counter-intuitive at first. However, modern Monte-Carlo based algorithm naturally combine random search (which provides exploration) with a tree search driven by a stochastic policy learned during the search, (which help improving good local solutions). These two features make modern Monte-Carlo based algorithms good candidates for the graph coloring problem.

In this paper, we evaluate the performance of two Monte-Carlo based algorithm, *Nested Monte Carlo Search* (NMCS) [5] and *Nested Rollout Policy Adaptation* (NRPA) [29], for the graph coloring problem. As we will show, our modeling of the coloring problem as a Monte-Carlo search algorithm provides good performance and can compete with state-of-the-art hybrid algorithms which have been studied and improved over the past 30 years.

In Section 2, we review related work concerning Monte Carlo Search methods and describe in Section 3 the two we will use in this paper. In Section 4, we discuss various modeling choices for our approach. In Section 5 we describe the other algorithms for graph coloring that we will compare to ours. Finally, in Section 6, we conduct thorough experiments to demonstrate the performance of NRPA.

2 Monte Carlo Search methods and Combinatorial Problems

Monte Carlo Tree Search algorithms (MCTS) have been most successful in the area of game artificial intelligence [3], and have obtained state-of-the-art in this field. They have also been applied to a variety of other problems in combinatorial optimization problems, but they remain marginally used in this area. For example, NRPA has been applied to the Traveling Salesman with Time Windows problem [7,10], and other applications also deal with 3D Packing with Object Orientation [12], the physical traveling salesman problem [13], the Multiple Sequence Alignment problem [14] or Logistics [11].

In 2017, Edelkamp and co-workers have applied Monte Carlo Search to graph coloring [9]. They compare various Monte-Carlo search algorithms such as NMCS, NRPA as well as a SAT-based approach. In their experiments they report that the best results were obtained with NMCS which contrast with our results in this paper. We propose to optimize further the modelling of the problem using node ordering, refined scoring and a different Adapt function. Our optimizations improve much the search time compare to the alternative modellings.

3 Nested Monte Carlo Search

In this section we describe the two Monte Carlo search algorithms that we have considered in the rest of this paper: *Nested Monte Carlo Search* (NMCS) and *Nested Rollout Policy Adaptation* (NRPA).

As most Monte-Carlo based algorithms, NMCS and NRPA produce a good solution by generating a large number of random sequences of branching decisions (a.k.a moves). The best sequence according to some objective function is then returned as a final solution to the problem. Since the quality of final sequence directly depends on the quality of the random sequences generated during the search, NMCS and NRPA combine a variety of techniques to improve the quality of the random sequence generator such as tree search, policy adaptation or nested algorithms.

At the lowest recursive level, the generation of random sequences is driven by a stochastic policy (a probability distribution over the moves). Random sequences are generated based on this policy by sampling moves from the policy using Gibbs sampling, as described in Algorithm 1. If we have access to background knowledge, it can be encoded as a non-uniform distribution over the moves in the policy. Otherwise, the initial stochastic policy assigns equal probability to each move.

Algorithm 1 The playout algorithm

playout ($state$, $policy$)
$sequence \leftarrow$ []
while true **do**
  if $state$ is terminal **then**
    return (score ($state$), $sequence$)
  end if
  $z \leftarrow 0.0$
  for m in possible moves for $state$ **do**
    $z \leftarrow z$ + exp ($policy$ [m])
  end for
  choose a move m with probability $\frac{exp(policy[m])}{z}$
  $state \leftarrow$ play ($state$, m)
  $sequence \leftarrow sequence + m$
end while

In NMCS, the policy remains the same throughout the execution of the algorithm. However, the policy is combined with a tree search to improve the quality over a simple random sequence generator. At each step, each possible move is evaluated by completing the partial solution into a complete one using moves sampled from the policy. Whichever intermediate move has led to the best completed sequence, is selected and added to the current sequence. (See Algorithm 2.) The same procedure is repeated to choose the following move, until the sequence has reached a terminal state.

A major difference between NMCS and NRPA, is the fact that NRPA uses a stochastic policy that is *learned* during the search. At the beginning of the algorithm, the policy is initialized uniformly and later improved using gradient descent steps based the best sequence discovered so far (See. Algorithm 3). The procedure used to update the policy from a given sequence is given in Algorithm 4. Note that this difference is crucial because unlike NMCS, NRPA is able to *acquire* background knowledge about the problem being solved, and does not require the user to specify it. Ultimately, this knowledge will contribute to speed up the discovery of a good solution.

