Efficiency issues in the RLF heuristic for graph coloring

Marco Chiarandini[1], Giulia Galbiati[2], Stefano Gualandi[3]

[1] IMADA, University of Southern Denmark
Campusvej 55, DK-5000, Odense, Denmark
marco@imada.sdu.dk

[2] Dipartimento di di Informatica e Sistemistica, Università degli Studi di Pavia

[3] Dipartimento di Matematica, Università degli Studi di Pavia
via Ferrata 1, Pavia, I-27100
stefano.gualandi@unipv.it,giulia.galbiati@unipv.it

Abstract

This paper presents an efficient implementation of the well-known Recursive Largest First (RLF) algorithm of Leighton to find heuristic solutions to the classical graph coloring problem. The main features under study are a *lazy* computation of the induced vertex degree and the use of efficient data structures. Computational experiments show that the *lazy* feature leads to a novel implementation that is faster than previous implementations on graphs with high density. Cache-misses are instead determinant for the assessment of data structures.

1 Introduction

The graph coloring problem (GCP) asks to color the n vertices of a given undirected graph $G = (V, E)$, that is, to map each vertex in V to a color in a given set of available colors K, in such a way that adjacent vertices take different colors. The set of available colors is commonly mapped to the set of integers $\{1, \ldots, |K|\}$. In the decision version of the problem, also called the *(vertex) k-coloring problem*, we are asked whether for some given $k = |K|$ a k-coloring exists. In the optimization version, we are asked for the smallest number k, called the *chromatic number* $\chi(G)$, for which a k-coloring exists. In general, the k-coloring problem is NP-complete [9]. For the optimization version, bad results exist also in terms of approximability, for example, a ratios of $n^{1-\epsilon}$ cannot be achieved in polynomial time unless ZPP=NP [8].

A number of polynomial time constructive algorithms to solve heuristically the graph coloring problem have been proposed in the literature. These algorithms do not exhibit guaranteed approximation ratios but they are very fast and they produce good solutions in practice. These features make them very appealing in practical applications. The simplest constructive algorithm orders the vertices of the graph by their degree, and sequentially color each vertex following the fixed order and using the smallest feasible color. A second well-known algorithm is DSATUR [2] that uses a dynamic order of the vertices, instead of a static precomputed order. The idea is again to sequentially color the vertices with the smallest color, but the order is based on a more elaborated idea: the next vertex is the one with the highest *saturation degree*, that is, it has the highest degree induced by the colored vertices (ties are broken by the original degree, in non increasing order). A third constructive algorithm is the so-called Recursive Largest First (RLF) algorithm, introduced in [10]. This algorithm has a different strategy: it sequentially colors stable sets, that is, it sequentially builds sets of vertices that can take the same color. Computational studies on these algorithms show that RLF clearly outperforms the other two in terms of quality on a wide range of graph classes [5]. However, RLF comes with a higher computational cost, having a $O(n^3)$ worst-case complexity, instead of $O(n^2)$ of the other two heuristics.

In spite of its best results in terms of quality, the RLF heuristic has been seldom used in the literature on metaheuristics methods for the GCP, both as a reference method and as a building block for metaheuristic algorithms (see [4] for a list of references).

In this paper we study efficient implementations of this algorithm. We put forward LAZY RLF, a variant of RLF that exploits possible savings in the computations during the execution of the algorithm.

In addition, we study different data structures to represent the graph. A intensive computational campaign indicates that our new implementation performs faster than the original implementation proposed by Leighton on dense graphs. We make our thoroughly tested and characterized C++ implementation publicly available to foster its use in future research and applications.

The paper is organized as follows. Section 2 formally introduces RLF. Section 3 presents in details LAZY RLF. Section 4 discusses the issues related with the data structures. Finally, Section 5 provides a detailed report on our computational experience.

