Alcom-FT Technical Report Series
ALCOMFT-TR-01-28

CHECKCOL: Improved Local Search for Graph Coloring *

Massimiliano Caramia[†] Paolo Dell'Olmo[‡] Giuseppe F. Italiano[§]

Abstract

In this paper we present a novel coloring algorithm based on local search. We analyze its performance, and report several experimental results on DIMACS benchmark graphs. From our experiments, this algorithm looks robust, and yields a substantial speed up on previous algorithms for coloring. Our algorithm improves the best known coloring for four different DIMACS benchmark graphs: namely, `Le450-25c`, `Le450-25d` and `Flat300_28_0` and `Flat1000_76_0`. Furthermore, we have run experiments on a simulator to get insights on its cache consciousness: from these experiments, it appears that the algorithm performs substantially less cache misses than other existing algorithms, and can be tuned in order to get to better performance. The results obtained justify also the fact that the running times of the new algorithm are very competitive for all our data sets.

*Work partially supported the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT) and by the Italian National MURST Project "Algorithms for Large Data Sets: Science and Engineering".

[†]Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Roma, Italy. Email: `caramia@disp.uniroma2.it`

[‡]Dipartimento di Statistica, Università di Roma "La Sapienza", Roma, Italy. Istituto di Analisi dei Sistemi ed Informatica, Roma, Italy. Email: `dellolmo@sunshine.iasi.rm.cnr.it`

[§]Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Roma, Italy. Email: `italiano@info.uniroma2.it`

1 Introduction

Let G be an undirected graph, with vertex (node) set V and edge set E. A *stable set* is a subset $S \subseteq V$ of vertices such that no two of them are adjacent. Given an integer $c > 0$, a *c-coloring* of G is a partition of vertices $V = S_1 \cup S_2 \cup \ldots \cup S_c$ such that each S_i is a stable set. We call each S_i a *color class*. The *chromatic number* of G, denoted by $\chi(G)$, is the smallest c for which there exists a c-coloring of G.

Graph coloring problems are ubiquitous, and have been motivated by several applications, including scheduling [1], timetabling [4, 12], and computer register allocation [23]. Although the wide applications of graph coloring techniques suggest that effective algorithms would be of great interest, a limited number of exact algorithms are presented in literature. This can be motivated by the theoretical and experimental complexity of the problem. In fact, the coloring problem is in general $\mathcal{NP}$-complete [18] and it remains $\mathcal{NP}$-complete also in many particular cases, such as the 3-colorability of a four regular planar graph [11]. It is also well known that approximating the chromatic number of a graph to within any constant factor (or even to within n^ϵ for a sufficiently small ϵ) is $\mathcal{NP}$-complete [28]. On the other side, experimental results show that exact coloring codes are able to solve instances having sizes (e.g., 70 vertices for all graph densities) which are small when compared with those generated by many real world applications.

This motivates the importance of designing good coloring heuristics. Indeed coloring applications tend to generate large graphs, where good solutions are required quickly, and for which exact codes fail. Another issue relates to obtaining good upper bounds. This is important for combinatorial optimization problems (e.g., in branch and bound algorithms), and in particular for coloring: for instance, it is well known that decreasing the upper bound from $\chi(G) + 1$ to $\chi(G)$ results in a sharp 40% reduction in the total number of subproblems that have to be considered [33].

Many classical and meta heuristics (see e.g., [31] for a definition of meta heuristic) have been implemented for coloring. In particular, among the most widely adopted techniques we cite tabu search [21], genetic algorithms [7, 14, 15] and simulated annealing [22]. The running times of these methods can be very high, especially for large graphs, as their main goal seems to be obtaining a high quality solution, at a price of a significant computational effort. For instance, Fleurent and Ferland [15] reported 41.35 hours for their genetic algorithm to color a random graph with 1000 vertices and edge probability equal to 0.5. This is clearly unsatisfiable in many applications, where coloring is used as a subroutine of more complex procedures, and where a solution is required in limited time. Among these applications, we cite scheduling problems, where a set of limited resources has to be allocated among a set of tasks: here, a coloring solution corresponds to a feasible schedule where edges represent constraints among tasks (e.g., see [2, 8, 24]).

From many contacts with users of coloring algorithms, we realized the importance of trading off solution quality and running times in the applications, especially for medium to large graphs. In particular, we were asked by a non-academic partner to engineer some coloring codes in order to speed up their running times without sacrificing too much the quality of the achieved solutions. In this paper, we report our findings in this direction. Among all the techniques employed in the design of coloring heuristics, we concentrate as a first step on tabu search. The reasons for this are mainly its wide applicability, and the fact that it seems simple to implement and to engineer.

Tabu search is similar to local or neighborhood search, and proceeds iteratively from one solution to another until a chosen termination criterion is satisfied. In more detail, each point x in the solution space X has an associated neighborhood $N(x) \in X$, and another point $x' \in N(x)$ can be reached from x by means of an operation called *move*: tabu search attempts to perform a sequence of moves towards the optimum. The set of neighbors generated by tabu search is restricted by the use of additional information, in order to avoid cycling and getting trapped in local optima. In particular, tabu search tries to maintain some kind of history of the states explored during the search. To achieve this goal, and similarly to other heuristics, tabu search stores some attributes into two kinds of memory: *short term memory* and *long term memory*.

Short term memory keeps track of solution attributes that have changed during the recent past, and is also referred to as *recency-based* memory. Its main objective is to prevent generating solutions that were analyzed recently. Selected attributes that occur in solutions recently visited are labeled *tabu-active*: solutions that contain tabu-active elements, or particular combinations of these attributes, become *tabu* and are not explored any further in the short term. In general, recency-based memory is managed by creating one or several tabu lists, which record the tabu-active attributes and implicitly or explicitly identify their status. The long term memory relates to solution attributes that are likely to become relevant in the long term, and can be implemented similarly to short term memory.

In several optimization problems, tabu search has typically a good experimental behavior, and this perhaps accounts for its large popularity. In particular, tabu search has been successfully applied to coloring problems (see e.g. [21, 33]), yielding good results especially when compared to the intrinsic difficulty of this problem. However, by examining carefully experimental data on large graphs, two main causes of inefficiency can be observed. The first is that maintaining tabu lists can require a large amount of space. The second is that algorithms based on tabu search tend to make a large number of moves without necessarily being able to improve the solution. Both issues yields a substantial slow down in the overall running times.

In previous work [6], we proposed a new local search algorithm, called HCD, based on tabu search. In particular, HCD tried to overcome most of the drawbacks of tabu search, while still retaining its substantial level of simplicity. The basic idea behind HCD was to make use of tabu concepts without explicitly representing tabu lists. Instead, a dynamic assignment of priorities to the vertices in the graph performed the same task, avoiding repetitions in subsequent moves of the algorithm. The experimental gain of HCD over tabu search was reported in [6].

This paper describes a further engineering of HCD, geared towards improving the ratio between solution quality and running times. In our quest to improve the running times, we worked especially on what we call a *large portion effect*. Indeed, we noticed that one of the most frequent drawbacks of a coloring algorithm (HCD included) is that it could spend a substantial amount of time in large portions of the graph without been necessarily able to improve the current solution. This is particularly negative, and it turns out into a mistake for which one pays twice. In this case indeed, not only one ends up performing a substantial amount of iterations which do not make any progress, but also spends more time for each such iteration – indeed, accessing large portions of the graph results in increased cache miss rates, and consequently to slower memory references.