Algorithm 2 The NMCS algorithm.

```
NMCS (state, level)
if level == 0 then
  return playout (state, uniform)
end if
BestSequenceOfLevel ← ∅
while state is not terminal do
  for m in possible moves for state do
    s ← play (state, m)
    NMCS (s, level − 1)
    update BestSequenceOfLevel
  end for
  bestMove ← move of the BestSequenceOfLevel
  state ← play (state, bestMove)
end while
```

Finally, both algorithms are nested, meaning that at the lowest recursive level, weak random policies are used to sample a large number of low quality sequences, and produce a search policy of intermediate quality. At the recursive level above, this policy is used to produce sequence of high quality. This procedure is applied recursively, in general 4 or 5 times. In both algorithm the recursive level (denoted l) is a crucial parameter. Increasing l increases the quality of the final solution at the cost of more CPU time. In practice it is generally set to 4 or 5 recursive levels depending on the time budget and the computational resources available.

Algorithm 3 The NRPA algorithm.

```
NRPA (level, policy)
if level == 0 then
  return playout (root, policy)
end if
bestScore ← −∞
for N iterations do
  (result,new) ← NRPA(level − 1, policy)
  if result ≥ bestScore then
    bestScore ← result
    seq ← new
  end if
  policy ← Adapt (policy, seq)
end for
return (bestScore, seq)
```

Algorithm 4 The Adapt algorithm

```
Adapt (policy, sequence)
polp ← policy
state ← root
for move in sequence do
    polp [move] ← polp [move] + α
    z ← 0.0
    for m in possible moves for state do
        z ← z + exp (policy [m])
    end for
    for m in possible moves for state do
        polp [m] ← polp [m] - α * exp(policy[m]) / z
    end for
    state ← play (state, move)
end for
policy ← polp
```

4 Graph Coloring as a Monte Carlo Search Problem

In this section we discuss several alternative models to capture the Graph Coloring problem as a Monte Carlo Search problem. Remark that we focus *decision* problem (i.e. deciding if a graph can be colored with a given number of colors).

We start by defining the possible moves, and then present the node ordering heuristic, and deal with the question of generating valid moves. Finally, we discuss the objective function and as well as an optimization of the adapt function for the graph coloring problem.

4.1 Legal Moves

In the context of the graph coloring problem, a *move* consists in assigning a particular color to an uncolored vertex of the graph. Thus, given a graph $G = (V, E)$ and a set of colors C, a move is a pair (v, c) where v is an uncolored vertex in V, and c is any color in c.

NMCS and NRPA only consider *legal moves* at each step, and lowering the number of legal moves is a key performance issue for both algorithms. In NMCS, a large number of legal moves leads to a very large branching factor which slows down the algorithm at each recursive level. For NRPA a large number of legal moves results in a large policy vector, which makes it more difficult to train with comparatively less training examples.

A first naive approach consists in considering every possible move at every step, leading to a number of possible moves that can be as big as $|V| \times |C|$ in the initial condition. This solution results in poor efficiency and low quality solution which we do not report here. To lower the number of possible moves down, we adopt a different model in which each move vertex is considered in a particular order (e.g. random order). At each step, only one node and all its legal coloration are considered. This reduces the maximum number of moves from $|V| \times |C|$ to $|C|$. As we will see, it leads to good

results in practice. However, imposing an order over the vertices induces a strong bias over the exploration of the search space, which we study in the next section.

4.2 Node order

The naive approach to node ordering is to fix a predefined or random order of the nodes and to color them in this order. A better heuristic is DSatur [2]. It chooses as the next node to color the node that has the less possible colors. In case multiple nodes have the same minimal number of possible colors it break ties by choosing the node that has the most neighbors. DSatur is a good heuristic to order nodes for NRPA since it propagates the constraints in the graph and avoids choosing colors for a node that would reveal inconsistent later due to more constrained neighbors. For example if a node has only one possible color it will always be chosen first by DSatur. By doing so the neighboring nodes have one less possible color and it avoids taking this impossible colors for neighboring nodes which would not have been the case if the one color node had been chosen later.

4.3 Selective Search

When trying possible colors for a node it is not wise to choose a color that is already assigned to a neighboring node. In order to avoid as much as possible bad branching decisions we use forward checking. When selecting the color for a node, all the colors of the neighboring nodes are removed from the set of possible colors for the node. This is related to selective NRPA [6] where heuristics are used to avoid bad moves. However in our case of Graph Coloring the moves that are discarded are moves that can never be part of a valid solution. So it is safe to remove them. Inconsistent colors are never considered as possible moves except if there a no possible color for a node since neighboring nodes already contains all the available colors. In this case all colors are considered possible and the algorithm chooses a color for the node even if it is inconsistent.