2 Recursive Largest First

The RLF algorithm was introduced in [10] as an algorithm to solve large scale graph coloring problems originated by university exam scheduling. An appendix of the paper contained an implementation of the algorithm in the PC-1 computer language, that can be easily translated into the ANSI C/C++ computer language. The implementation has a worst-case complexity of $O(n^3)$.

Algorithms 1 and 2 show the pseudo code of RLF. We present RLF in a slightly different manner from the original paper in order to simplify the exposition of our implementation. Algorithm 1 is an iterative extraction of stable sets from the (reduced) graph $G = (V, E)$. The core ideas are given in Algorithm 2 that details the FINDSTABLESET procedure.

Algorithm 1 RECURSIVE LARGEST FIRST(G)

In $G = (V, E)$: input graph
Out k : upper bound on $\chi(G)$
Out c : a coloring $c : V \mapsto K$ of G

1: $k \leftarrow 0$
2: **while** $|V| > 0$ **do**
3: $k \leftarrow k + 1$ ▷ Use an additional color
4: FINDSTABLESET(V, E, k) ▷ $G = (V, E)$ is reduced
5: **end while**
6: **return** k

Let $G[X]$ be the subgraph of G induced by the set of vertices X, i.e., $G[X] = (X, \{uv \in E(G) | u, v \in X\})$. Denote by $\delta_X(v)$ the set of vertices adjacent to v in $G[X \cup v]$. Let $d_X(v) = |\delta_X(v)|$ be the degree of v induced by X. Further, let P be the set of uncolored vertices and U the set of vertices that cannot be selected for becoming part of the current stable set. Initially, P is set equal to V, the set of vertices in the reduced graph passed to the FINDSTABLESET procedure, U is empty and the degree induced by U for all vertices is equal to 0 (Line 2). The procedure successively selects a vertex v to be colored (Lines 4 and 5), moves its neighbors $\delta_P(v)$ from P to U (Line 8), reduces the input graph G (Lines 6 and 9), and updates the degree induced by U of every vertex adjacent to a vertex that has moved from P to U.

2.1 Worst-Case Complexity

The worst-case complexity of RLF is $O(n^3)$, as proved in [10]. The FINDSTABLESET procedure is executed k times, but we can exploit the fact that each vertex is selected only once, and then removed, along with its incident edges. Each edge is removed only once giving an overall complexity of $O(m)$ for the edge removals. The highest computational cost is given by the nested loop at lines 10-12: in the worst-case it yields a complexity of $O(n^2)$, or, using the maximum vertex degree Δ in G, $O(\Delta^2)$. The nested loop is executed only once for each vertex, since it can be selected only once at line 4, yielding a complexity of $O(n\Delta^2)$. All the other operations can be implemented in constant time, apart from lines 2 and 4 that have complexity $O(n)$. Therefore the overall complexity is $O(m + n\Delta^2)$, that can be as high as $O(m + n^3)$ in accordance with [10].

Algorithm 2 - procedure FINDSTABLESET(G, k)

```
In G = (V, E) : input graph (in output G is the reduced graph)
In k : color for current stable set
Var P : set of potential vertices for the stable set
Var U : set of vertices not in the current stable set

 1: P ← V, U ← ∅
 2: for all v ∈ P do d_U(v) ← 0
 3: while |P| > 0 do
 4:    v ← argmax_{w∈P} d_U(w)                          ▷ vertex with max degree induced by U
 5:    c(v) ← k                                          ▷ vertex v takes color k
 6:    V ← V \ {v}
 7:    for all w ∈ δ(v) do
 8:       if w ∈ P then   P ← P \ {w},  U ← U ∪ {w} end if    ▷ move w from P to U
 9:       E ← E \ {v, w}                                 ▷ remove v from δ(w)
10:       for all u ∈ δ(w) do
11:          if u ∈ P then d_U(u) ← d_U(u) + 1
12:       end for
13:    end for
14: end while
```

3 Lazy Recursive Largest First

RLF relies on the choice of the vertex with maximum degree induced by U. To select such vertex, it updates an array every time a vertex is selected, visiting all its neighbors. If the induced degrees for each vertex are correctly maintained, the selection of the vertex is linear in n.