To improve the solution quality, we tried to avoid the *coloring similarity effect*. Namely, we noticed that in many cases coloring algorithms spend a significant amount of time to find solutions which share not only the same value but also basically a very similar structure (i.e., partition in color classes) with previous solutions. More precisely, in some cases the algorithms find a “new” solution which is not far from a simple permutation of color classes in a previous solution.

Our new local search, which we call CHECKCOL, is designed so as to minimize the amounts of time spent in large portions of the graph without making any progress in the solution. We make sure that the portion of the graph which is currently examined by the algorithm does not becomes too large, so as to work typically on smaller portions of the graph: in our experiments this gets to faster and better colorings, and reduces substantially the cache miss rate and the running times. In order to minimize the coloring similarity effect, we propose a new and more effective long term memory scheme based on simple vertex priorities, and its integration with short term memory.

We experimented extensively on DIMACS benchmark graphs for coloring, which are available in [9] and compared CHECKCOL to tabu search, HCD and the solutions achieved by other coloring algorithms as well (such as [15, 22, 27, 30]). Although we report the results of our experimental study on coloring, we believe that our techniques are general and can be applied to other problems as well. Our source code (in C) is publicly available at `http://www.info.uniroma2.it/~italiano`.

2 Local Search for Coloring

2.1 TABUCOL: Tabu search for Coloring

TABUCOL [21] is a coloring algorithm based on tabu search. It maintains a partition of vertices of a graph: each block in the partition is assigned a different color, although it is not always guaranteed to be a stable set. In other words, TABUCOL permits coloring violations: namely, two adjacent vertices can be in the same block of the partition throughout the execution of the algorithm. This implies that TABUCOL works with solutions which are not necessarily admissible.

An iteration of TABUCOL consists of generating a sample of neighbors of each given vertex, i.e., different partitions that can be obtained from the current partition by moving one vertex to a different existing block. The set of neighbors generated for each vertex is restricted by a tabu list, which prevents a vertex from moving back into a block to which it belonged on a previous iteration. This helps the algorithm to avoid getting stuck in local optima. At most $O(|V|^2)$ possible partitions to choose from are produced at each iteration. Among these partitions, TABUCOL selects the one with the smallest number of coloring violations. We remark that in practice, TABUCOL does not generate all of the possible neighbor partitions at each iteration: rather, it will keep a lower and upper bound on the number of different partitions to be considered each time. Assume that TABUCOL is working on a partition consisting of k blocks. If it finds a feasible coloring, i.e., each block appears to be a stable set, then TABUCOL updates the best current solution to k and restarts with a new partition of $(k-1)$ blocks, in the quest of an improved $(k-1)$-coloring.

Despite its simplicity, there are some drawbacks related to the performance of TABUCOL. First of all, it tends to make a large number of moves without substantially changing the solution. This is perhaps due to the inherent definition of iteration in TABUCOL, and results in a significant increase in the overall running times. Second, the space required by a tabu list can be very large, as it needs to store for each pair of vertices several pieces of information [20]. This has also effects on the time performance: indeed, in our experience, implementations with high memory consumption tend to be rather slow in practice.

Moreover, as it happens in the case of other combinatorial problems as well, tabu search seems very hard to tune for coloring. Indeed, there are several parameters that need a fine tuning. The first is the stopping criterion, i.e., the maximum number of (unsuccessful) iterations that we wish to carry out after the last improvement has been detected. A second parameter to be tuned is the number (maximum and minimum) of neighbors that we need to generate before switching to a new partition. Another important parameter to tune is the tabu list size, which has several practical implications for TABUCOL. Finally, TABUCOL does not seem very robust. Indeed from the experimental analysis it turns out that TABUCOL is extremely sensitive to the initial coloring: if the initial coloring is not good enough, then the performance of TABUCOL degrades substantially.

2.2 HCD: A Priority Local Search for Coloring

We now briefly sketch the main ideas underlying HCD. We assume that the vertices of the underlying graph are numbered from 1 to $|V|$. HCD consists of four different functions. The first function, *Initial-*

ization, computes a trivial starting solution and then is in charge of initializing some data. In particular, it assigns to each node i a color c_i equal to its number, and a priority p_i equal to its color, i.e., $p_i = c_i$. The initial upper bound on the number of required colors (UB) is set to the highest color used by this initial assignment, i.e., $|V|$. Next, the function *Pull-colors* selects the vertex with highest priority, say v. HCD assigns to v the lowest color c which is compatible with the colors already assigned to neighbors of v, and updates its priority to $p_i = c$. If the attempt of pulling down the color of v fails, i.e., if $c = c_i$, *Pull-colors* considers the next vertex with highest priority. Note that this step can be unsuccessfully iterated at most $|V|$ times (if no color can be modified). If no color can be modified by *Pull-colors*, then the function *Push-colors* attempts to assign to each vertex the highest possible color that does not exceed the current upper bound UB. If the coloring is not changed by *Push-colors* either, then we are likely to be trapped in a local optimum. The function *Pop-colors* tries to escape from this solution by assigning color $(UB + 1)$ to the independent set formed by the vertices having color 1. Along with this new assignment, the priority $p_i = (1/c_i)$ is associated with each vertex $i \in V$, and the function *Pull-colors* is called again.

Few remarks are in order at this point. First of all, the reason for choosing the independent set with color 1 in *Pop-colors* lies in the fact that the corresponding vertices have been updated less recently than the others, and thus their update could result in a perturbation which decreases the current solution. Secondly, the reason for which Pop-colors changes the priorities is the following. If the old priority assignment $p_i = (UB + 1)$ were used, then the subsequent call to *Pull-colors* would have chosen as vertices with the highest priority those in the independent set having color $(UB + 1)$. Each of this color would be pulled to 1, thus returning back to the solution we were trying to escape from.

We note that the assignment of priorities simulates a kind of short term memory, as the one represented by tabu lists in TABUCOL. Indeed, this is implicit in the ordering given by vertex priorities: vertices with very low priorities are "tabu" as they are not likely to be updated unless their color is changed because of a *Push-colors*.

Finally, we remark that the solution found by HCD is dependent from the initial numbering of the vertices: different numberings can give rise to different solutions. This is clearly unacceptable: to obtain a better performance, HCD is typically executed after different node numberings in order to let the initial solution be more effective on HCD. However, and differently from TABUCOL, HCD does not need a good initial solution: the trivial solution itself, consisting of $|V|$ different colors, can provide a starting point.

2.3 CHECKCOL: A Further Reengineering of HCD

As it was already mentioned in Section 1, our work, starting from tabu search methods for coloring, concentrates on achieving a good solution quality and on improving the running times while keeping an acceptable level of simplicity. This is achieved by reengineering HCD so as to reduce the *large portion effect*, i.e., preventing the algorithm to spend unnecessarily too much time in large portions of the graph, and the *coloring similarity effect*, i.e., avoiding the generation of similar solutions.

CHECKCOL retains much of the structure of HCD, with two significant differences. To reduce the *large portion effect*, we tried to minimize the time the algorithm spends in wandering ineffectively in large portions of the graph. The main idea used to accomplish this task is a simple but effective *checkpointing* scheme: the algorithm stops at certain steps, releases basically all of its memory, and starts a new local search after this. If the checkpointing interval is properly tuned, one can notice a significant impact on the cache behavior and running times of the algorithm.