4.4 Scoring function

Another design choice is the way to score a playout. In [5] the depth of the playout was used for Sudoku and the playouts were stopped when reaching an inconsistent state, i.e. a state where a variable has no more possible values. For Graph Coloring we use a more informed score. We count the number of inconsistent edges, i.e. monochromatic edges. If there are two adjacent vertices with the same color, the score decrease by one. A score equal to the number of edges of the graph means that we have found a solution. Note that we also tried a scoring function mixing both the number of colors and the number of inconsistent edges (trying to decrease both), which would allow the algorithm to solve the optimization problem directly, but it didn't work well.

We also experimented with NMCS. For the sake of completeness NMCS is given in algorithm 2. The score of the playouts, the selection of edges, and the selectivity of colors in NMCS are the same as in NRPA.

4.5 Coding the moves

In NRPA it is important to design how moves are coded. There is a bijection between moves and integer such that moves are associated to weights. We choose to use a simple coding for our moves: the index of a node multiplied by the number of colors plus the index of the color in the move.

4.6 Adapt all the colors

When modifying the weights with the adapt function, there are two options. The first one is to modify the weights of the possible moves and to adapt using only the probabilities of the moves that can be played in the current state. The second one is to modify the weights for all the colors, including the colors that were discarded as possible moves since they were inconsistent with neighboring nodes.

The standard Adapt function is given in algorithm 4. The modified function that modifies the probabilities for all the colors is given in algorithm 5.

Algorithm 5 The AdaptAll algorithm

AdaptAll ($policy$, $sequence$)
$polp \leftarrow policy$
$state \leftarrow root$
for $move$ in $sequence$ **do**
  $polp$ [code($move$)] $\leftarrow$ $polp$ [code($move$)] + α
  $z \leftarrow 0.0$
  for m in all possible colors even the illegal ones **do**
    $z \leftarrow z$ + exp ($policy$ [code(m)])
  end for
  for m in all possible colors even the illegal ones **do**
    $polp$ [code(m)] $\leftarrow$ $polp$ [code(m)] - $\alpha * \frac{exp(policy[code(m)])}{z}$
  end for
  $state \leftarrow$ play ($state$, $move$)
end for
$policy \leftarrow polp$

5 Compared approaches for graph coloring

In this section we present the other Graph Coloring algorithms that we used to compare with our approach.

5.1 SAT

We used the following SAT encoding to decide if one can color a graph $G = (V, E)$ with k colors (this formulation is for example used in [21]). We add a variable $x_{v,i}$ for

each $v \in V$ and each $i \in [k]$. Then, for each $v \in V$, we add a clause $(\bigvee_i x_{v,i})$, ensuring that each vertex receives a color, and for each edge $uv \in E$ and each color $i \in [k]$, we add a clause $(\neg x_{u,i} \vee \neg x_{v,i})$ such that adjacent vertices receives different colors. This formula is true iff there is a k-coloration of G. Note that if there is no truth assignment for the formula, it tells that the graph is not k-colorable. However, we will not use this in our experiments.

As a solver, we used MiniSat to solve the built formula [15].

5.2 HEAD

HEAD [27] is an hybrid meta-heuristic, more precisely a memetic algorithm, mixing a local search procedure (Tabu-Search) with an evolutionary algorithm. It is based on the *Hybrid Evolutionary Algorithm* (HEA) by Galinier and Hao [17].

The general principle of HEA and HEAD is to start with a population of individuals, which are first improved using a local search procedure. Then, a crossover operator is applied to the best individuals in order to generate new diverse individuals and the procedure is repeated for a fixed number of steps which are called generations.

However general crossover operators do not work well for the graph coloring problem, so the main contribution of HEA is a specialized crossover operator. In HEA each individual is a partition of vertices into color classes, and the crossover operator is required to preserve color classes or subset of color classes.

HEAD builds on HEA by introducing various improvements including an original method to maintain diversity inside the population: individuals from earlier generations are re-introduced as candidate individuals in later generations. Using these technique, HEAD has been able to rediscover known coloring at much lower computational cost that earlier approaches [27].

The source-code of HEAD is available online[1]. Authors provided experiments showing good performances on the classical benchmarks and is probably the faster heuristic to date.

5.3 Greedy Coloring

For greedy coloring we use the same generation of possible moves as NRPA except that we only play one playout and that the maximum numbers of colors is not fixed. The order in which vertices are visited during this greedy coloring is the one given by DSatur [2]. Greedy Coloring is used to establish initial upper bounds for NRPA, NMCS, SAT and HEAD that are lowered down using search to establish better upper bounds.

