INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Reactive Local Search for the Maximum Clique Problem

R. Battiti * M. Protasi †

TR-95-052

September 1995

Abstract

A new Reactive Local Search (*RLS*) algorithm is proposed for the solution of the Maximum-Clique problem. *RLS* is based on local search complemented by a feedback (memory-based) scheme to determine the amount of diversification. The reaction acts on the single parameter that decides the temporary prohibition of selected moves in the neighborhood, in a manner inspired by Tabu Search. The performance obtained in computational tests appears to be significantly better with respect to all algorithms tested at the the second DIMACS implementation challenge. The worst-case complexity per iteration of the algorithm is $O(\max\{n, m\})$ where n and m are the number of nodes and edges of the graph. In practice, when a vertex is moved, the number of operations tends to be proportional to its number of missing edges and therefore the iterations are particularly fast in dense graphs.

Key words: maximum clique problem, heuristic algorithms, tabu search, reactive search.

*Dipartimento di Matematica, Universita' di Trento, Via Sommarive 14, 38050 Povo (Trento) - Italy, battiti@science.unitn.it

†Dipartimento di Matematica, Universita' di Roma "Tor Vergata", Via della ricerca scientifica, 00133 Roma - Italy, protasi@mat.utovrm.it Work partially done while visiting the International Computer Science Institute, Berkeley, Ca

1 Introduction

Maximum Clique (MC for short) is a paradigmatic combinatorial optimization problem with relevant applications and, because of its computational intractability, it has been extensively studied in the last years [19].

Let $G = (V, E)$ be an arbitrary undirected graph, $V = \{1, 2, \ldots, n\}$ its vertex set, $E \subseteq V \times V$ its edge set, and $G(S) = (S, E \cap S \times S)$ *the subgraph induced by* S, where S is a subset of V. A graph $G = (V, E)$ is *complete* if all its vertices are pairwise adjacent, i.e. $\forall i, j \in V$, $(i, j) \in E$. A *clique* K is a subset of V such that $G(K)$ is complete. The Maximum Clique (MC) problem asks for a clique of maximum cardinality.

MC is an NP-hard problem, furthermore strong negative results have been shown about its approximation properties (for a survey on the approximability of NP-hard problems see [1]). In particular, if $P \neq NP$, no polynomial time algorithm can approximate the Maximum Clique problem within a factor $n^{\frac{1}{4}}$, where n is the number of nodes of the graph [8].

These theoretical results stimulated a research effort to design efficient heuristics for this problem. Consequently, computational experiments have been executed to show that the optimal values or close approximate values can be efficiently obtained for significant families of graphs related to practical situations [17, 19].

In this paper a new *reactive* heuristic is proposed for the Maximum Clique problem: Reactive Local Search (*RLS*). *RLS* complements local-neighborhood-search with *prohibition-based diversification* techniques, where the amount of diversification is determined in an automated way through a *feedback* scheme.

Local search is a well known technique that can be very effective in searching for good locally optimal solutions. On the other hand, local search can be trapped in local optima and be unable to reach a global optimum or even good approximate solutions. Many improvements have been proposed, in particular F. Glover's *Tabu Search* [10] (TS for short) has been successfully applied to a growing number of problems, including MC [20, 21]. TS is based on prohibitions: some local moves are temporarily prohibited in order to avoid cycles in the search trajectory and to explore new parts of the total search space.

Although powerful, some algorithmic schemes based on TS are complex and contain many possible choices and parameters, whose appropriate setting is a problem shared by many heuristic techniques [4]. In some cases the parameters are tuned through a *feedback loop* that includes the user as a crucial *learning* component: depending on preliminary tests, some values are changed and different options are tested until acceptable results are obtained. The quality of results is not automatically transferred to different instances and the feedback loop can require a lengthy "trial and error" process.

Reactive schemes aim at obtaining algorithms with an internal feedback (*learning*) loop, so that the tuning is automated. Reactive schemes are therefore based on *memory*: information about past events is collected and used in the future part of the search algorithm. A TS-based reactive scheme (*RTS*) has been introduced in [5]. *RLS* adopts a reactive strategy that is appropriate for the neighborhood structure of MC: the feedback acts on a single parameter (the *prohibition period*) that regulates the search diversification and an explicit memory-influenced restart is activated periodically.

The quality of the experimental results obtained by *RLS* is very satisfactory. Standard

benchmark instances and timing codes for MC have been designed as part of the international Implementation Challenge, organized in 1993 by the Center for Discrete Mathematics and Theoretical Computer Science to study effective optimization and approximation algorithms for Maximum Clique, Graph Coloring, and Satisfiability. Thirtyseven significant MC instances have been selected by the organizers to provide a "snapshot" of the algorithm's effectiveness (see Table 1), and the results obtained by the participants on a benchmark containing a wide spectrum of graphs have been presented at a DIMACS workshop [17].

The results obtained by *RLS* on these instances are as follows: if one considers the best among all values found by the fifteen heuristic algorithms presented at the DIMACS workshop, *RLS* reaches the same value or a better one in **34** out of **37** cases. In two instances corresponding to large graphs *RLS* finds new values (in one case better by one, in the other by three vertices). In the three cases where the current best value is not obtained, the difference is of one vertex in two cases, of four vertices in one case (corresponding to a graph designed to "fool" local-search based algorithms).

As a comparison, the best four competitors obtained the best value in $23 \div 27$ instances (**21** $\div$ **25** after considering the two new values found by *RLS*). The scaled times needed by three of them are larger than *RLS* times by at least a factor of ten. The fourth algorithm is slightly faster than *RLS* but found only 24 best values (**22** if the new values are considered), by executing hundreds of runs for most tasks.

The experimental efficacy and efficiency of *RLS* is strengthened by an analysis of the complexity of a single iteration. It is shown that the worst-case cost is $O(\max\{n, m\})$ where n and m are the number of nodes and edges, respectively. In practice, the cost analysis is pessimistic and the measured number of operations tends to be a small constant times the average degree of nodes in $\overline{G}$, the complement of the original graph.

The remaining part of this paper is organized as follows. After a short review of existing approaches for Max-Clique based on Tabu Search (Sec. 2) the motivation for *reactive* schemes is discussed (Sec. 3) and the *RLS* algorithm is presented (Sec. 4). The realization of *RLS* with data structures with minimal computational complexity is studied in Sec. 5. Then the experimental results obtained on a series of tasks recently proposed in the DIMACS challenge [17] are presented and discussed in Sec. 6. Two variants studied during the development of *RLS* are discussed in Sec. 7. A final discussion concludes the paper (Sec. 8).