We put forward a new strategy to select the vertex v of maximum $d_U(v)$. Instead of maintaining an array of vertex degrees and modifying its values at each vertex selection, we compute $d_U(v)$ *lazily* by exploiting the following proposition. Let $d_U^{max} = \max\{d_U(v)|v \in P\}$ and $\bar{d}_U$ denote the greatest degree induced by U found so far in the current call of FINDSTABLESET.

Proposition 1. *A vertex v with $d_V(v) < \bar{d}_U$ has $d_U(v) < d_U^{max}$.*

Proof. In the reduced graph $G(V, E)$, that is the graph induced by the set of yet uncolored vertices, we have that $\delta_V(v) = \delta_P(v) \cup \delta_U(v)$, and therefore $d_V(v) = d_P(v) + d_U(v)$. Thus, if $d_V(v) < \bar{d}_U$, then $d_U(v) \le d_V(v) < \bar{d}_U \le d_U^{max}$. □

As a practical consequence, once we have found a vertex with $\bar{d}_U$, we can skip the computation of d_U for all the vertices with $d_V(v) < \bar{d}_U$. In addition, to further exploit the proposition, instead of computing $d_U(v)$ starting from 0 and increasing its value every time one of its neighbors belongs to U (METHOD A below), we can start from $d_V(v)$ and decrease its value every time one of its neighbors belongs to the set P (i.e., it is not in U) (METHOD B below). In this way, whenever $d_U(v) < \bar{d}_U$, we perform at most $d_V(v) - \bar{d}_U$ operations instead of $d_V(v)$. Clearly, the larger is the value $\bar{d}_U$, the greater is the saving. Further, if we start with the vertex with maximum degree in G, we have good chances that it also has a high value of $\bar{d}_U(v)$.

METHOD A:

```
d_U(v) ← 0
for all w ∈ δ_V(v) do
   if w ∈ U then
      d_U(v) = d_U(v) + 1
return d_U(v)
```

METHOD B:

```
d_U(v) ← d_V(v)
for all w ∈ δ_V(v) do
   if w ∈ P then
      d_U(v) = d_U(v) − 1
      if d_U(v) < d̄_U then return 0
return d_U(v)
```

In the light of these observations we propose in Algorithm 3 a new version of the procedure FIND-STABLESET. It drops lines 2, and 10-12 of the Algorithm 2, and it modifies line 4 by introducing the LAZYSELECTVERTEX procedure. Algorithm 4 sketches LAZYSELECTVERTEX that exploits the consequences of Proposition 1 to compute the vertex with maximum degree induced by U (ties are first broken in favor of the vertex with smallest degree in G, and then are broken randomly).

Algorithm 3 - procedure LAZYFINDSTABLESET(G, k)

In $G = (V, E)$: input graph (in output G is the reduced graph)
In k : color for current stable set
Var P : set of potential vertices for the stable set
Var U : set of vertices not in the current stable set

```
1:  P ← V, U ← ∅
2:  while |P| > 0 do
3:      v ← LAZYSELECTVERTEX(G, P, U)
4:      c(v) ← k                                ▷ vertex v takes color k
5:      V ← V \ {v}
6:      for all w ∈ δ_P(v) do
7:          P ← P \ {w},  U ← U ∪ {w}           ▷ move w from P to U
8:          E ← E \ {v, w}                      ▷ remove v from δ(w)
9:      end for
10: end while
```

3.1 Worst-Case Complexity

The procedure LAZYFINDSTABLESET is executed k times, but overall, each vertex is selected once and every edge is removed once. The edge removals implies a complexity of $O(m)$. The vertex selection complexity depends on the complexity of the LAZYSELECTVERTEX procedure. Line 1 costs $O(n)$, but it is only used as an heuristic to increase the possibility to start with a high value of $\bar{d}_U$, and therefore can be omitted. Lines 3–4 cost $O(\Delta)$. This gives $O(n\Delta)$, since each vertex is selected once, but the time is dominated by the next loop.