Unfortunately, the checkpointing scheme retains and in some cases even amplifies the drawbacks that are inherent in the *coloring similarity effect*. Namely, when we release all the memory used due to checkpointing, we are also risking to loose some vital information, such as solutions previously explored, or data which could have been helpful in avoiding getting trapped in similar solutions or stuck in local minima. To solve this problem, we propose a new and more effective long term memory scheme, and its integration with short term memory. This is achieved with a careful assignment of priorities after a checkpointing.

We start by analyzing checkpointing in detail. Define an *iteration* as an attempt (not necessarily successful) to change color class to a node. Roughly speaking, we define a checkpoint after a certain number of iterations: after each checkpoint, CHECKCOL restarts again a new local search from the current solution, with a consequent release of memory used. The rationale behind this is the following. Typically, coloring algorithms suffer from memory problems on large graphs (e.g., graphs with thousands of nodes), as there can be many parameters to be saved at each iteration. In this situation, releasing memory once in a while helps in decreasing the space consumption. This has a positive effect on running times as well, as algorithms with high space usage tend to be very slow in practice. Moreover, we noticed in our experiments that a properly tuned checkpoint prevents that the portion of the graph which is currently examined by local search becomes too large, and forces the algorithm to work on smaller portions of the graph.

We experimented with two different types of checkpointing: *constant* and *adaptive*. The first is the simpler: the interval at which checkpointing is performed is constant, i.e., checkpointing is performed every ℓ iterations, with ℓ being a properly chosen constant. The second is a bit more complicated, as the checkpointing interval depends on the number of iterations performed, i.e., it grows as the algorithm performs more iterations. Namely, let ℓ_0 be the initial checkpointing interval, and $\alpha > 1$ be a constant. Let $\ell_i = \alpha \ell_{i-1}$, $i \geq 1$. For the first ℓ_1 iterations (i.e., in the range $(0, \ell_1]$), the algorithm performs checkpointing at interval ℓ_0. For the subsequent $(\ell_2 - \ell_1)$ iterations (i.e., in the range $(\ell_1, \ell_2]$), the algorithm performs checkpointing at interval $\ell_1 > \ell_0$. In general, for iterations in the range $(\ell_i, \ell_{i+1}]$, the algorithm performs checkpointing at interval $\ell_i > \ell_{i-1}$.

Note that the main impact of checkpointing is the release of space during the execution of the algorithm, which has a deep influence on the capability of exploiting long term memory throughout the iterations. Indeed, the more frequent the checkpoint and the consequent memory releases, the less advantage the algorithm can take from long term memory. In particular, adaptive checkpointing releases space at different rates. At the initial stages, when the current solution is likely to be improved without much effort, one can afford the risk of being *short-sighted*: the algorithm can rely on short term memory, and time and space can be saved by reducing substantially its long term memory exploitation. In this case, we make the algorithm work with smaller checkpointing intervals, and release memory more frequently. As the algorithm progresses, improving the solution is likely to become more difficult, and it probably pays off to become more *long-sighted*: we thus increase the impact of long term memory by enlarging the checkpointing intervals, and consequently releasing memory less frequently.

However, as it was previously mentioned, the memory releases caused by (constant or adaptive) checkpointing can also have a negative effect, as they might increase the risk of getting trapped in *coloring similarity* effects. Define a *phase* as the interval between any two successive checkpoints. As the algorithm retains no information of what happened before a checkpoint, it could explore solutions which are exactly the same or very similar to solutions that were generated in previous phases, without being able to notice it. To circumvent this problem, one needs to rely more on long term memory: in particular, one needs some sort of long term memory which is able to bridge different phases by bypassing the checkpoints.

We accomplish this task by means of an appropriate reassignments of priorities to the vertices after each checkpointing. The integration of these two features, i.e., checkpointing and priorities, allows us to start successive local searches which are closely interacting, without increasing the space consumption.

We now define how priorities are reassigned. Define the *update count* of a vertex v as the number of times v changed color class in the current phase (i.e., since the last checkpoint). Roughly speaking, the update count of a vertex measures its active participation in trying to improve the solution during that phase. After a checkpoint and before the update counts are reset, the vertex priorities are set as $(1/update\_count)$. With this assignment, a vertex with a low update count (i.e., which was not very active in the last phase) gets a high priority, and thus there is a better chance to get this vertex involved into a local search in the next phases. This clearly prevents the algorithm to continue its visit on a same subset of vertices. In our experiments, this particular choice of priorities generated a sequence of phases which were interacting closely with each other: in most cases a phase was building on the priorities and solutions offered by the previous phase, and consistently offered better priorities and solutions to the next phase.

Similarly to HCD, CHECKCOL works only with admissible solutions: this implies that the current solution is always a proper coloring. In the applications this is very important, as the algorithm can be stopped at any time and still returns an admissible solution.

$CHECKCOL$
1. $Initialize()$;
2. $Pull-colors()$;

Figure 1: The function CHECKCOL

In order to give a flavor of the simplicity underlying its implementation, we give a low-level description of CHECKCOL in pseudocode. As shown in Figure 1, after some initialization we start by invoking *Pull-colors*. The initialization is carried out as illustrated in Figure 2. We first find an initial upper bound on the chromatic number by using a simple greedy approach, as illustrated in Figure 3. In our experiments, the time needed to compute this initial solution was negligible with respect to the overall running times. For each vertex i, we use two variables, which are both initialized to 0: $update\_count_i$ stores the number of updates of vertex i in the current checkpointing interval, and $last\_update_i$ stores the iteration which last updated the color of vertex i. We also initialize the priority p_i of vertex i to its color c_i. Next, we initialize the checkpointing interval and the final stopping criterion, and sort the vertices by non-increasing priorities as shown in Figure 4. Note that ties are broken by choosing first the vertex having the smallest *last_update*; if two vertices have the same priority and *last_update*, ties are broken arbitrarily. The ordering of the vertices are kept in an array $Order$, such that $Order[k]$ contains the k-th largest priority vertex. Finally, we initialize k, an index to the array $Order$, and *iterations*, which stores the number of iterations performed.

Pull-colors tries to pull down the color of each vertex by assigning to it the lowest admissible color (compatible with its neighbors). It proceeds according to priorities, i.e., it examines the highest priority vertices first. Whenever a vertex color is updated, then all its variables (*update_count* and *last_update*) are modified accordingly. *Pull-colors* ends when one of the following three cases occurs: either the stopping criterion is met (in which case the algorithm terminates), or we have to perform a checkpoint (in which case we start a new phase), or all the vertices have been pulled down (in which case we invoke *Push-colors*). The function is described in Figure 5.

Push-colors tries to push up the color of each vertex as much as possible: this is accomplished by assigning to a vertex the highest admissible color not exceeding the current upper bound UB (see Figure 6). We claim that *Push-colors* accomplishes this task by scanning vertices by non-decreasing colors, starting from vertices in the smallest color class first. Indeed whenever *Push-colors* is invoked,

$Initialize()$
1. $Starting\_upper\_bound()$;
2. For each $i \in V$
    2.1. $update\_count_i = 0$;
    2.2. $last\_update_i = 0$;
    2.3. $p_i = c_i$;
3. Set the *checkpoint* interval;
4. Set the stopping criterion;
5. $Sort\_nodes()$;
6. $k = 1$;
7. $iterations = 0$;

Figure 2: The function $Initialize()$

$Starting\_upper\_bound()$
1. $V' = V$;
2. $UB = 0$;
3. While $V' \neq \emptyset$ do
    3.1. Find a maximal stable set, say S;
    3.2. $V' = V' \setminus S$;
    3.3. $UB = UB + 1$.