2 Tabu Search heuristics for Max-Clique

As a recent bibliography about max-clique is present in [19], let us only mention some examples and results that are needed in the following discussion. Heuristics are powerful tools to search for good sub-optimal solutions of MC instances, and clearly they are a valued option if an exact solution (or an approximated solution within the requirements) cannot be guaranteed in the allotted number of iterations, as it is the case for large-size problems, given the theoretical results summarized in the Introduction. In addition, heuristics are crucial instrument to diminish the size of the search tree in exact branch and bound algorithms (as an example, coloring heuristics are used in the seminal work of Balas and Yu [3]).

Here the scope is limited to methods based on local search with *prohibition-based diversification* techniques. In particular, in the Tabu Search (TS) framework, diversification

is obtained through the *temporary prohibition* of some moves. Based on ideas developed independently by Glover [10] and Hansen and Jaumard [13], TS aims at maximizing a function f by using an iterative *modified local search*. At each step of the iterative process, the selected move is the one that produces the highest f value in the neighborhood. This move is executed even if f decreases with respect to the value at the current point, to exit from local optima. As soon as a move is applied, the inverse move is prohibited (i.e., not considered during the neighborhood evaluation) for the next T iterations. Prohibitions can be realized by using a first-in first-out list of length T (the "tabu list"), where the inverse of moves enter immediately after their execution, are shifted at each iteration, and therefore exit after T steps. A move is prohibited at a given iteration if and only if it is located in the "tabu list." This realization explains the traditional term *list size* for the parameter T, here the term *prohibition period* is preferred because it does not refer to a specific implementation. As a final remark, it is useful to contrast reactive memory-based schemes with algorithms based on Markov (i.e., memory-less) processes like Simulated Annealing [18], where the next configuration during the search is chosen with a probability that depends only on the current configuration.

Tabu Search has been used in Frieden et al. [9] for finding large stable sets (STABULUS). The size s_b of the independent set to search for is fixed, and the algorithm tries to minimize the number of edges contained in the current subset of s_b nodes (while aiming at reducing this number to zero).

Gendreau et al. [12] consider a different framework: the search space consists of legal cliques, whose size has to be maximized. Three different versions of TS are introduced and successfully compared with an iterated version of STABULUS. In the "iterated STABULUS" algorithm, an initial clique is found with a greedy technique (let $\tilde{k}$ be its size), then STABULUS is applied to the complement graph $\overline{G}$, trying to find cliques of size $\tilde{k}+1, \tilde{k}+2, ...$, until it fails to find one of the target size in a given maximum number of iterations. Two of the newly introduced TS versions are deterministic, one (ST) based on a single tabu list of the last $|T_1|$ solutions visited, the other (DT) adding a second list of the last $|T_2|$ vertices deleted. Only additions of nodes to the current clique can be restricted (deletions are always possible). The third version (PT) is stochastic: let S_t be the set of the vertices that can be added to the current clique I_t, if $|S_t| > 0$ a random sample of S_t is considered for a possible (non-tabu) addition, otherwise, if the current solution I_t is a local optimum and no nodes can be added, a number of randomly extracted nodes in I_t are removed from it. Additional diversification strategies are considered in [20] and used in [21].

3 Reactive search: the framework

One of the frequently raised criticisms about heuristic techniques is that it is difficult to judge about the *intrinsic quality* of schemes that contain many possible choices and free parameters [4]. As an example, Genetic Algorithms [14, 2] and advanced Simulated Annealing [16] versions with about five free parameters are not unusual, and one finds versions in the literature with up to about ten parameters. Tabu Search is not an exception: in the last years many versions with widely different characteristics have been studied and used [11].

In some cases parameters are tuned through a *feedback loop* that includes the user as a crucial *learning* component: depending on preliminary tests, some values are changed and

different options are tested until acceptable results on a set of instances are obtained. The quality of results is not automatically transferred to different instances and the feedback loop can require a lengthy "trial and error" process before acceptable results are obtained.

Reactive schemes aim at obtaining algorithms with an internal feedback loop. These schemes maintain the flexibility needed to cover in an efficient and effective way different instances of a problem, but the tuning is automated through feedback schemes that consider the past history of the search. Reaction is therefore memory-based: relevant information about past events is collected and used to influence the future part of the search.

In particular, it is of interest to study reactive algorithms based on *local-neighborhood search*. Local search is one of the most widely used heuristics, in which, after starting from an initial point (possibly randomly selected), one generates a *search trajectory* $X^{(t)}$ (t is the iteration counter) in the admissible search space $\mathcal{X}$. At each iteration, the successor $X^{(t+1)}$ of a point is selected from a *neighborhood* $N(X^{(t)})$ that associates to the current point $X^{(t)}$ a subset of $\mathcal{X}$. Local search can be classified as an *intensification* scheme, and, if the neighborhood structure is appropriate, it can be very effective in searching for good locally optimal configurations. Nonetheless, for many optimization problems of interest, a closer approximation to the global optimum is required, and therefore more complex schemes are needed (an example of a straightforward modification are multiple runs of local search, in which one starts from a different random point after reaching a local optimum).

Our research is focussed onto *automated diversification* schemes: diversification is enforced only when there is evidence – obtained from the past history – that diversification is needed. The basic Tabu Search cannot guarantee the absence of cycles and depends on an appropriate choice of T for its success. Reactive Tabu Search (*RTS*) [5] adapts T during the search so that its value is appropriate to the local structure of the problem, and uses a second long-term reactive mechanism to deal with confinements of the search trajectory that are not avoided by the use of temporary prohibitions: if too many configurations are repeated too often a sequence of random steps is executed. Hashing is used for the memory look-up and insertion operations. In the computational tests *RTS* generally outperforms non-reactive versions of TS and competitive algorithms like Simulated Annealing, Genetic Algorithms, Neural Networks [6, 7].

4 Reactive Local Search for Max-Clique

The *RLS* algorithm modifies *RTS* by taking into account the particular neighborhood structure of MC. This is reflected in the following two facts: feedback from the search history determines the prohibition parameter T, and an explicit memory-influenced restart is activated periodically as a long-term diversification tool (to assure that each vertex is eventually tried as a member of the current solution). Both building blocks of *RLS* use the memory about the past history of the search (set of visited cliques).