6 Experiments

In this section, we compare the performance of two Monte-Carlo based approaches described in Section 3 and 4 with the other approaches described in Section 5.

[1] https://github.com/graphcoloring/HEAD/

6.1 Experimental Protocol

Execution strategy: In practice we observe a high variance in runtimes and best results throughout the different runs of the same algorithm. To reduce the variance in the results, and allow a fair comparison, we proceed as follows: each algorithm is executed 5 times for a given number of color k with a timeout of 30 minutes. If the algorithm discovers at least one valid k-coloring for the graph instance, we decrease k by one, and repeat this procedure until the algorithm is unable to discover a k-coloring within the timeout limit. Then, we report the lowest k for which the algorithm was able to discover a coloring (denoted **UB** in the result tables), as well as the success rate for this lowest k (denoted **Reached**). The initial value of k to start with is determined with the simple greedy algorithm with nodes ordered according to DSatur (denoted **UBI**). Sometimes the algorithm is unable to improve over the simple greedy algorithm, which we signal with a '–' in the result table (unless the greedy algorithm has already discovered the minimum number χ).

Test instances: We used standard benchmark instances available on the website maintained by Gualandi and Chiarandini [18], collected from DIMACS benchmark. This benchmark has been used extensively to evaluate graph coloring algorithms and is now considered to be the standard set for experimentation [24] in this field. Moreover, because these instances have been extensively studied, the optimal chromatic number χ is known for most of them.

These instances are sorted by difficulty. Instances marked NP-m (resp. NP-d) should be solved in less than a hour (resp. than a day). For the harder instances marked NP-?, either the chromatic number is unknown or the time needed to solve them is unknown to [18].

Hardware and implementations details: Every execution reported in this experimental section has been conducted on a Intel Xeon E5-2630 v3 (Haswell, 2.40GHz). Although some algorithm support parallel execution, (e.g. NRPA and HEAD), we on report execution times on sequential execution (using one thread) to reduce the variance in execution times, and to allow meaningful comparison with other purely sequential algorithms.

The peak memory usage is limited to 4GB which is not a limitation for any of the algorithm except for the SAT model which runs out of memory sometimes.

We implemented algorithms using SAT, NRPA and NMCS in C++, using the Boost Graph Library. For NRPA, we use our implementation described in [28] and available online[2]. In our experiments we use $l = 7, N = 100$ for NRPA[3]. For NMCS, we use increasing nesting levels. This mean that we start the search with a level of 1, and if no solution is found, we increment the level and repeat.

Table 1. Results for the easy instances (marked NP-m) with a timeout of 30 minutes.

					NMCS		NRPA		SAT		HEAD	
Instance	$\|V\|$	$\|E\|$	χ	**UBI**	**UB**	**Reached**	**UB**	**Reached**	**UB**	**Reached**	**UB**	**Reached**
1-FullIns_4	93	593	5	5	**5**	100%	**5**	100%	**5**	100%	**5**	100%
2-FullIns_4	212	1621	6	6	**6**	100%	**6**	100%	**6**	100%	**6**	100%
3-FullIns_3	80	346	6	6	**6**	100%	**6**	100%	**6**	100%	**6**	100%
4-FullIns_3	114	541	7	7	**7**	100%	**7**	100%	**7**	100%	**7**	100%
5-FullIns_3	154	792	8	8	**8**	100%	**8**	100%	**8**	100%	**8**	100%
ash608GPIA	1216	7844	4	6	**4**	100%	**4**	100%	**4**	100%	**4**	100%
ash958GPIA	1916	12506	4	6	**4**	60%	**4**	100%	**4**	100%	**4**	100%
le450_15a	450	8168	15	17	**15**	60%	**15**	100%	**15**	100%	**15**	100%
mug100_1	100	166	4	4	**4**	100%	**4**	100%	**4**	100%	**4**	100%
mug100_25	100	166	4	4	**4**	100%	**4**	100%	**4**	100%	**4**	100%
qg.order40	1600	62400	40	42	**40**	100%	**40**	100%	**40**	100%	**40**	100%
wap05a	905	43081	50	50	**50**	100%	**50**	100%	**50**	100%	**50**	100%
myciel6	95	755	7	7	**7**	100%	**7**	100%	**7**	100%	**7**	100%
school1_nsh	352	14612	14	26	**14**	100%	**14**	100%	**14**	100%	**14**	100%
Avg. ratio to χ					1.0000		1.0000		1.0000		1.0000	

Table 2. Results for the instances marked NP-h by [18] with a timeout of 30 minutes.