Figure 3: The function $Starting\_upper\_bound()$

we have the invariant that for each vertex i, $p_i = c_i$. This is true since *Push-colors* can be invoked by *Pull-colors* only, which assigns $p_i = c_i$ to each vertex i (see Line 4. of Figure 5). Furthermore, before invoking *Push-colors*, *Pull-colors* sort the vertices according to their priorities (see Line 8.1. on Figure 5). Since $p_i = c_i$ for each vertex i, this implies that vertices are sorted by non-increasing colors as well. Noting that *Push-colors* scans the vertices in $Order[k]$ from $k = |V|$ until $k = 1$ (see Line 1. on Figure 6) yields that *Push-colors* scans vertices by non-decreasing colors.

This scanning by non-decreasing colors is important for many reasons. Indeed vertices that are considered first in this type of scanning have a larger distance from UB, and thus are likely to be "pushed up" as closer to UB as possible. This tends to make the smaller color classes empty, leaving room for effective color pulldowns (carried out by subsequent *Pull-colors*) towards better solutions (i.e., upper bounds smaller than UB). Furthermore, in order to avoid that a subsequent *Pull-colors* will scan the vertices with the same ordering, thus risking to come back to a similar solution, we set $p_i = (1/c_i)$ on Line 1.4. This way, vertices that have been analyzed first by *Push-colors* (i.e., vertices which had a small color) are likely to be assigned a very large color and thus a very low priority. A subsequent *Pull-colors* will thus examine them for last.

Finally, we remark that similarly to *Pull-colors*, whenever a vertex color is updated, then all its variables (*update_count* and *last_update*) are modified accordingly. *Push-colors* terminates when one of these three cases occurs: either the stopping criterion is met (in which case the algorithm terminates), or a checkpoint is to be performed (in which case we start a new phase), or all of the vertices have been pushed up. We distinguish two cases here. If the coloring has not been changed by *Push-colors*, there is

$Sort\_nodes()$
1. Order nodes by non-increasing priorities;
2. Break ties according to lowest $last\_update$;
3. Let $Order$ be the array containing sorted nodes,
i.e. $p[Order[k]] \geq p[Order[k+1]]$, for $k = 1, \ldots, \|V\| - 1$.

Figure 4: The function $Sort\_nodes$

$Pull - colors()$
1. Choose the highest priority vertex, i.e., $i = Order[k]$;
2. Assign to i the lowest admissible color c;
3. $iterations = iterations + 1$;
4. Assign $p_i = c$;
5. If $c \neq c_i$ then
6.1. $update\_count_i = update\_count_i + 1$;
6.2. $last\_update_i = iterations$;
6. If the stopping criterion is met then stop;
7. If *iterations*>*checkpoint* then $Start\_New\_Phase()$;
8. If $k = \|V\|$ then
8.1. $Sort\_nodes()$;
8.2. $Push - colors()$;
else
8.3. $k = k + 1$;
8.4. $Pull - colors()$;

Figure 5: The function $Pull - colors$

no point in calling *Pull-colors* again: we invoke Pop-colors on Line 3. Otherwise, there was some progress in *Push-colors*: we first reorder the vertices with $Sort\_nodes$ and then go back to *Pull-colors*.

$Start\_New\_Phase$ handles the checkpointing: each vertex i is assigned a new priority $p_i = (1/update\_count_i)$ and all variables but c_i and p_i are released. Finally, vertices are reordered and *Pull-colors* is invoked (see Figure 7).

The last function to be examined is *Pop-colors*, which is invoked when both *Pull-colors* and *Push-colors* have failed to change the current solution. In order to escape from this solution, differently from (and perhaps more accurately than) HCD, we push to color $(UB + 1)$ the color of vertices that were updated less recently. Note that we can use *last_update* as a direct measure of how recently a vertex has been updated: we assign $(UB + 1)$ to the vertices having the smallest *last_update* (see Line 1.1. on Figure 8). Finally, the priorities are reassigned for each vertex i as $p_i = (1/c_i)$, vertices are sorted according to priorities and *Pull-colors* is started again.

3 Experimental Results

In this section we report the results of our experiments, which were run on a Workstation Digital Alpha Model 21164 at 500MHz, with 256 MB RAM, and an external cache of 8 MB. Our code was written in C,

$Push - colors()$
1. For $k = |V|$ downto 1 do
   1.1. $iterations = iterations + 1$;
   1.2. Let $i = Order[k]$;
   1.3. Let c be the largest color in $[c_i, UB]$ that can be assigned to i. If $c \neq c_i$ then
      1.3.1. Set $c_i = c$;
      1.3.2. $update\_count_i = update\_count_i + 1$;
      1.3.3. $last\_update_i = iterations$;
   1.4. Set $p_i = (1/c_i)$;
   1.5. If *iterations*>*checkpoint* then $Start\_New\_Phase()$;
2. If the stopping criterion is met then stop;
3. If coloring not changed then $Pop - colors()$ else
   3.1. $k = 1$;
   3.2. $Sort\_nodes()$;
   3.3. $Pull - colors()$;

Figure 6: The function $Push - colors$

$Start\_New\_Phase()$
1. Compute the next checkpoint;
2. For each $i \in V$ assign $p_i = (1/update\_count_i)$;
3. Release all variables but c_i and p_i;
4. $k = 1$;
5. $Sort\_nodes()$;
6. $Pull - colors()$;

Figure 7: The function $Start\_New\_Phase$

and compiled with the optimization flag `-O3`. In order to allow meaningful comparisons with programs implemented on other platforms, we report in Table 1 the values of the DIMACS Machine benchmarks obtained on our platform. These benchmarks are available at the DIMACS ftp site [10], and consist of a benchmark program (DFMAX) for the maximum clique problem plus five benchmark graphs (r100.5, r200.5, r300.5, r400.5, r500.5). The knowledge of these data on different platforms allows one to infer information on their relative speeds.

The algorithms have been tested on the set of large benchmark DIMACS graphs (Leighton graphs, Johnson's random graphs DSJC and DSJR, Flat graphs) widely used for heuristics in the literature. We believe that DSJR are of particular importance in this benchmark, as they are known to be difficult for local search algorithms.

Graph	r100.5	r200.5	r300.5	r400.5	r500.5
CPU	0.00	0.07	0.64	4.00	15.46

Table 1: CPU time (user time in seconds) obtained for the DIMACS Machine benchmarks

$Pop - colors()$
1. For all vertices $i \in V$ having smallest $last\_update_i$
1.1. Set $c_i = UB + 1$;
1.2. $update\_count_i = update\_count_i + 1$;
1.3. $iterations = iterations + 1$;
1.4. $last\_update_i = iterations$;
2. If the stopping criterion is met then stop;
3. For each $i \in V$ assign $p_i = (1/c_i)$;
4. $k = 1$;
5. $Sort\_nodes()$;
6. $Pull - colors()$;

Figure 8: The function $Pop - colors$

Our experimental analysis is organized in four different parts. A first series of results, contained mainly in Table 2, focuses on the effect of checkpointing. In this set of experiments, we consider a partial version of CHECKCOL which performs only checkpointing but no reassignment of priorities: we refer to this as CHECKCOL$'$. To assess the merits of checkpointing, we compare the CPU times and the solution quality achieved by CHECKCOL$'$ with those obtained by TABUCOL and HCD.