The admissible search space $\mathcal{X}$ is the set of all cliques X in an instance graph $G(V, E)$. The function to be maximized is the clique size $f(X) = |X|$, and the neighborhood $N(X)$ consists of all cliques that can be obtained from X by adding or dropping a single vertex. The neighborhood can be partitioned into $N^-(X)$ obtained by applying **drop** moves, and

$N^+(X)$ obtained by applying **add** moves.

$$N^+(X) = \{X' : X' \text{ is a clique }, X' = X \cup \{x\}, x \in V \setminus X\} \quad (1)$$
$$N^-(X) = \{X' : X' \text{ is a clique }, X' = X \setminus \{x\}, x \in X\} \quad (2)$$

Let us note that the neighborhood structure is symmetric ($X' \in N^-(X)$ iff $X \in N^+(X')$). The same neighborhood is exploited by many branch and bound algorithms and is used in the TS application in [21].

At a given iteration t of the search, the neighborhood set $N(X)$ is partitioned into the set of *prohibited* neighbors and the set *allowed* neighbors. The same terms *prohibited* and *allowed* are used for the corresponding add-drop moves. The prohibition rule is as follows: as soon as a vertex is added (dropped), it remains prohibited for the next T iterations. The prohibition period T is related to the amount of diversification. Let us define as $H(K, K')$ the symmetric difference of sets K and K'. In other words, $H(K, K')$ is the Hamming distance if the membership functions of the two sets are represented with binary strings with a bit for each vertex. In an admissible search space consisting of all n-bit binary strings, the requirement that $T \leq (n-2)$ is necessary and sufficient to assure that at least two moves are allowed (so that the search is not stuck and the move choice *is* influenced by the cost function value). In the MC case not every string corresponds to a clique and the requirement is only necessary but not sufficient (prohibitions need to be relaxed if no move is allowed, see Sec. 4.2). In the assumption that the above requirement is valid and that only allowed moves are executed, the relationship between T and the diversification [7] is as follows:

- The Hamming distance H between a starting point and successive point along the trajectory is strictly increasing for $T+1$ steps.

$$H(X^{(t+\tau)}, X^{(t)}) = \tau \quad \text{for} \quad \tau \leq T+1$$

- The minimum repetition interval R along the trajectory is $2(T+1)$.

$$X^{(t+R)} = X^{(t)} \;\Rightarrow\; R \geq 2(T+1)$$

The prohibition expires after a finite number of steps T because the prohibited moves can be necessary to reach the optimum in a later phase. In *RLS* the prohibition period is time-dependent, and therefore the notation $T^{(t)}$ will be used to stress this dependency. For a given $T^{(t)}$ the prohibition of a move is realized as follows:

Definition 4.1 *Let* LastMoved$[v]$ *be the last iteration that vertex* $v \in G$ *has been moved, i.e., added to or dropped from the current clique (*LastMoved$[v] = -\infty$*, at the beginning of the search).*
Vertex v *is* prohibited *at iteration* t *if and only if it satisfies:*

$$\text{LastMoved}[v] \geq (t - T^{(t)}) \quad .$$

```
REACTIVE-LOCAL-SEARCH
1     ▷ Initialization.
2     t ← 0 ; T ← 1 ; t_T ← 0 ; t_R ← 0 ;
3     X ← ∅ ; I_b ← ∅ ; k_b ← 0 ; t_b ← 0
4     repeat
5         T ← MEMORY-REACTION(X, T)
6         X ← BEST-NEIGHBOR (X)
7         t ← (t + 1)
8         if f(X) > k_b
9             then    I_b ← X ; k_b ← |X| ; t_b ← t
10        if (t − max{t_b, t_R}) > A
11            then  t_R ← t ; RESTART
12    until k_b is acceptable or maximum no. of iterations reached
```

Figure 1: *RLS* Algorithm: Pseudo-Code Description.

4.1 *RLS* : top-level view

The top-level description of the *RLS* algorithm is shown in Fig. 1. The description uses a pseudocode (lines beginning with "▷" are comments, "←" is the assignment, functions **return** values to the calling routines, fields of a compound object are accessed using $object.field$, etc.).

First the relevant variables are initialized: they are the iteration counter t, the prohibition period T, the time t_T of the last change of T, the last restart time t_R, the current clique X, the largest clique I_b found so far with its size k_b, and the iteration t_b at which it is found. The initialization of additional data structures will be described as soon as they are encountered. Then the loop (lines 5-11) continues to be executed until a satisfactory solution is found or a limiting number of iterations is reached.

In the loop, MEMORY-REACTION searches for the current clique in memory, inserts it into the hashing memory if it is a new one, and adjusts the prohibition T through feedback from the previous history of the search.

Then the best neighbor is selected and the current clique updated (line 6). The iteration counter is incremented. If a better solution is found, the new solution, its size and the time of the last improvement are saved (lines 8-9). A restart is activated after a suitable number A of iterations are executed from the last improvement and from the last restart (lines 10-11). In our tests A is set to $100 \cdot k_b$, as explained in Sec 4.3.

The prohibition period T is equal to one at the beginning, because in this manner one avoids coming back to the just abandoned clique. Nonetheless, let us note that *RLS* behaves exactly as local search in the first phase, as long as only new vertices are added to the current clique X (and therefore prohibitions do not have any effect). The difference starts when a maximal clique with respect to set inclusion is reached and the first vertex is dropped.

The differences with respect to multiple runs of local search (choice of best neighbor, restart when no improving move is available) are that the choice of the best neighbor takes the prohibition rule of Def. 4.1 into account and that the restart is executed after a suitably long search period and not after the first local optimum is encountered (Sec 4.3).

4.2 Choice of the best neighbor

```
BEST-NEIGHBOR (X)
1     ▷v is the moved vertex, type is ADDMOVE, DROPMOVE or NOTFOUND
2     type ← NOTFOUND
3     if |S| > 0 then
4         ▷try to add an allowed vertex first
5         ALLOWEDFOUND ← ({ allowed v ∈ S} ≠ ∅)
6         if ALLOWEDFOUND then
7             type ← ADDMOVE
8             MAXDEGALLOWED ← max_{allowed j∈S} deg_{G(S)}(j)
9             v ← random allowed w ∈ S with deg_{G(S)}(w) = MAXDEGALLOWED
10    if type = NOTFOUND then
11        ▷adding an allowed vertex was impossible: drop
12        type ← DROPMOVE
13        if ({ allowed v ∈ X} ≠ ∅) then
14            MAXDELTAS ← max_{allowed j∈X} DELTAS[j]
15            v ← random allowed w ∈ X with DELTAS[w] = MAXDELTAS
16        else
17            v ← random w ∈ X
18    INCREMENTAL-UPDATE(v, type)
19    if type = ADDMOVE then return X ∪ {v}
20    else return X \ {v}
```

Figure 2: *RLS* Algorithm: the function BEST-NEIGHBOR .