The loop 5–17 is executed $O(n)$ times, and due to the nested loop 8–13 has a worst-case complexity of $O(n\Delta)$, yielding $O(n^2\Delta)$ in total for this loop. Overall the worst-case complexity is $O(m + n^2\Delta)$. However, due the conditional tests at lines 6 and 11, we completely execute the loop 8–13 only very few times.

4 Data Structures

In both RLF and LAZY RLF the graph G must be dynamically updated after each reduction. It is therefore important to choose a graph representation that supports these operations efficiently. (For a review on graph representations see, e.g., Chap. 8 in [11].)

We use an *adjacency list* graph representation, since it allows vertex and edge removals in constant time. The vertex set is stored in a list where each element has four fields: color, degree, membership in P, and a pointer to the adjacency list of the vertex. The adjacency list that contains the neighbors of a vertex i (i.e., indirectly its incident edges) has two fields: a pointer to the position of each neighbor j in the vertex list, and a pointer to the element corresponding to the vertex i in the adjacency list of the neighbor j. These two pointers allow for constant time edge removals. In order to minimize the chances of cache-misses we use an array-based implementation of linked lists, as recommended in [1], instead of a standard list implementation (e.g., the class list of the Standard Template Library in C++). Cache-misses are reduced because the arrays are allocated in contiguous areas of memory [3].

Algorithm 4 - function LAZYSELECTVERTEX(G, P, U)

In $G = (V, E)$: input graph
Var P : set of potential vertices for the stable set
Var U : set of vertices not in the current stable set
Out v : vertex with max degree induced by U

1: $v \leftarrow \text{argmax}_{v \in P} d_V(v)$ ▷ vertex with max degree in G
2: $\bar{d}_U \leftarrow 0$ ▷ compute $d_U(v)$ from scratch
3: **for all** $w \in \delta_V(v)$ **do**
4: **if** $w \in U$ **then** $\bar{d}_U = \bar{d}_U + 1$
5: **for all** $w \in P \setminus \{v\}$ **do**
6: **if** $d_V(w) \geq \bar{d}_U$ **then** ▷ skip all the vertices with $d_V < \bar{d}_U$
7: $d_U(w) \leftarrow d_V(w)$
8: **for all** $u \in \delta_V(w)$ **do**
9: **if** $u \in P$ **then**
10: $d_U(w) \leftarrow d_U(w) - 1$
11: **if** $d_U(w) < \bar{d}_U$ **continue from 5** ▷ skip as soon as $d_V < \bar{d}_U$
12: **end if**
13: **end for**
14: **if** $d_U(w) = \bar{d}_U$ and $d_V(w) \geq d_V(v)$ **continue from 5**
15: $v \leftarrow w,\ \bar{d}_U \leftarrow d_U(v)$
16: **end if**
17: **end for**
18: **return** v

Differently, the implementation described in the original Leighton's paper [10] uses an *adjacency array* graph representation in order to minimize the memory consumption. However, since with *adjacency array* graph updates cannot be done in constant time, Leighton's implementation does not dynamically reduce the graph, instead it traverses the whole adjacency array.

5 Experimental Results

We compare three implementations of the RLF algorithm:

LEI : is the original Leighton's implementation ported in C++, i.e., it corresponds to use an adjacency array for representing the graph.

RLF : implements procedure 2 and array-based lists for representing the graph.

LAZY RLF : implements procedure 3 and array-based lists for representing the graph.

The last two versions have been implemented in ANSI C++ using the same data structures and the same procedures, in order to fairly compare Algorithm 2 and 3. All the three implementations produce solutions of exactly the same quality. Since our interest is in their efficiency, we compare execution time and memory consumption.