In a second series of results (Table 3) we analyze the effect of a new priority assignment integrated with the checkpoint mechanism. The third series is devoted to measuring how many updates are carried out in the solution, and the total number of iterations obtained to achieve the best solution (Table 5). The last set of experiments investigates more deeply the running times of CHECKCOL. In particular, we analyze the correlation between the running times and the cache miss rates, reporting in Tables 6– 9 our findings for the three algorithms.

Experimental results on CHECKCOL are divided into three different scenarios, each of which represents a different value of the checkpoint. We denote with 0.2 and 0.5 those related to set a constant checkpoint, respectively at 20% and 50% of the maximum number of iterations. We denote with *adaptive* the implementation of an adaptive checkpoint: more precisely the test reported are performed setting $\ell_0 = 100$ and $\alpha = 10$. We refer the reader to Section 2.3 and to Figures 2 and 7 for the definitions of these parameters.

In all the tables we denote with TABU and HCD the columns respectively related to TABUCOL and HCD. Unless defined otherwise, experiments are performed setting the maximum number of iterations to 10 millions. We remark that this was done only for sake of convenience, as the algorithms do not necessarily need to run for a constant number of iterations.

3.1 Analysis of checkpointing

Table 2 reports our experiments with CHECKCOL$'$, where we implemented checkpointing only. From these experiments, there are few points that might be worth emphasizing.

First of all, we notice that our implementation of adaptive checkpointing was always superior to the best constant checkpointing, with respect to both CPU times and the quality of solutions. We believe that this phenomenon can be explained with the considerations already anticipated in Section 2.3. Namely, the strength of adaptive checkpointing is to release memory at different rates throughout the algorithm execution. In particular, at the initial stages when the current solution is likely to be improved without

Graph	TABU CPU	TABU Sol.	HCD CPU	HCD Sol.	0.2 CPU	0.2 Sol.	0.5 CPU	0.5 Sol.	Adaptive CPU	Adaptive Sol.
Le450-5a	60	7	45	7	122	6	148	6	108	6
Le450-5b	62	8	47	7	118	6	130	6	110	6
Le450-5c	50	5	4	5	2	5	4	5	2	5
Le450-5d	48	5	5	5	2	5	5	5	2	5
Le450-15a	195	17	140	17	550	16	645	16	545	16
Le450-15b	152	17	120	17	857	16	956	16	756	16
Le450-15c	60	22	25	17	758	16	834	16	534	16
Le450-15d	62	22	26	17	899	16	976	16	576	16
Le450-25a	80	25	4	25	3	25	4	25	3	25
Le450-25b	84	25	4	25	3	25	4	25	3	25
Le450-25c	202	27	178	27	165	27	177	27	147	27
Le450-25d	198	28	183	27	180	27	182	27	174	27
DSJC125.1	2	5	1	5	0	5	1	5	0	5
DSJC125.5	136	19	118	19	110	19	112	19	110	19
DSJC125.9	11	45	5	45	4	45	4	45	4	45
DSJC250.1	32	8	30	8	28	8	30	8	28	8
DSJC250.5	1706	30	699	30	600	30	657	30	557	30
DSJC250.9	595	74	190	73	182	73	189	73	182	73
DSJC500.1	5	13	4	13	4	13	4	13	4	13
DSJC500.5	2330	50	1984	50	1800	48	1889	49	1789	48
DSJC500.9	6252	127	2299	127	2187	127	2245	127	2045	127
DSJC1000.1	158	21	149	21	142	21	148	21	142	21
DSJC1000.5	17800	90	7565	86	7219	86	7425	86	7025	84
DSJC1000.9	40674	230	13678	229	12844	229	13245	229	12545	226
DSJR500.1	3	14	1	14	1	14	1	14	1	14
DSJR500.5	486	125	152	125	110	125	137	125	110	125
DSJR500.1c	830	89	214	87	146	87	155	87	145	87
Flat300_20_0	2	21	1	20	0	20	1	20	0	20
Flat300_26_0	10	28	1	26	0	26	1	26	0	26
Flat300_28_0	212	35	15	30	0	28	1	28	0	28
Flat1000_50_0	4	95	1	50	1	50	1	50	1	50
Flat1000_60_0	12	93	1	61	1	61	1	61	1	61
Flat1000_76_0	486	92	20	84	10	77	9	77	9	77

Table 2: CPU times (in seconds) and solutions obtained using checkpointing only

much effort, adaptive checkpointing is *short-sighted*: the algorithm tend to rely more on short term memory as it works with smaller checkpointing intervals. As the algorithm progresses, improving the solution is likely to become much more difficult, and adaptive checkpointing makes the algorithm become more *long-sighted*: the impact of long term memory is increased by enlarging the checkpointing intervals, and consequently releasing memory less frequently.

The second observation that one can infer from these experiments is that checkpointing seems to be very effective in practice. Indeed, it can be observed that for all graphs in the benchmark, CHECKCOL′ achieved either a better solution or the same solution but faster than both HCD and TABUCOL. This was more than what we expected as in our plans the goal of checkpointing was to reduce the running times only. For this reasons, we were surprised to find out that in many cases CHECKCOL′ was even able to improve substantially the solutions found by HCD and TABUCOL with faster CPU times. Only for `Le450-5a`, `Le450-5b`, `Le450-15a`, `Le450-15b`, `Le450-15c` and `Le450-15d` the improved colorings of CHECKCOL′ were obtained at a price of higher running times.

3.2 Analysis on priority assignments

Table 3 reports the results of our experiments with the full implementation of CHECKCOL, including the effects of checkpointing and priorities. We recall that in this case after a checkpointing the priority of each vertex is changed as a function of the updates performed on the vertex itself, and the local search basically starts from scratch guided by the new priorities. Finally, we summarize in Table 4 the results reported in Tables 2-3.

The data contained in Tables 3 and 4 give a clear indication of the great impact of priorities on the solution quality. As it can be seen, the coloring of many graphs is substantially improved, and in many cases this improvement is achieved without any deterioration in the running times (i.e., with respect to CHECKCOL′). Once again, only for some Leighton graphs, such as `Le450-5a`, `Le450-5b`, `Le450-15a`, `Le450-15b`, `Le450-15c`, `Le450-15d`, `Le450-25c` and `Le450-25d`, the improvement in the solution is obtained at a price of more work (i.e., higher running times).

Finally, we remark that CHECKCOL matches the best colorings previously known (to the best of our knowledge most of these solutions are obtained in [15, 22, 27, 30]). Even more, there are few cases (e.g., `Le450-25c`, `Le450-25d`, `Flat300_28_0` and `Flat1000_76_0`) where CHECKCOL improves the best known coloring.

3.3 Analysis on the number of updates and iterations

In these runs of experiments, we measured the total number of updates of the solution and the total number of iterations performed to find the best coloring. Note that the latter is an indication on how fast a particular algorithm finds the best solution. Our results are summarized in Table 5, where we do not include the data for constant checkpointing, as it is inferior to adaptive checkpointing (i.e., constant checkpointing achieves always a lower number of solution updates and a higher number of iterations to obtain the best solution than adaptive checkpointing).

When considering the the total number of solution updates, we had two extremes in our experiments. On one extreme stood Leighton graphs, where CHECKCOL seemed to perform consistently always more solution updates than HCD and TABU. Even when CHECKCOL achieved the same solution as HCD, it performed up to 50% more updates than HCD. This was substantially higher when CHECKCOL improved the coloring.