Let us define the set $S^{(t)}$ as follows:

Definition 4.2 *Let $X^{(t)}$ be the current clique at iteration t. $S^{(t)}$ is the vertex set of possible additions, i.e., the vertices that are connected to all $X^{(t)}$ nodes:*

$$S^{(t)} = \{v : v \in (V \setminus X^{(t)}), (v, j) \in E, \forall\, j \in X^{(t)}\}$$

Fig. 2 shows the selection algorithm (let us note that the t in iteration-dependent items like $S^{(t)}$ is dropped in the corresponding variable, like S). The choice of the best neighbor is influenced by the prohibition rule of Def. 4.1. The selection is executed in stages with this overall scheme: first an allowed vertex that can be added to the current clique is searched for (lines 3–9).

If no allowed addition is found, an allowed vertex to drop is searched for (lines 13–15). Finally, if no allowed moves are available, a random vertex in $X^{(t)}$ is dropped (line 17). Let us note that $X \neq \emptyset$ at line 17 (if $X = \emptyset$, then $S = V$, $|S| > 0$ and at least an allowed vertex is guaranteed at line 5 by the enforced bound $T \leq (n - 2)$).

Ties among *allowed* vertices that can be added are broken by preferring the ones with the largest degree [15, 21] in the subgraph $G(S^{(t)})$ induced by the set $S^{(t)}$. A random selection is executed among vertices with equal degree (lines 8–9).

Ties among *allowed* vertices that can be dropped are broken by preferring those causing the largest increase ($|S^{(t+1)}| - |S^{(t)}|$). A random selection is then executed if this criterion selects more that one winner (lines 14–15).

The above dropping choice is realized by introducing the set SMINUS and the quantities DELTAS[v].

Definition 4.3 *Let $X^{(t)}$ be the current clique at iteration t.* SMINUS$^{(t)}$ *is the set of ordered couples (v, x) such that vertex v has exactly one edge missing to the nodes of $X^{(t)}$, the edge (v, x):*
SMINUS$^{(t)} = \{(v, x) : v \in V, x \in X^{(t)}, (v, x) \notin E, (v, x') \in E \ \ \forall x' \in X^{(t)}, x' \neq x\}$
A vertex v is such that (v, x) is in SMINUS$^{(t)}$ *if and only if the number of edges in $G(V)$ incident to v and to $X^{(t)}$ nodes is $(|X^{(t)}| - 1)$.*

Because the vertex x that is not connected to $v \in$ SMINUS$^{(t)}$ is unique, SMINUS$^{(t)}$ can be projected to V (by considering the first element of the couple). The same term SMINUS$^{(t)}$ will be used for the projection (the meaning will be clear from the context).

Definition 4.4 *If a vertex $v \in X^{(t)}$ is dropped in passing from $X^{(t)}$ to $X^{(t+1)}$, $S^{(t+1)}$ receives all nodes that were lacking the edge to v but had all other edges to member of $X^{(t)}$. For each $v \in X^{(t)}$, let us define:*
DELTAS$[v] = |\{w : w \in (V \setminus S^{(t)}), (w, v) \notin E, (w, v') \in E \ \forall \ v' \in X^{(t)}, v' \neq v\}|$

Clearly, if $X^{(t+1)} = X^{(t)} \setminus \{v\}$, DELTAS$[v] = |S^{(t+1)}| - |S^{(t)}|$.

The *prohibition* status of a vertex is immediately determined if the function LASTMOVED$[v]$ is realized with an array. The data structures and operations concerning the just introduced sets are discussed in Sec. 5.2 (routine INCREMENTAL-UPDATE).

The relationship between the above introduced subsets of V is illustrated in Fig. 3 for an example graph (only the relevant connections are shown). Note that all vertices of X are present in SMINUS (or, better, in its projection), in fact each vertex $x \in X$ is not connected to itself.

4.3 Reaction and periodic restart

The memory about the past history of the search is used in two ways in the *RLS* algorithm: to adapt the prohibition parameter T (and therefore the amount of diversification) and to influence the restarts.

The prohibition T is minimal at the beginning ($T = 1$), and is then determined by two competing requirements. T has to be sufficiently large to avoid short cycles and the related waste of processing time during the search, it therefore increases when the same clique is repeated after a short interval along the trajectory, a symptom that diversification is required. On the other hand, large T values reduce the search freedom (in particular one has the requirement $T \leq (n - 2)$, see [7]): therefore, T is reduced as soon as frequent repetitions disappear.

The MEMORY-REACTION algorithm is illustrated in Fig. 4. The current clique X is searched in memory. If X is found, a reference Z is returned to a data structure containing the last visit time (line 2). If the repetition interval R is sufficiently short (only short cycles can be avoided through the prohibition mechanism [7]), cycles are discouraged by increasing T (lines 7–9).

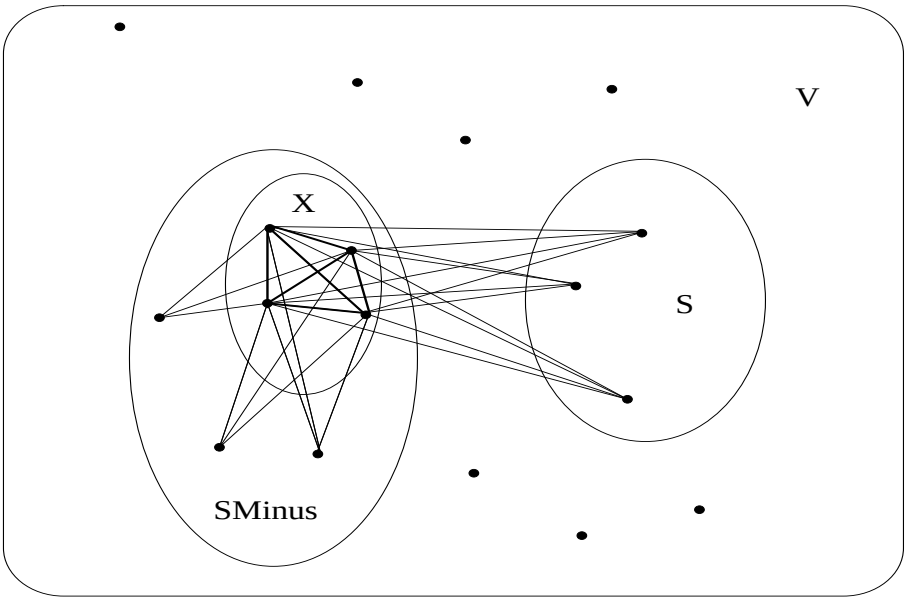

Figure 3: Subsets of V corresponding to $X^{(t)}$, $S^{(t)}$, and $\textsc{SMinus}^{(t)}$.