Experimental settings. The computational experiments are carried out on a standard desktop machine with an Intel(R) Xeon(R) 2.66GHz CPU, 8Gb of RAM, 64KB of cache L1, 4096 KB of cache L2 and running the linux Ubuntu 10.04.1 operating system. The programs are compiled with the gnu g++ compiler version 4.4.3 and optimization flag -O3.

We generated two sets of random graphs controlling the number of vertices n and the edge density p.

Uniform Random Graphs: These graphs are generated according the $G(n, p)$ Erdös-Rényi model (see, e.g., [7]).

Weight Biased Graphs: These graphs were proposed in [6] where they were shown to be among the hardest to color. They are designed to avoid large cliques and at the same time to favor flatness criteria such as small variations in vertex degree and hidden color classes. We used Culberson's generator to produce these graphs specifying the parameters α and γ to be 0 and 1 respectively.

We generated 5 graphs at each combination of $n = \{1000, 2000, 4000, 8000\}$, $p = \{0.1, 0.2, \ldots, 0.9\}$ and type of graph. Weight biased graphs of size 8000 resulted computationally too costly to be generated and were excluded.

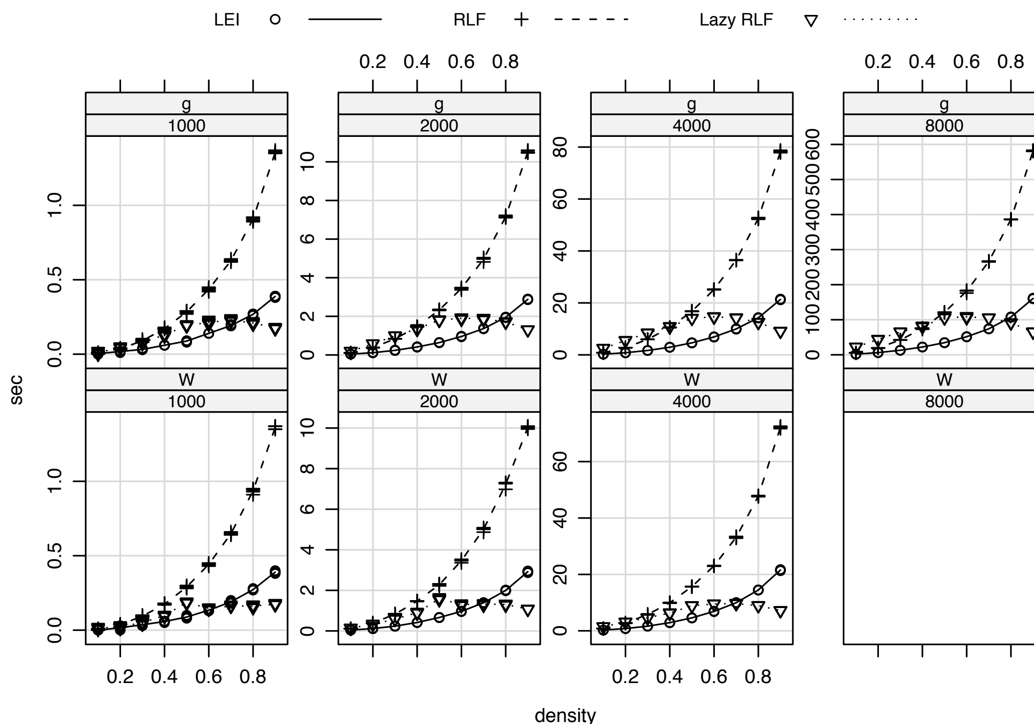

Figure 1: Execution time vs density subdivided per graph type and number of vertices.