On the other extreme, CHECKCOL seemed to perform less solution updates than HCD, such as in the case of Flat graphs. Albeit a bit odd, this can be explained by taking into account the other parameter,

Graph	TABU CPU	TABU Sol.	HCD CPU	HCD Sol.	0.2 CPU	0.2 Sol.	0.5 CPU	0.5 Sol.	Adaptive CPU	Adaptive Sol.
Le450-5a	60	7	45	7	122	5	148	5	108	5
Le450-5b	62	8	47	7	118	5	130	5	110	5
Le450-5c	50	5	4	5	2	5	4	5	2	5
Le450-5d	48	5	5	5	2	5	5	5	2	5
Le450-15a	195	17	140	17	2250	15	2345	15	2145	15
Le450-15b	152	17	120	17	2857	15	2956	15	2756	15
Le450-15c	60	22	25	17	4758	15	4834	16	4534	15
Le450-15d	62	22	26	17	4899	15	4976	16	4576	15
Le450-25a	80	25	4	25	3	25	4	25	3	25
Le450-25b	84	25	4	25	3	25	4	25	3	25
Le450-25c	202	27	178	27	3605	25	3777	26	3477	25
Le450-25d	198	28	183	27	4850	25	4924	26	4524	25
DSJC125.1	2	5	1	5	0	5	1	5	0	5
DSJC125.5	136	19	118	19	110	17	112	17	110	17
DSJC125.9	11	45	5	45	4	44	4	44	4	44
DSJC250.1	32	8	30	8	28	8	30	8	28	8
DSJC250.5	1706	30	699	30	600	28	657	30	557	28
DSJC250.9	595	74	190	73	182	72	189	73	182	72
DSJC500.1	5	13	4	13	4	12	4	13	4	12
DSJC500.5	2330	50	1984	50	1800	48	1889	49	1789	48
DSJC500.9	6252	127	2299	127	2187	126	2245	127	2045	126
DSJC1000.1	158	21	149	21	142	21	148	21	142	21
DSJC1000.5	17800	90	7565	86	7219	84	7425	85	7025	84
DSJC1000.9	40674	230	13678	229	12844	226	13245	228	12545	226
DSJR500.1	3	14	1	14	1	12	1	13	1	12
DSJR500.5	486	125	152	125	110	123	137	124	110	120
DSJR500.1c	830	89	214	87	146	87	155	87	145	87
Flat300_20_0	2	21	1	20	0	20	1	20	0	20
Flat300_26_0	10	28	1	26	0	26	1	26	0	26
Flat300_28_0	212	35	15	30	0	28	1	28	0	28
Flat1000_50_0	4	95	1	50	1	50	1	50	1	50
Flat1000_60_0	12	93	1	61	8	60	9	60	8	60
Flat1000_76_0	486	92	20	84	10	76	9	77	9	76

Table 3: CPU times (in seconds) and solutions obtained using checkpointing and priorities

Graph	TABU CPU	TABU Sol.	HCD CPU	HCD Sol.	CHECKCOL′ CPU	CHECKCOL′ Sol.	CHECKCOL CPU	CHECKCOL Sol.
Le450-5a	60	7	45	7	108	6	108	5
Le450-5b	62	8	47	7	110	6	110	5
Le450-5c	50	5	4	5	2	5	2	5
Le450-5d	48	5	5	5	2	5	2	5
Le450-15a	195	17	140	17	545	16	2145	15
Le450-15b	152	17	120	17	756	16	2756	15
Le450-15c	60	22	25	17	534	16	4534	15
Le450-15d	62	22	26	17	576	16	4576	15
Le450-25a	80	25	4	25	3	25	3	25
Le450-25b	84	25	4	25	3	25	3	25
Le450-25c	202	27	178	27	147	27	3477	25
Le450-25d	198	28	183	27	174	27	4524	25
DSJC125.1	2	5	1	5	0	5	0	5
DSJC125.5	136	19	118	19	110	19	110	17
DSJC125.9	11	45	5	45	4	45	4	44
DSJC250.1	32	8	30	8	28	8	28	8
DSJC250.5	1706	30	699	30	557	30	557	28
DSJC250.9	595	74	190	73	182	73	182	72
DSJC500.1	5	13	4	13	4	13	4	12
DSJC500.5	2330	50	1984	50	1789	48	1789	48
DSJC500.9	6252	127	2299	127	2045	127	2045	126
DSJC1000.1	158	21	149	21	142	21	142	21
DSJC1000.5	17800	90	7565	86	7025	84	7025	84
DSJC1000.9	40674	230	13678	229	12545	226	12545	226
DSJR500.1	3	14	1	14	1	14	1	12
DSJR500.5	486	125	152	125	110	125	110	123
DSJR500.1c	830	89	214	87	145	87	145	87
Flat300_20_0	2	21	1	20	0	20	0	20
Flat300_26_0	10	28	1	26	0	26	0	26
Flat300_28_0	212	35	15	30	0	28	0	28
Flat1000_50_0	4	95	1	50	1	50	1	50
Flat1000_60_0	12	93	1	61	1	61	8	60
Flat1000_76_0	486	92	20	84	9	77	9	76

Table 4: Summary of results

Graph	TABU Up.	TABU Iter.	HCD Up.	HCD Iter.	Adaptive Up.	Adaptive Iter.
Le450-5a	307595	907537	555570	1117339	757805	1258252
Le450-5b	388974	987644	675442	1187342	798511	1259391
Le450-5c	3335	8376	6235	12339	6972	9284
Le450-5d	2765	7943	5134	11739	5950	9268
Le450-5a	342344	842384	442997	843339	879223	1116252
Le450-5b	357453	856353	484446	894339	995386	1116465
Le450-15c	353445	853785	443982	953987	3957417	5576928
Le450-15d	392746	896846	483989	963835	3891934	5596043
Le450-25a	1865	4032	2093	3861	2986	3105
Le450-25b	1834	3889	2259	3375	2978	3339
Le450-25c	353369	753489	453284	853569	924931	1126397
Le450-25d	243456	643956	423634	893900	934872	1136275
DSJC125.1	2286	5423	2687	4339	2799	4124
DSJC125.5	143886	348624	156245	289379	162241	268765
DSJC125.9	4256	9000	4281	8539	4911	8443
DSJC250.1	79650	168723	81559	143339	84798	123743
DSJC250.5	1272980	3604000	1729346	3454339	2014528	3343534
DSJC250.9	345969	720674	358243	703339	441280	684165
DSJC500.1	8034	16172	8102	14390	8425	12145
DSJC500.5	1407392	3078935	1422738	2983226	1943029	2904309
DSJC500.9	2078145	4211289	2089322	4113869	2562185	4074145
DSJC1000.1	129293	295448	145998	283239	157192	204205
DSJC1000.5	5469243	9921100	5798011	9083801	5900023	8941390
DSJC1000.9	5592414	9982830	6354955	9809362	7627717	9004109
DSJR500.1	710	1457	731	1437	846	1244
DSJR500.5	1242954	2978432	1422738	2953320	1943029	2927314
DSJR500.1c	528235	1122402	615151	1102869	652296	911259
Flat300_20_0	12893	33534	14364	32142	17934	30344
Flat300_26_0	593857	1823044	403824	625223	62683	172224
Flat300_28_0	4578900	9723645	278422	582993	86425	159234
Flat1000_50_0	823565	1664564	424570	644468	289191	362445
Flat1000_60_0	2780559	5548989	293277	542824	301025	539212
Flat1000_76_0	278299	540132	279144	523924	287355	499825