If X is not found, it is stored in memory with the time t when it was encountered (line 12). If T remained constant for a number of iterations greater than B, it is decreased (lines 14–15). It is appropriate that B scales with the maximum number of elements in a clique k_b, so that all clique members have many chances to be substituted as members of the current clique before a possible reduction of T is executed (the size of the current clique is close to k_b during the search). The value used in our tests is $B = 10 \cdot k_b$. Increases and decreases (with a minimal change of one unit, plus upper and lower bounds) are realized by the two following functions:

$$\begin{aligned} \textsc{Increase}(T) &= \min\{\max\{T \cdot 1.1, T+1\}, n-2\} \\ \textsc{Decrease}(T) &= \max\{\min\{T \cdot 0.9, T-1\}, 1\} \end{aligned}$$

Periodic restarts are needed to assure that the search is not confined in a limited portion of the search space (e.g., this is the case if the graph is composed of more than one connected component). Restarts are activated every $A = 10 \cdot B = 100\ k_b$ iterations, a period that permits a non-trivial dynamics of T with more possible increases and decreases (i.e., many B periods).

The routine Restart is adapted from [21]. Firts the prohibition parameter T is reset and the hashing memory structure is cleared (lines 1–2). If there are vertices that have never been part of the current clique during the search (i.e., that have never been moved since the beginning of the run), one of them with maximal degree in V is randomly selected (lines 4–7). If all vertices have already been members of X in the past, a random vertex in V is selected (line 9). Data structures are updated to reflect the situation of $X = \emptyset$, see

```
MEMORY-REACTION (X, T)
1     ▷search for clique X in the memory, get a reference Z
2     Z ← HASH-SEARCH(X)
3     if Z ≠ NULL then
4        ▷find the cycle length, update last visit time:
5        R ← t − Z.LASTVISIT
6        Z.LASTVISIT ← t
7        if R < 2 (n − 1) then
8           t_T ← t
9           return INCREASE(T)
10    else
11       ▷if the clique is not found, install it:
12       HASH-INSERT(X, t)
13    if (t − t_T) > B then
14       t_T ← t
15       return DECREASE(T)
16    return T
```

Figure 4: *RLS* Algorithm: routine MEMORY-REACTION .

lines 10–14 (XMISS and XMISSLIST are introduced in Sec. 5.2), then the selected vertex is added and the incremental update applied (lines 15–16).

5 Data structures and complexity analysis

The computational complexity of each iteration of *RLS* is the sum of a term caused by the usage and updating of reaction-related structures, and a term caused by the local search part (evaluation of the neighborhood and generation of the next clique).

Let us first consider the reaction-related part. The overhead per iteration incurred to determine the prohibitions is $O(|N(X)|)$, that for updating the last usage time of the chosen move is $O(1)$, that to check for repetitions, and to update and store the new *hashing value* of the current clique has an average complexity of $O(1)$, if an incremental *hashing* calculation is applied. If the entire clique is stored with the *digital tree* method [5] the worst case complexity is of $O(n)$.

In the maximum clique problem the complexity is dominated by the neighborhood evaluation. It is therefore crucial to consider incremental algorithms, in an effort to reduce the complexity below that required by a naive calculation "from scratch" of $|N(X)|$ different function values. As an example, an incremental evaluation is used to update S during successive **add** moves in [12], while S is recomputed from scratch after a **drop** move, with a worst-case complexity of $O(n^2)$. Now, after a transient phase of successive **add** moves if X is initially empty, **add** and **drop** moves are intermixed (long chains of **add** moves are rare) with approximately the same frequency.

This paper extends the incremental evaluation so that it is applied both after adding and after dropping a vertex. To this end, some auxiliary data structures are used. In particular, both the current clique X, the set S and SMINUS are represented with an *indicator set*,

```
RESTART
1     T ← 1   ;   t_T ← t
2     ▷ Clear the hashing memory
3     ▷ search for the "seed" vertex v
4     SOMEABSENT ← true  iff  ∃v ∈ V with LASTMOVED[v] = −∞
5     if  SOMEABSENT then
6         [ L ← {w ∈ V : LASTMOVED[w] = −∞}
7         [ v ←  random vertex with maximum deg_{G(V)}(v) in  L
8     else
9         v ←  random vertex   ∈ V
10    S ← V
11    SMINUS ← ∅
12    forall  v ∈ V
13        [ XMISSLIST[v] ← ∅ ; XMISS[v] ← 0
14        [ DELTAS[v] ← 0
15    X ← {v}
16    INCREMENTAL-UPDATE(v, ADDMOVE)
```

Figure 5: *RLS* Algorithm: routine RESTART .

see 5.1, DELTAS[v] with a n-dimensional array.

5.1 Indicator set

To realize some of the needed data structures with the lowest computational complexity, let us introduce a set structure that contains integers from 1 to n (with no duplications), and, in some cases, an additional positive integer for each contained element. The relevant operations to be executed are:

- The insertion of element i with related information $info$: INSERT($i, info$).
  If the information is not needed: INSERT(i).
- The removal of a single element i: DEL(i), returning $info$.
- The check for the presence of the i-th element: TEST(i), returning *true* or *false*.
- Action loops (with possible deletions) on all contained elements. The listing does not have to be in order.

The indicator set data structure is illustrated in Fig. 6. The structure consists of an n-dimensional array of records. The i-th record contains two indices (*prev* and *next*) used to realize a double-linked list of the contained elements (with a *NULL* index to signal the two ends of the list), and an additional variable (*info*) used as an indicator of the presence/absence of the i-th item, and possibly to contain additional information. The meaning is that *info*= -1 if and only if the item is not present, while all other values are used to store information associated to contained items. An additional variable *first* contains the index of the first item in the double-linked list (*NULL* if list is empty). Note that the

obtained linked list is not sorted. Clearly, pointers can be used instead of indices for *prev* and *next*. In addition, the total number of contained elements is recorded in *length*.