Comparison between implementations. Figure 1 shows execution times in seconds plotted against the graph densities[1] for the three implementations. The execution time of both LEI and RLF grows with both the number of vertices and the graph densities, i.e., the number of edges. On the contrary, the execution time of LAZY RLF grows with the graph density, but only up to density 0.6, after which it starts decreasing. The comparison between LAZY RLF and RLF, which both use the same data structures, makes it possible a fair assessment of the *lazy* feature. The fact that the two corresponding curves intersect indicates that the comparison depends on a density threshold. However, the improvement provided by the *lazy* feature for densities greater than the threshold is remarkable, while for densities less than the threshold the extra cost of LAZY RLF is very contained. The threshold seems slightly to vary between uniform graphs and weight biased graphs, being around 0.4 for the former and 0.3 for the latter.

Figure 2 presents the same data of Figure 1 from another perspective: it shows execution time divided by the cube of the graph size against the graph density. The main observation is that the curves for different instance size overlap in these plots. This fact suggests that the real complexity of LEI and RLF is close to its worst-case complexity, that is, $O(n^3)$ independently on its density. It also emphasizes a

[1] Recall that the graph density of a graph with n vertices and m edges is computed as $\frac{2m}{n(n-1)}$.

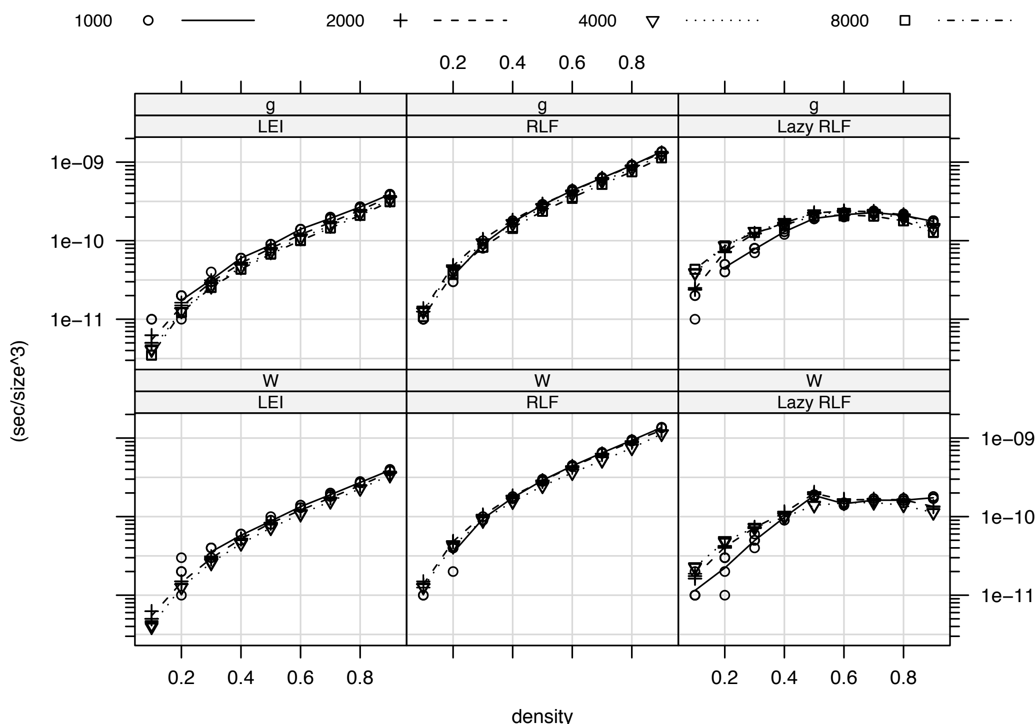

Figure 2: Execution Time in sec/n^3 vs density.