Table 5: Number of solution updates and number of iterations performed to find the best coloring

Graph	HCD	TABUCOL	0.2	0.5	Adaptive
Le 450-5a	0.24	0.34	0.23	0.23	0.22
Le 450-5b	0.24	0.34	0.23	0.23	0.22
Le 450-5c	0.24	0.35	0.21	0.23	0.18
Le 450-5d	0.24	0.36	0.21	0.23	0.18
Le 450-15a	0.30	0.39	0.28	0.30	0.25
Le 450-15b	0.30	0.39	0.28	0.30	0.25
Le 450-15c	0.30	0.42	0.30	0.30	0.27
Le 450-15d	0.30	0.41	0.30	0.30	0.27
Le 450-25a	0.29	0.41	0.27	0.28	0.25
Le 450-25b	0.29	0.41	0.27	0.28	0.25
Le 450-25c	0.29	0.44	0.26	0.29	0.24
Le 450-25d	0.29	0.44	0.26	0.29	0.24

Table 6: Cache misses of 100,000 iterations on Leighton graphs for different values of checkpoint. The experiments simulate 128KB mapped cache memory

i.e., the number of iterations required to find the best solution. In these cases indeed, CHECKCOL was able to find its best solution within a very limited number of iterations outperforming both TABUCOL and HCD in the solution. This motivates the fact that the number of updates in some cases are lower than the other two algorithms.

From the analysis of the data in Table 5, it turns out that CHECKCOL has always a better ratio between solution updates and iterations to the best than either HCD or TABUCOL. This means higher efficacy of CHECKCOL, as it implies more solution updates and less iterations to find the best coloring. Indeed the best ratio achieved by CHECKCOL is $(updates/iterations) = 0.85$ while those of TABUCOL and HCD are respectively 0.56 and 0.65; the average ratio of $(updates/iterations)$ computed for all data sets gives 0.67 for CHECKCOL and respectively 0.51 and 0.56 for TABUCOL and HCD.

3.4 Analysis on cache miss rates

To get a further insight on the good CPU times obtained by CHECKCOL, we decided to make a further run of experiments by measuring the cache miss rates of the different algorithms. Perhaps the most striking difference between CHECKCOL and TABUCOL is that CHECKCOL needs not to maintain large data structures (such as a tabu list) on the underlying graph: each vertex is simply augmented with few extra bytes that store its number, color and priority. We expected thus that most of the computational gain of CHECKCOL derived from this better memory usage, and particularly from a better cache performance. We ran some experiments with ATOM [34], a toolkit developed by DEC for instrumenting program executables on Alpha workstations, and found out that HCD outperforms significantly TABUCOL in cache efficiency. In particular, we present our results simulating both a 128KB and a 2MB cache. We did not notice much difference between these two cases.

Tables 6 and 7 report our experiments on Leighton graphs [26]: note that HCD has already substantially less cache misses – between 7% and 32% less – than TABUCOL. Similar or superior gains hold for other benchmark graphs as well. In particular, as it is shown in Tables 8 and 9, for Johnson's benchmark graphs [22] HCD has between 17% and 81% less cache misses than TABUCOL. The higher improvement achieved here could be explained by considering that Johnson's random graphs are larger than

Graph	HCD	TABUCOL	0.2	0.5	Adaptive
Le 450-5a	0.23	0.34	0.22	0.23	0.20
Le 450-5b	0.23	0.34	0.22	0.23	0.20
Le 450-5c	0.23	0.35	0.20	0.21	0.17
Le 450-5d	0.23	0.35	0.20	0.21	0.17
Le 450-15a	0.29	0.39	0.25	0.28	0.24
Le 450-15b	0.29	0.39	0.25	0.28	0.24
Le 450-15c	0.29	0.41	0.26	0.28	0.25
Le 450-15d	0.29	0.41	0.26	0.28	0.25
Le 450-25a	0.29	0.41	0.25	0.28	0.24
Le 450-25b	0.29	0.41	0.25	0.28	0.24
Le 450-25c	0.29	0.44	0.24	0.28	0.22
Le 450-25d	0.29	0.44	0.24	0.28	0.22

Table 7: Cache misses of 100,000 iterations on Leighton graphs for different values of checkpoint. The experiments simulate 2MB mapped cache memory

Leighton's graphs, and thus more likely to highlight cache effects. From our experiments, CHECKCOL further improves on HCD: indeed it does between 6% and 35% less cache misses than HCD on benchmark graphs [9]. This perhaps can further motivate its increased efficiency in terms of faster running times and better solutions.

Our experiments on cache misses further supports most of the conclusions that were already drawn for CHECKCOL. For instance, even in the metric of cache misses CHECKCOL was superior to HCD and TABUCOL for all graphs in our benchmark, and adaptive checkpointing was consistently superior to constant checkpointing. However, we learned some new lessons from the cache experiments as well. For instance, we noticed that for constant checkpointing the cache miss rates of CHECKCOL changed significantly with the checkpointing interval. Recall that an *iteration* is defined as an attempt (not necessarily successful) to change color class to a node, and that in constant checkpointing we insert a checkpoint after a fixed number of iterations. We noticed that large values of the checkpoint (e.g., having only one checkpoint at 0.5) yielded similar cache miss rate to HCD. This makes sense as when CHECKCOL performs few checkpoints, it basically degrades to HCD (which performs no checkpoints at all). For decreasing values of the checkpoint, the cache performance of CHECKCOL improved substantially. Indeed, when we perform many checkpoints, the algorithm tends to explore smaller portions of the graph in each phase, and this improves the probability of cache hits during the same phase. We notice, however, that the cache miss ratio tended to increase again at very small values of the checkpoint: the algorithm was probably performing too many phases, and perhaps the cache misses incurred at each switch of phase started to become significant in the overall picture.

4 Conclusions

In this paper we have presented a new local search algorithm for graph coloring. The algorithm is obtained as an engineering of existing tabu search-based algorithms geared towards improving the tradeoffs between solution quality and running times. Our new local search is designed so as to reduce the amount of time spent wandering in large portions of the graph without making any progress in the

Graph	HCD	TABUCOL	0.2	0.5	Adaptive
DSJC125.1	0.05	0.20	0.05	0.05	0.05
DSJC125.5	0.05	0.20	0.05	0.05	0.05
DSJC125.9	0.05	0.26	0.04	0.04	0.04
DSJC250.1	0.08	0.25	0.07	0.07	0.07
DSJC250.5	0.08	0.27	0.08	0.08	0.08
DSJC250.9	0.08	0.30	0.08	0.08	0.07
DSJC500.1	0.32	0.40	0.15	0.18	0.15
DSJC500.5	0.32	0.45	0.15	0.18	0.14
DSJC500.9	0.26	0.50	0.15	0.15	0.15
DSJC1000.1	0.36	0.50	0.24	0.33	0.23
DSJC1000.5	0.34	0.55	0.24	0.30	0.23
DSJC1000.9	0.24	0.55	0.18	0.18	0.17

Table 8: Cache misses of 500,000 iterations on Johnson's random graphs DSJC for different values of checkpoint. The experiments simulate 128KB mapped cache memory