					NMCS		NRPA		SAT		HEAD	
Instance	$\|V\|$	$\|E\|$	χ	**UBI**	**UB**	**Reached**	**UB**	**Reached**	**UB**	**Reached**	**UB**	**Reached**
flat300_28_0	300	21695	28	41	38	20%	35	20%	39	100%	**31**	100%
r1000.5	1000	238267	234	248	243	20%	**240**	40%	247	100%	248	–
r250.5	250	14849	65	67	**65**	100%	**65**	100%	**65**	100%	66	40%
DSJR500.5	500	58862	122	132	125	60%	**122**	40%	126	100%	124	60%
DSJR500.1c	500	121275	85	88	88	–	87	60%	**86**	100%	**86**	80%
DSJC125.5	125	3891	17	23	19	100%	18	100%	19	100%	**17**	100%
DSJC125.9	125	6961	44	50	45	40%	**44**	100%	46	100%	**44**	100%
DSJC250.9	250	27897	72	90	84	20%	76	20%	86	100%	**72**	100%
queen10_10	100	2940	11	14	**11**	60%	**11**	40%	12	100%	**11**	100%
queen11_11	121	3960	11	14	13	100%	13	100%	13	100%	**12**	100%
queen12_12	144	5192	12	16	14	100%	14	100%	14	100%	**13**	100%
queen13_13	169	6656	13	17	15	100%	15	100%	15	100%	**14**	100%
queen14_14	196	4186	14	19	16	100%	16	100%	16	100%	**15**	100%
queen15_15	225	5180	15	20	17	40%	17	100%	18	100%	**16**	100%
Avg. ratio to χ					1.1101		1.0851		1.1276		1.0428	

6.2 Results

We give results of Monte Carlo approaches (NMCS and NRPA) compared to other approaches in tables 1,2,3 and 4.

[2] https://github.com/bnegreve/nrpa

[3] Note that level 7 will probably never be reached in a reasonable amount of time, this is to allow NRPA to continue to search until a solution is found ; this value of N gave often better results than smaller or bigger values

Table 3. Results for the difficult instances marked NP-? by [18] with a timeout of 30 minutes.

					NMCS		NRPA		SAT		HEAD	
Instance	$\|V\|$	$\|E\|$	χ	**UBI**	**UB**	**Reached**	**UB**	**Reached**	**UB**	**Reached**	**UB**	**Reached**
le450_5a	450	5714	5	10	6	20%	**5**	100%	**5**	100%	**5**	100%
le450_5b	450	5734	5	7	6	40%	**5**	20%	**5**	100%	**5**	100%
le450_15b	450	8169	15	17	**15**	100%	**15**	100%	**15**	100%	**15**	100%
le450_15c	450	16680	15	24	22	100%	21	100%	22	100%	**15**	100%
le450_15d	450	16750	15	24	22	100%	20	20%	22	100%	**15**	100%
le450_25c	450	17343	25	28	27	100%	**26**	100%	27	100%	**26**	100%
le450_25d	450	17425	25	29	27	100%	**26**	100%	27	100%	**26**	100%
qg.order60	3600	212400	60	63	**60**	40%	62	100%	61	100%	**60**	100%
qg.order100	10000	990000	100	106	–	20%	102	20%	–	20%	**100**	100%
Avg. ratio to χ					1.7626		1.0963		1.1300		1.0089	

Table 4. Results for the very difficult problems (NP-?) with a timeout of 30 minutes. The chromatic number of these graphs seems unknown and we only know a lower bound via [18].