		Instructions	Cache Accesses	L1 cache-misses	L2 cache-misses
g-2000-0.2	LEI	618.52	354.13	5.03	1.85
	RLF	404.20	216.29	33.13	13.91
	LAZY RLF	690.77	279.85	57.46	17.11
g-2000-0.5	LEI	2671.01	1630.70	18.86	10.33
	RLF	1393.54	739.73	197.11	85.28
	LAZY RLF	1660.97	675.39	163.23	64.55
g-2000-0.8	LEI	7528.56	4692.46	48.68	37.56
	RLF	3396.67	1771.08	581.45	266.61
	LAZY RLF	1852.92	839.43	140.10	61.29

Table 1: Results on running the `valgrind` cache profiler on the three algorithms. All the numbers are divided by 10^6.

clear dependency on the graph density of the real complexity of all implementations, with LAZY RLF that seems to have a maximum in complexity close to density 0.6.

Figure 3 shows the memory consumption of LEI and RLF. As expected, the adjacency array data structure leads to better memory performance than array-based lists, since it has to store less pointers.

An accurate profiling of the three algorithms reveals that cache-misses in L1 and L2 memory are crucial for the efficiency of these data structures. Table 1 reports the results of running the `valgrind` profiler on all the three algorithms on three different instances. Each row gives the number of CPU instructions, the number of cache accesses, the number of cache-misses to L1, and the number of cache-misses to L2 (all numbers are divided by 10^6). Comparing the number of CPU instructions and cache accesses of LEI and RLF, it is evident that the dynamic graph data structure is beneficial. The worse performance of RLF in execution comes therefore unexpected. Its explanation lays all in the number of cache-misses to L1 and L2. The increase in this number cancels out the improvement in the number of basic operations and slows down significantly the overall execution. The decrease of basic operations is more pronounced with LAZY RLF, which, on dense graphs, results to be the fastest algorithm.

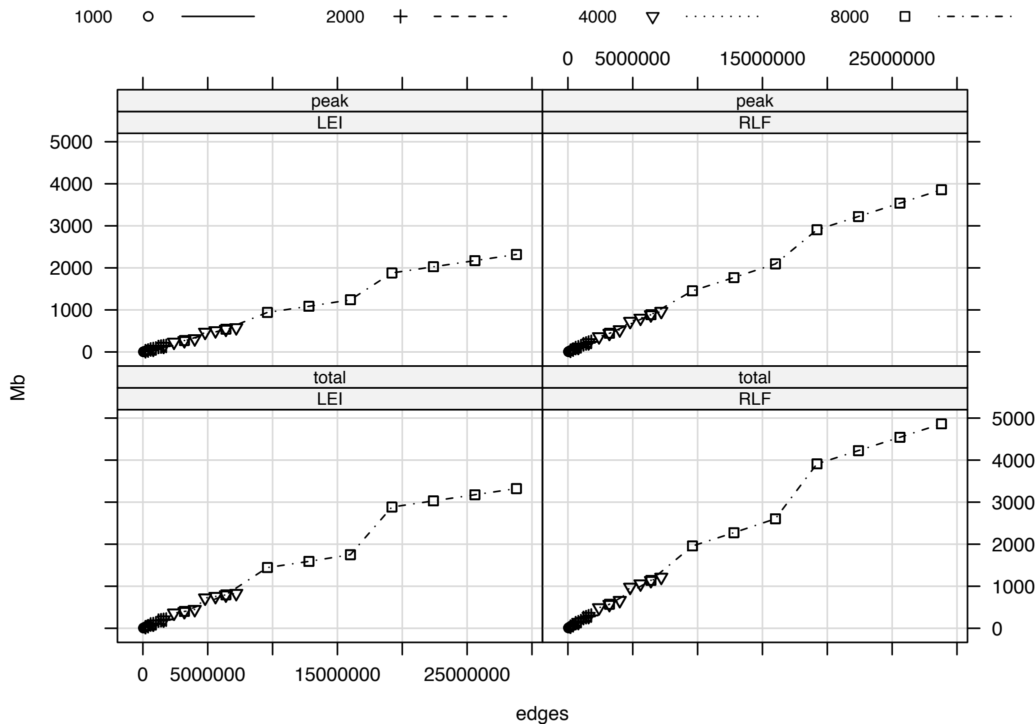

Figure 3: Memory consumption for LEI and RLF.