Graph	HCD	TABUCOL	0.2	0.5	Adaptive
DSJC125.1	0.05	0.20	0.05	0.05	0.05
DSJC125.5	0.05	0.20	0.05	0.05	0.05
DSJC125.9	0.05	0.25	0.04	0.04	0.04
DSJC250.1	0.08	0.25	0.07	0.07	0.07
DSJC250.5	0.08	0.27	0.08	0.08	0.08
DSJC250.9	0.08	0.30	0.08	0.08	0.07
DSJC500.1	0.28	0.40	0.09	0.12	0.08
DSJC500.5	0.28	0.45	0.15	0.18	0.14
DSJC500.9	0.25	0.48	0.15	0.18	0.14
DSJC1000.1	0.30	0.50	0.23	0.25	0.23
DSJC1000.5	0.30	0.55	0.23	0.25	0.23
DSJC1000.9	0.23	0.55	0.18	0.18	0.17

Table 9: Cache misses of 500,000 iterations on Johnson's random graphs DSJC for different values of checkpoint. The experiments simulate 2MB mapped cache memory

solution. To accomplish this task we have introduced the notion of checkpointing: the algorithm is forced to stop at certain steps, releases all of its memory, and starts a new local search. Another ingredient of our algorithm is a dynamic assignment of priorities to the vertices. We use these priorities to define a new and more effective long term memory scheme, which is integrated with the short term memory implied by checkpointing. The employment of checkpointing and priorities yields a significant impact on the solution quality, cache consciousness, and running times of the resulting algorithm.

It is quite natural to ask whether the same engineering approach may be applicable to other combinatorial optimization problems as well. In particular, there are some weighted coloring problems (e.g., timetabling and frequency assignments) which seem to be natural candidates for further investigations.

References

[1] Bianco, L., Caramia, M., and Dell'Olmo, P., *Solving a Preemptive Scheduling Problem using Coloring Technique*, in Ed. Weglarz, Project Scheduling Recent Models, Algorithms and Applications, Kluwer Academic Publishers (1998).

[2] Blazewicz, J., Ecker, K. H., Pesch, E., Schmidt, G., and Weglarz, J., *Scheduling Computer and Manufacturing Processes*, Springer (1996).

[3] Brelaz, D., "New methods to color the vertices of a graph", *Communications of the ACM* 22 (1979), 251–256.

[4] Cangalovic, M., and Schreuder, J., "Exact coloring algorithm for weighted graphs applied to timetabling problems with lectures of different lengths", *EJOR* 51 (1991) 248–258.

[5] Caramia, M.: "Coloring Algorithms and Their Applications to Project Scheduling", Ph.D. Thesis in Operations Research, University of Rome "La Sapienza" (2000).

[6] Caramia, M., and Dell'Olmo, P.: "A Fast and Simple Local Search for Graph Coloring", Proc. of the 3-rd Workshop on Algorithm Engineering (WAE'99), Lecture Notes in Computer Science 1618 (1999) 319–333.

[7] Chams, M., Hertz, A., and de Werra, D.: "Some experiments with simulated annealing for coloring graphs", *EJOR* **32** (1987), 260–266.

[8] Coffman, E. G., "An Introduction to Combinatorial Models of Dynamic Storage Allocation", *SIAM Review* 25 (1983), 311–325.

[9] DIMACS ftp site for benchmark graphs:
ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/color/.

[10] DIMACS ftp site for benchmark machines:
ftp://dimacs.rutgers.edu/pub/challenge/graph/benchmarks/volume/machine.

[11] Dailey, D. P., "Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete", *Discrete Mathematics* 30 (1980), 289–293.

[12] De Werra, D., "An introduction to timetabling", *European Journal of Operational Research*, 19 (1985) 151–162.

[13] Dorne, R., and Hao, J.K. "Tabu Search for Graph Coloring, T-colorings and Set T-colorings", in *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization*, Kluwer Academic Publishers, (1999) 77–92.

[14] Eiben, A. E., Van Der Hauw, J. K., and Van Hemert, J. I.: "Graph Coloring with Adaptive Evolutionary Algorithms" *Journal of Heuristics* **4** (1998), 25–46.

[15] Fleurent, C., and Ferland, J. A.: "Genetic and hybrid algorithms for graph coloring", *Annals of Operations Research*, **63** (1995), 437-461.

[16] Fleurent, C., and Ferland, J. A.: "Object-Oriented Implementation of Heuristic Search Methods for Graph Coloring, Maximum Clique and Satisfiability", in D. S. Johnson and M. A. Trick eds. Volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* AMS (1996), 619-652.

[17] Garey, M. R., and Johnson, D. S, "The complexity of near-optimal graph coloring", *Journal of the ACM* 23 (1976), 43–49.

[18] Garey, M. R., Johnson, D. S., and Stockmeyer, L., "Some simplified NP-complete graph problems", *Theor. Comput. Sci.* 1 (1976), 237–267.

[19] Goldberg, D.E., *Genetic Algorithms in Search, Optimization, and Machine Learninig*, Addison-Welsey, (1989).

[20] Glover, F., Laguna, M.: Tabu Search, Kluwer Academic Publishers (1997).

[21] Hertz, A., and de Werra, D.: "Using Tabu Search Techniques for Graph Coloring", *Computing* **39** (1987), 345–351.

[22] Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C.: "Optimization by simulated annealing: An experimental evaluation; Part II, Graph coloring and number partitioning," *Operations Research* **39** (1991), 378-406.

[23] Kannan, S., and Proebsting, T., "Register Allocation in Structured Programs", *Journal of Algorithms* 29 (1998) 223–237.

[24] Krawczyk, H., and Kubale, M., "An approximation algorithm for diagnostic test scheduling in multicomputer systems", *IEEE Trans. Comput.* C-34 (1985), 869–872.

[25] Kubale, M., and Jackowski, B., "A generalized implicit enumeration algorithm for graph coloring", *Communications of the ACM* 28 (1985), 412–418.

[26] Leighton, F. T.: "A graph coloring algorithm for large scheduling problems", *Journal of Research of the National Bureau of Standards* **84** (1979), 412–418.

[27] Lewandowski, G., "Experiments with Parallel Graph Coloring Heuristics", in D. S. Johnson and M. A. Trick eds. Volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* AMS (1996).

[28] Lund, C., and Yannakakis, M., "On the Hardness of Approximating Minimization Problems", Proc. 25th Annual ACM Symp. on Theory of Computing, (1993), 286–293 (Full version in *J. ACM* 41 5 (1994), 960-981.)

[29] Mehrotra, A., and Trick, M. A., "A Column Generation Approach for Graph Coloring", *INFORMS J. on Computing* 8 (1996), 344–354.

[30] Morgenstern, C.: "Distributed Coloration Neighborhood Search", in D. S. Johnson and M. A. Trick eds. Volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* AMS (1996), 335-358.

[31] Osman, I. H., and Kelley, J. P., *Metaheuristics: Theory and Applications*, Kluwer Academic Publishers, Hingam, MA (1996).

[32] Sager, T. J., and Lin S., "A Pruning Procedure for Exact Graph Coloring", *ORSA J. on Computing* 3 (1993), 226–230.

[33] Sewell, E. C., "An Improved Algorithm for Exact Graph Coloring", in D. S. Johnson and M. A. Trick eds. Volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* AMS (1996), 359-373.

[34] Srivastava, A., and Eustace, A.: "ATOM: A system for building customized program analysis tools", Proc. ACM Symp. on Programming languages, Design and Implementation, 1994, 196–205.