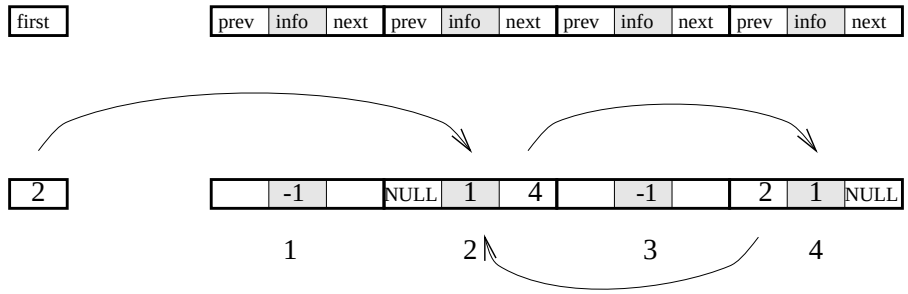

Figure 6: Indicator set structure (above) and example with n=4 and *length*=2 (below).

With the above data structure, INSERT , DEL, and TEST are $O(1)$, listing all elements requires $O(length)$ time. In fact, insertion requires an update of the *info* variable, then the item is inserted at the beginning of the double-linked list by modifying the appropriate *prev*, *next* and *first* indices. Deletion has a similar realization, the previous and next element are linked together, after setting $info \leftarrow -1$. Finally, all elements are listed in $O(length)$ time by starting from the *first* index and following the *next* pointer. Special cases (first or last element, empty list) are straightforward to take care of.

Indicator sets are used as a data structure for SMINUS (see Def. 4.3). The items to be stored are ordered couples of integers from 1 to n (a couple is present at most once). Now, for each v there is at most one x such that $(v, x) \in$ SMINUS, and therefore the associated x can be stored in the above described *info* variable of the indicator set. Removal of vertex v from SMINUS requires decrementing DELTAS[x]. After removing vertex v, the x value returned by SMINUS.DEL(v) is used to know which DELTAS[x] is to be decremented.

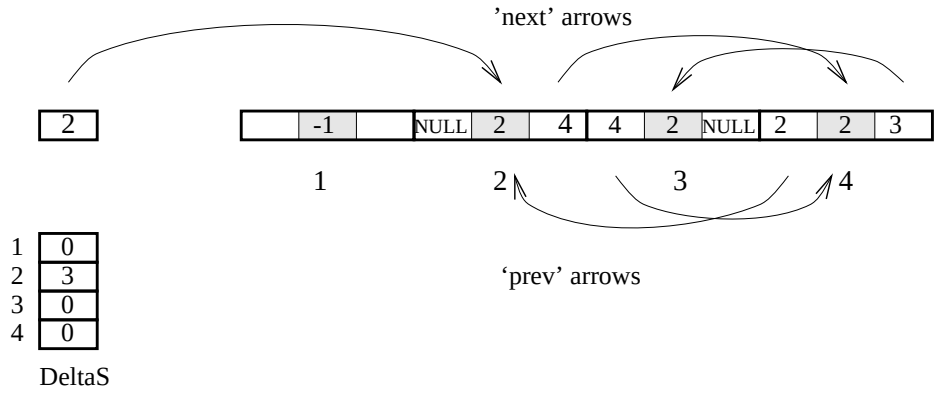

Figure 7: Data structure for SMINUS, and array DELTAS, an example with n=4 and three vertices (2,3,4) not connected to vertex 2 in X.

The data structures are illustrated in Fig. 7. The figure shows an example for SMINUS containing the couples $(2,2)$, $(3,2)$, and $(4,2)$.

5.2 The INCREMENTAL-UPDATE algorithm

The data structures described in Sec. 5 are adopted to realize the sets X, S, and SMINUS. An n-dimensional array XMISS is used to record the number of missing connection to set X for each vertex,

$$\text{XMISS}[v] = |\{i : i \in X, (i,v) \notin E\}|$$

and an *indicator set* XMISSLIST$[v]$ for each vertex is used to contain the list of lacking edges to members of X ("missing connections"). Let us note that XMISS$[v]$ can be stored as the *length* of the XMISSLIST$[v]$ indicator set, the notation has been chosen for clarity.

When an element is added to or dropped from X, the data structures S, SMINUS, XMISS and XMISSLIST are updated through the algorithm of Fig. 8.

```
INCREMENTAL-UPDATE (v, type)
1     ▷ Comment: v is is the vertex acted upon by the last move
2     ▷ type is a flag to differentiate between ADDMOVE and DROPMOVE
3     LASTMOVED[v] ← t
4     if type=ADDMOVE then
5         forall j ∈ N_Ḡ(v)
6             XMISSLIST[j].INSERT(v) ; XMISS[j] ← XMISS[j] + 1
7             if XMISS[j] = 1 then
8                 S.DEL(j)
9                 SMINUS.INSERT(j, v) ; DELTAS[v] ← DELTAS[v] + 1
10            else if XMISS[j] = 2 then
11                x ← SMINUS.DEL(j) ; DELTAS[x] ← DELTAS[x] − 1
12    else
13        forall j ∈ N_Ḡ(v)
14            XMISSLIST[j].DEL(v) ; XMISS[j] ← XMISS[j] − 1
15            if XMISS[j] = 0 then
16                x ← SMINUS.DEL(j) ; DELTAS[x] ← DELTAS[x] − 1
17                S.INSERT(j)
18            else if XMISS[j] = 1 then
19                x ← the only vertex contained in XMISSLIST[j]
20                SMINUS.INSERT(j, x) ; DELTAS[x] ← DELTAS[x] + 1
```

Figure 8: INCREMENTAL-UPDATE routine.

Let us demonstrate the correctness of the algorithm. First, let us note that the vertices *connected* to the just moved vertex v (defining $N_G(v)$) do not change their membership status with respect to S. Clearly, this property is not satisfied by v because $(v,v) \notin E$.

The membership of $w \in S$ does not change after **add** moves: if w was lacking at least one edge, trivially w will continue to lack the same edge, *vice versa*, if w had all edges to the old X, w will have all edges after the move. Similarly, S membership does not change after **drop** moves: if w was in S it will remain there (trivial), if w was not in S, then some

other edge beyond (w, v) must be missing to X members (in fact $(w, v) \in E$ for the above assumption that w is connected to v). At least the same edge must be missing after the move.