					NMCS		NRPA		SAT		HEAD	
instance	$\|V\|$	$\|E\|$	χ_{LB}	**UBI**	**UB**	**Reached**	**UB**	**Reached**	**UB**	**Reached**	**UB**	**Reached**
DSJC250.1	250	3218	4	10	9	100%	**8**	40%	9	100%	**8**	100%
DSJC250.5	250	15668	26	37	34	100%	32	100%	35	100%	**28**	100%
DSJC500.1	500	12458	9	16	14	40%	14	100%	15	100%	**12**	100%
DSJC500.5	500	62624	43	65	62	20%	59	80%	63	100%	**48**	100%
DSJC500.9	500	112437	123	163	161	20%	148	20%	163	–	**126**	100%
DSJC1000.1	1000	49629	10	25	25	–	24	100%	25	–	**21**	100%
DSJC1000.5	1000	249826	73	114	114	–	112	40%	114	–	**83**	60%
DSJC1000.9	1000	449449	216	301	301	–	299	40%	301	–	**223**	20%
flat1000_50_0	1000	245000	15	113	113	–	111	40%	113	–	**50**	100%
flat1000_60_0	1000	245830	14	112	112	–	112	–	112	–	**60**	100%
flat1000_76_0	1000	246708	14	115	115	–	110	20%	113	100%	**82**	80%
r1000.1c	1000	485090	96	107	107	–	107	–	105	100%	**98**	20%
abb313GPIA	1557	53356	8	11	11	–	11	–	**9**	100%	**9**	60%
latin_square_10	900	307350	90	129	129	–	121	20%	129	–	**103**	20%
wap01a	2368	110871	41	47	46	60%	45	40%	43	100%	**42**	60%
wap02a	2464	111742	40	46	46	–	45	100%	**42**	100%	**42**	100%
wap03a	4730	286722	40	57	55	40%	55	100%	46	100%	**44**	20%
wap04a	5231	294902	40	46	46	–	46	–	**44**	100%	**44**	100%
wap06a	947	43571	40	44	42	100%	**41**	100%	**41**	100%	42	60%
wap07a	1809	103368	40	47	44	80%	43	60%	**42**	100%	**42**	100%
wap08a	1870	104176	40	44	43	20%	43	100%	**42**	100%	**42**	100%
C2000.5	2000	999836	99	207	207	–	207	–	207	–	**151**	40%
C4000.5	4000	4000268	107	376	376	–	376	–	376	–	**281**	20%
Avg. ratio to χ					2.3746		2.3173		2.3410		1.7027	

First, we observe that the simple greedy algorithm is generally able to discover the χ value for a number of instances in Table 1. When it is not the case, all the approaches

we discuss in this paper have been able to tackle these instances, and discover the χ. However, NMCS does not reach 100% success rate on two of these instances.

When we look at the other results, we can see that NMCS is generally the weakest algorithm, and obtains the worst (higher) ratio to χ for the very difficult instances in Table 3 and 4, often significantly worse than the ratio for NRPA. Note that with a different representation of the problem, authors of [9] reported better results with NMCS, contrary to our results. We also observe that the performance gap between the two approaches is small on the medium NP-m instances (Table 1), but large on the most difficult instances (Table 3). This suggests that with our modeling, NRPA is able to acquire better policies along the execution of the algorithm. The benefit of the learned policies over the tree search becomes more visible in long runs on difficult instances. This motivates the general idea of introducing learning into Monte-Carlo search in order to improve the quality of the search.

We can also see that NRPA is generally better than the SAT model, but SAT is surprisingly good for the difficult wap instances in Table 4, unfortunately, we are unable to explain this result and further experiments are needed.

However, a pair-wise comparison between the upper bounds discovered by NRPA and the ones discovered by HEAD demonstrates that HEAD is better in general. It is worth mentioning that algorithms such as HEAD include specialized graph coloring operators which have been extensively studied over the past decades. In contrast our algorithm is based on a general purpose implementation of NRPA, and only includes the specializations we have described in Section 4. Nevertheless, NRPA is the second best in all the datasets, and is often as good, or even better than HEAD.

In Figure 1 we further analyze the behaviour of the two algorithms by comparing the execution times. We selected the best of 5 runs for each tested number of colors and for each algorithm. The time is the accumulated time, starting with the number of colors computed by the greedy algorithm. The y-axis represents the number of improvement by an algorithm for this instance starting from the greedy coloring (i.e. a value of 3 means that the algorithm gave a coloration with 3 less colors than the greedy algorithm did). For readability, we didn't include instances like C4000.5, where HEAD is very good. It seems that HEAD improves quite fast the number of colors but struggles to improve more over time (this is not always true however), while NRPA seems to benefit of longer running time. Indeed, there is not much improvement after 2000s for HEAD while NRPA continues to find colorations after 3000s. This could be because HEAD is stuck in local optimum and cannot find new solutions, while NRPA continues to improve policies while exploring the search space.

This demonstrate that Monte Carlo based algorithms have the ability to compete with state-of-the-art hybrid algorithms on the graph coloring problem, and deserve further investigation with more optimizations (more specific strategies, restarts...). From a broader perspective, this also shows that a continuous optimization algorithms can be used to solve discrete problems such as the graph coloring problem.