An adaptive mechanism. The previous results suggest that the use of LAZYSELECTVERTEX from Algorithm 4 pays off only for density larger than a certain value close to 0.4. The best choice seem to depend from the density of the graph left to color. Therefore, we implement a fourth version of Algorithm 1 that adaptively chooses between Algorithm 2 and Algorithm 3 in order to extract the next stable set. The decision between the two is regulated by a threshold t: If the residual graph has density smaller than t then the procedure of Algorithm 2 is chosen, otherwise Algorithm 3 is used.

We experimented with values of t ranging from 0 to 1 at steps of 0.1. Note that $t = 0$ corresponds to RLF and $t = 1$ to LAZY RLF. We report the results in the semilogarithmic panels of Figure 4. In each panel we repeat the two limits given by $t = 0$ and $t = 1$. We can conclude that values of $t = \{0.3, 0.4\}$ for uniform graphs and $t = \{0.1, 0.2\}$ for weight biased graphs, lead to an adaptive implementation that performs best over all the spectrum of possible densities. Closer investigations unveil however that the gain is not large. With respect to LAZY RLF, indeed, the better performance occurs on the small densities where the extra cost of LAZY RLF is not large. Moreover, we have to account for a small extra cost of computing the density of the residual graph before deciding which procedure to use.

6 Conclusions

In this paper, we focused on efficiency issues in the implementations of the RLF heuristic algorithm for graph coloring. We observed that, under some conditions, the selection of the next vertex to include in the stable set can be done faster than in the original implementation described by Leighton. A careful analysis allowed us to design an adaptive implementation that exploits the benefits of both the original and the novel *lazy* computation. In addition, we analyzed two different data structures for the storage of the graph. In our experiments, an adjacency array representation performed better than an adjacency array-based list. This result is somehow surprising and certainly due to modern computer architectures, where the size of the L1 and L2 caches has a direct impact on the overall speed of algorithms due the number of cache-misses. As future work, we plan to design a *cache-friendly* version of LAZY RLF,

that, hopefully, will reduce the number of cache-misses, and it will better exploit the advantages of lazily computing the induced vertex degree.

We make publicly available a C++ implementation that is the result of the engineering process here presented (visit the website given in [4]).

References

[1] J. Black, C. Martel, and H. Qi. Graph and hashing algorithms for modern architectures: Design and performance. In K. Mehlhorn, editor, *Proceedings of 2nd International Workshop on Algorithm Engineering, WAE '98*, pages 37–48. Max-Planck-Institut für Informatik, 1998.

[2] D. Brelaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, 1979.

[3] D. Bulka and D. Mayhew. *Efficient C++: performance programming techniques*. Addison-Wesley, 1999.

[4] M. Chiarandini and S. Gualandi. Bibliography on graph-vertex coloring, 2010. `http://www.imada.sdu.dk/~marco/gcp`.

[5] M. Chiarandini and T. Stützle. An analysis of heuristics for vertex colouring. In P. Festa, editor, *Experimental Algorithms, Proceedings of the 9th International Symposium, (SEA 2010)*, volume 6049 of *Lecture Notes in Computer Science*, pages 326–337. Springer, May 2010.

[6] J. Culberson, A. Beacham, and D. Papp. Hiding our colors. In *Proceedings of the CP'95 Workshop on Studying and Solving Really Hard Problems*, pages 31–42, Cassis, France, Sept. 1995.

[7] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6(2):290–297, 1959.

[8] U. Feige and J. Kilian. Zero knowledge and the chromatic number. *Journal of Computer and System Sciences*, 57(2):187–199, 1998.

[9] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, USA, 1972.

[10] F. Leighton. A graph coloring algorithm for large scheduling problems. *J. Res. National Bureau of Standards*, 84(6):489–506, 1979.

[11] K. Mehlhorn and P. Sanders. *Algorithms and data structures: The basic toolbox*. Springer-Verlag, 2008.