An analogous argument can be repeated for SMINUS membership, while the fact that XMISSLIST$[w]$ and XMISS$[w]$ are not changed if $(v, w) \in E$ is clear.

Therefore, all membership changes can possibly occur only for vertices *not* connected to the just moved one (i.e., for $j \in N_{\overline{G}}(v)$, the neighboring vertices of v in the complement graph).

Let us consider the case when vertex v is added to X (lines 4–11). For all non-connected j, v is added to the list of missing connections and the number of missing connections to X increases (line 6). If the number of missing connections from j to X is one, j was in S before the addition and now enters SMinus (lines 8–9). It could be added to X if v is dropped, therefore DELTAS$[v]$ increases. If the number of missing connections from j to X is two, j was in SMinus and has now to be deleted from it (lines 10–11), the value DELTAS$[x]$ is decreased for the single vertex x to which j was not connected.

The case when a vertex is dropped is easily demonstrated with analogous arguments. In particular, if XMISS$[j]$ is zero, j transfers from SMinus to S (lines 15–17), if XMISS$[j]$ is one, the vertex in X corresponding to the single missing edge is extracted from XMISSLIST$[j]$ and j enters SMINUS (lines 19–20).

If the lists of missing connections for each vertex are not available in the structure defining the task graph, they can be calculated in the preprocessing phase and stored for future use, for example in an *adjacency vectors* representation of $\overline{G}$. If this preprocessing is executed the following theorem is derived:

Theorem 5.1 *The incremental algorithm for updating X, S and* SMINUS *during each iteration of RLS has a worst case complexity of $O(n)$. In particular, if vertex v is added to or deleted from S, the required operations are $O(deg_{\overline{G}}(v))$.*

Let us note that the actual number of operations executed when vertex v is moved is a small constant times $deg_{\overline{G}}(v)$ and therefore the algorithm tends to be faster when the average degree in the complement graph $\overline{G}$ becomes smaller (e.g., for dense graphs with $deg_{\overline{G}}(v) << n$).

5.3 Update of $G(S)$ degrees

The INCREMENTAL-UPDATE algorithm is used to assure that the *sets* S and SMINUS reflect the current configuration along the search trajectory. In addition, the particular tie-breaking rule adopted in BEST-NEIGHBOR is based on the degrees in the induced subgraph $G(S)$ (see Fig. 2, lines 8–9). Their computation costs at most $O(m)$, m being the number of edges, by the following trivial algorithm. All the edges are inspected, if both end-points are in S, the corresponding degrees are incremented by 1.

Putting together all the complexity considerations the following corollary is immediately implied:

Corollary 5.1 *The worst-case complexity of a single iteration is $O(\max\{n, m\})$.*

In practice the above worst-case computational complexity is pessimistic. The degree is not computed from scratch but it is updated incrementally with a much lesser computational effort: in fact the maximum number of nodes that enter or leave $S^{(t)}$ at a given iteration is at most $deg_{\overline{G}}(v)$, v being the just moved vertex. Therefore the number of operations performed is at most $O(deg_{\overline{G}}(v) \cdot |S^{(t+1)}|)$. In actual runs, because the search aims at maximizing the clique $X^{(t)}$, the set $S^{(t)}$ tends to be very small (at some steps empty!) after a first transient period, and the dominant factor in the number of performed operations is the same $O(deg_{\overline{G}}(v))$ factor that appears in the Theorem 5.1 (Sec. 5.2).

5.4 Memory usage

Memory is used by the *RLS* algorithm to store information about the visited configurations (see the use of hashing in routine MEMORY-REACTION) and to realize the remaining data structures. In the assumption that t_{max} iterations are executed, that $O(t_{max})$ hash-table slots are available and that a fixed-size item for each new configuration is stored in a linked list corresponding to a given slot (collisions are resolved by chaining), the following theorem about the memory usage of the *RLS* algorithm holds:

Theorem 5.2 *The memory required by the RLS algorithm is* $O(n^2 + t_{max})$.

The demonstration is immediate after noting that each *indicator set* can be realized with $O(n)$ memory and that $O(n)$ of them are needed to realize the XMISSLIST sets used in the INCREMENTAL-UPDATE algorithm. All other data structures (apart from the hashing memory) do not require more that $O(n^2)$ memory space.

Let us note that $\Omega(n^2)$ is a lower bound for the memory usage if the adjacency matrix $A_G = (a_{ij})_{n \times n}$ of G is stored (its definition is: $a_{ij} = 1$ if $(i,j) \in E$ is an edge of G, and $a_{ij} = 0$ if $(i,j) \notin E$).

6 Experimental results

The *RLS* algorithm has been implemented in an object-oriented high-level language (C++) and tested on a series of benchmark tasks. In order to run the larger tasks on our machine (graphs with up to 4 000 vertices and 5 506 380 edges) the use of XMissList is avoided and line 19 in Fig. 8 is substituted with a loop over all X members, that is broken as soon as the missing edge to j is found. If the RAM memory is sufficient, the version in Fig. 8 should be preferred because of its better worst-case complexity (Theorem 5.1). Clearly, the memory usage can decrease if the individual bits of integer variables are used to store set-related information, while the computation speed can increase through the use of low-level languages. We did not pursue these issues because the results obtained by the object-oriented C++ version are fully satisfactory.

It is quite essential that heuristics are tested on problems arising in different areas and that the obtained results are compared with those obtained by competitive schemes on the same task suite. In particular, we present the results obtained on the benchmark defined as part of the international challenge organized by DIMACS [17].

6.1 DIMACS challenge

An international Implementation Challenge organized by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) has taken place in 1993. The purpose was to find effective optimization and approximation algorithms for **Maximum Clique**, **Graph Coloring**, and **Satisfiability**. The results obtained by the participants have been presented at a DIMACS workshop in Fall of 1993, with proceedings published by the AMS [17].

A small number of instances have been selected to provide a "snapshot" of the algorithm's effectiveness. Some benchmark graphs stress – or test the limits of – the various algorithms, and constitute an important base point to evaluate new heuristics in this area. The selected max-clique instances are listed in Table 1. The 37 tasks contains the following graphs (see [17] for additional details and references).