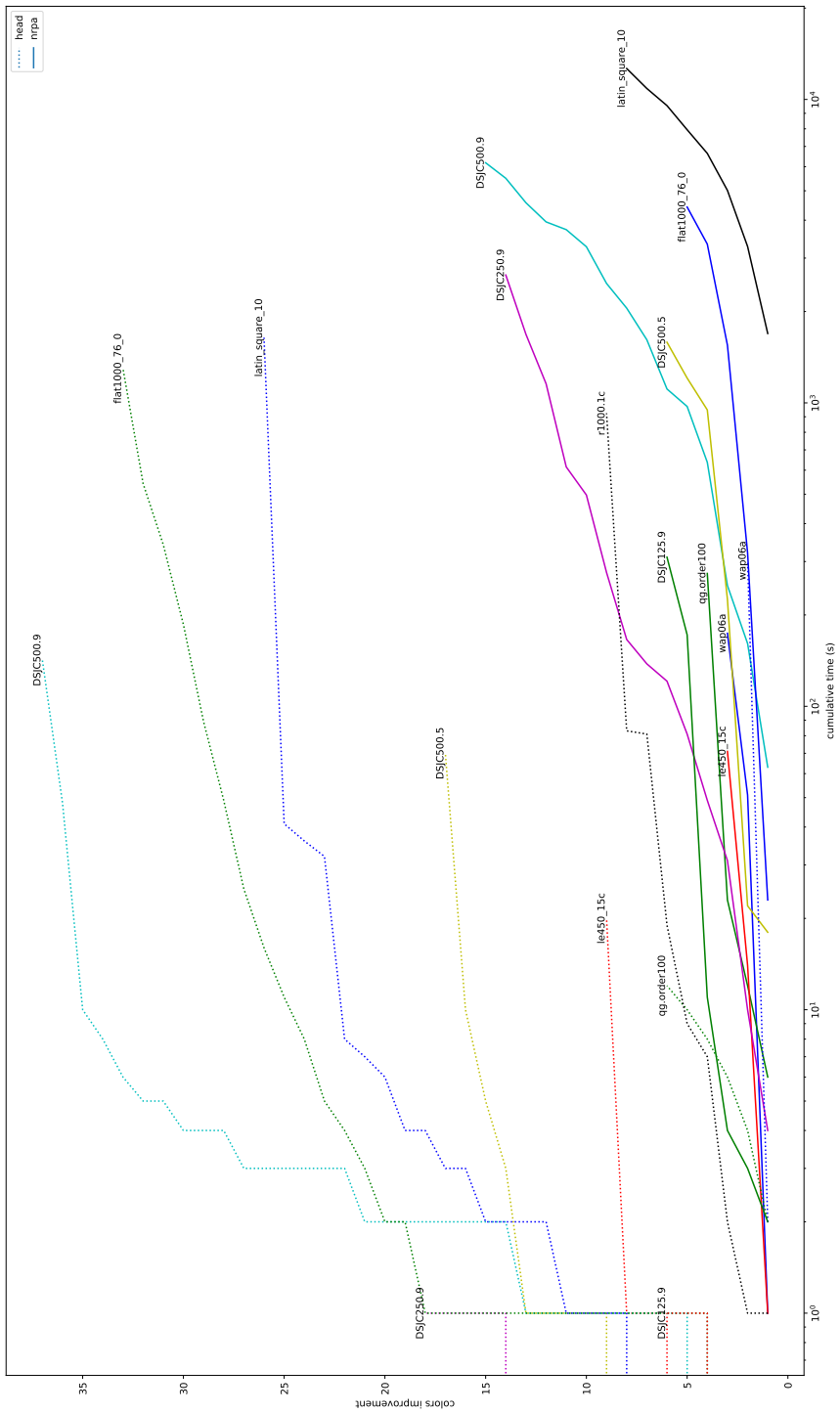

Fig. 1. Comparison of computation times (log scale) between NRPA (solid lines) and HEAD (dotted lines) for some instances.

7 Conclusion

In this paper, we deepen our understanding of Monte Carlo Search algorithms applied to Graph Coloring. Our method is significantly different from most other methods from the literature, and yet, it is able to compete with state-of-the-art algorithms which have been intensively optimized during the past decades. These results suggest that Monte Carlo search combined with policy adaptation are able to explore the search to discover good, yet diverse solutions. And that this technique should be investigated further alongside with more standard approaches.

It would be also interesting to see if NMCS could combine with a good heuristic like HEAD, by either using the local search to improves the solution or by using the genetic algorithm as playouts.

Future works includes the use of Graph Convolution Networks to model more complex policies that can make branching decisions based on the structure of graph at hand, and generalize knowledge from one graph to another.

Finally, we could apply Monte Carlo methods to other variants of graph coloring, like for example Weighted Vertex Coloring or Minimum Sum Coloring, since only the evaluation function would change.

Acknowledgment

This work was supported in part by the French government under management of Agence Nationale de la Recherche as part of the “Investissements d’avenir” program, reference ANR19-P3IA-0001 (PRAIRIE 3IA Institute).

References

1. A. Björklund, T. Husfeldt, and M. Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009.
2. D. Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.
3. C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, Mar. 2012.
4. L. Cai. Parameterized complexity of vertex colouring. *Discrete Applied Mathematics*, 127(3):415–429, 2003.
5. T. Cazenave. Nested Monte-Carlo Search. In C. Boutilier, editor, *IJCAI*, pages 456–461, 2009.
6. T. Cazenave. Nested rollout policy adaptation with selective policies. In T. Cazenave, M. H. M. Winands, S. Edelkamp, S. Schiffel, M. Thielscher, and J. Togelius, editors, *Computer Games - 5th Workshop on Computer Games, CGW 2016, and 5th Workshop on General Intelligence in Game-Playing Agents, GIGA 2016, Held in Conjunction with the 25th International Conference on Artificial Intelligence, IJCAI 2016, New York City, NY, USA, July 9-10, 2016, Revised Selected Papers*, volume 705 of *Communications in Computer and Information Science*, pages 44–56, 2016.

7. T. Cazenave and F. 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, Paris, France, January 16-20, 2012, Revised Selected Papers*, pages 42–54, 2012.
8. E. D. Demaine, M. T. Hajiaghayi, and K. Kawarabayashi. Algorithmic graph minor theory: Decomposition, approximation, and coloring. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 637–646. IEEE Computer Society, 2005.
9. S. Edelkamp, E. Externest, S. Kühl, and S. Kuske. Solving graph optimization problems in a framework for Monte-Carlo search. In *Tenth Annual Symposium on Combinatorial Search*, 2017.
10. S. Edelkamp, M. Gath, T. Cazenave, and F. Teytaud. Algorithm and knowledge engineering for the TSPTW problem. In *Computational Intelligence in Scheduling (SCIS), 2013 IEEE Symposium on*, pages 44–51. IEEE, 2013.
11. S. Edelkamp, M. Gath, C. Greulich, M. Humann, O. Herzog, and M. Lawo. Monte-Carlo tree search for logistics. In *Commercial Transport*, pages 427–440. Springer International Publishing, 2016.
12. S. Edelkamp, M. Gath, and M. 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.
13. S. Edelkamp and C. Greulich. Solving physical traveling salesman problems with policy adaptation. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8. IEEE, 2014.
14. S. Edelkamp and Z. Tang. Monte-Carlo tree search for the multiple sequence alignment problem. In *Eighth Annual Symposium on Combinatorial Search*, 2015.
15. N. Eén and N. Sörensson. An Extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.
16. C. Fleurent and J. A. Ferland. Genetic and hybrid algorithms for graph coloring. *Annals of Operations Research*, 63(3):437–461, 1996.
17. P. Galinier and J.-K. Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of combinatorial optimization*, 3(4):379–397, 1999.
18. S. Gualandi and M. Chiarandini. Graph coloring benchmarks. https://sites.google.com/site/graphcoloring/vertex-coloring. Accessed: 2019-11-17.
19. A. Hertz and D. de Werra. Using tabu search techniques for graph coloring. *Computing*, 39(4):345–351, 1987.
20. A. Hertz, M. Plumettaz, and N. Zufferey. Variable space search for graph coloring. *Discrete Applied Mathematics*, 156(13):2551–2560, 2008.
21. A. Ignatiev, A. Morgado, and J. Marques-Silva. Cardinality encodings for graph optimization problems. In C. Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 652–658. ijcai.org, 2017.
22. B. M. P. Jansen and S. Kratsch. Data reduction for graph coloring problems. *Inf. Comput.*, 231:70–88, 2013.
23. M. Lampis. Finer tight bounds for coloring on clique-width. In I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 86:1–86:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

24. E. Malaguti and P. Toth. A survey on vertex coloring problems. *ITOR*, 17(1):1–34, 2010.
25. D. Marx. Parameterized coloring problems on chordal graphs. *Theor. Comput. Sci.*, 351(3):407–424, 2006.
26. N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.
27. L. Moalic and A. Gondran. Variations on memetic algorithms for graph coloring problems. *Journal of Heuristics*, 24(1):1–24, 2018.
28. B. Negrevergne and T. Cazenave. Distributed nested rollout policy for samegame. In *Workshop on Computer Games*, pages 108–120. Springer, 2017.
29. C. D. Rosin. Nested rollout policy adaptation for Monte Carlo Tree Search. In *IJCAI*, pages 649–654, 2011.
30. D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. *Theory of Computing*, 3(1):103–128, 2007.