- Random Graphs. `Cx.y` and `DSJCx.y`, of size `x` and density `0.y`.
- Steiner Triple Graphs. `MANNx`
- Brockington Graphs. `brockx_2 and brockx_4` of size `x`.
- Sanchis graphs. `genx_p0.9_z` and `genx_p0.9_z`. of size `x`.
- Hamming graphs. `hamming8-4` and `hamming10-4`. with 256 and 1024 nodes, respectively.
- Keller Graphs. `keller4, keller5, keller6`.
- P–hat Graphs. `p_hatx-z`, of size `x`.

Standard timing routines for MC have been provided [17], the user times in seconds obtained by our workstation are listed in Table 2.

In order to assess the statistical variation, 10 runs have been performed for each task, with different sequences of pseudo-random numbers (used to break ties and during restarts). The maximum allotted time for each instance is the same that was used in [21], a competitive application of TS to MC.

The computational results are shown in Table 3. The CPU times include those for initializing the problems, while the number of iterations per second are calculated by subtracting the initialization times. Let BR denote the best result obtained by all DIMACS workshop participants. Most runs reach the "BR" value after a small fraction of the allotted time, in addition, the variation in the clique size obtained is zero for most instances. It is of interest to compare the performance of *RLS* with those obtained by the following fifteen heuristic algorithms presented at the DIMACS workshop [17]: 1) AtA (Grossman), 2) SA plus greedy (Homer and Peinado), 3) VHP (Gibbons, Pardalos, Hearn), 4) SM1 (Brockington and Culberson), 5) CLIQUEMERGE (Balas and Niehaus), 6) ST (Soriano and Gendreau), 7) DT (Soriano and Gendreau), 8) PT (Soriano and Gendreau), 9) GSD($\emptyset$) (Jagota, Sanchis, Ganesan), 10) SSD($\emptyset$) (Jagota, Sanchis, Ganesan), 11) $SSD_{RL}(\emptyset)$ (Jagota, Sanchis, Ganesan), 12) RB-clique (Goldberg and Rivenburgh) 13) tabu search and genetic hybrids (Fleurent and Ferland) 14) A (Bar-Yehuda, Dabholkar, Govindarajan, Sivakumar) 15) C (Bar-Yehuda, Dabholkar, Govindarajan, Sivakumar).

Given the size and difficulty of some tasks, only 20 values in Table 2 correspond to proved global optima (obtained with exact algorithms). The other values listed in the "BR" column are the best results obtained by the above heuristics. *RLS* reaches the BR value or a better one in **34** out of **37** cases. In two instances corresponding to large graphs *RLS* finds new values (for `C1000.9` better by one, for `C2000.9` better by three vertices). In the three cases where the current best value is not obtained, the difference is of one vertex in two cases, of four vertices in one case (`brock400_2`, a graph designed to "camouflage" the optimal solution).

As a comparison, the best four competitors (no. 2, 5, 12, 13 in the above list) obtained the best value in $23 \div 27$ instances (**21** $\div$ **25** after considering the two new values found by *RLS*). After taking into account the different computer speed, the scaled CPU times needed by three of them are larger than *RLS* times by at least a factor of ten. Algorithm no. 2 is slightly faster but found only 24 best values (**22** if the new values are considered), by executing hundreds of runs for most tasks.

File	Nodes	Edges	Density
C125.9	125	6963	0.898
C250.9	250	27984	0.899
C500.9	500	112332	0.900
C1000.9	1000	450079	0.901
C2000.9	2000	1799532	0.900
DSJC500.5	500	62624	0.501
DSJC1000.5	1000	249826	0.500
C2000.5	2000	999836	0.500
C4000.5	4000	4000268	0.500
MANN_a27	378	70551	0.990
MANN_a45	1035	533115	0.996
MANN_a81	3321	5506380	0.998
brock200_2	200	9876	0.496
brock200_4	200	13089	0.657
brock400_2	400	59786	0.749
brock400_4	400	59765	0.748
brock800_2	800	208166	0.651
brock800_4	800	207643	0.649
gen200_p0.9_44	200	17910	0.900
gen200_p0.9_55	200	17910	0.900
gen400_p0.9_55	400	71820	0.900
gen400_p0.9_65	400	71820	0.900
gen400_p0.9_75	400	71820	0.900
hamming8-4	256	20864	0.639
hamming10-4	1024	434176	0.828
keller4	171	9435	0.649
keller5	776	225990	0.751
keller6	3361	4619898	0.818
p_hat300-1	300	10933	0.243
p_hat300-2	300	21928	0.488
p_hat300-3	300	33390	0.744
p_hat700-1	700	60999	0.249
p_hat700-2	700	121728	0.497
p_hat700-3	700	183010	0.748
p_hat1500-1	1500	284923	0.253
p_hat1500-2	1500	568960	0.506
p_hat1500-3	1500	847244	0.753

Table 1: DIMACS "snapshot" benchmark files

r100.5	r200.5	r300.5	r400.5	r500.5
0.04	0.94	8.17	50.37	191.25

Table 2: User times for DIMACS machine benchmarks instances (HP 747i, 100 MHz clock, 32 Mb RAM, compiler: gcc -O2)

Name	Time to Best Avg (S.Dev.)	Avg Iter.	Iter./Sec.	Clique Size Min	Clique Size Avg(S.Dev.)	Clique Size Max	BR
C125.9	0.3 (0)	86.0	7700.0		34 (0)		34 *
C250.9	0.5 (0.1)	1484.6	14336.5		44 (0)		44 *
C500.9	20.4 (12.6)	170046.2	8632.6		57 (0)		57
C1000.9	145.3 (78.2)	998394.9	6930.3		**68** (0)		67
C2000.9	332.2 (306.0)	1435686.8	4357.8	76	77.1 (0.6)	**78**	75
DSJC500.5	0.6 (0.5)	2205.1	3517.6		13 (0)		14 *
DSJC1000.5	22.2 (16.0)	53173.8	2548.3		15 (0)		15 *
C2000.5	26.0 (23.8)	29734.4	1340.9		16 (0)		16
C4000.5	69.5 (29.4)	29294.5	585.4	17	17.2 (0.4)	18	18
MANN_a27	5.9 (6.2)	55247.2	9473.9		126 (0)		126 *
MANN_a45	178.1 (297.9)	1249278.1	6933.8	343	343.3 (0.5)	344	345 *
MANN_a81	883.2 (1019.3)	2591475.4	2942.7	1097	1097.6 (0.5)	1098	1098
brock200_2	5.6 (6.4)	22158.5	4082.4		12 (0)		12 *
brock200_4	6.8 (11.4)	36237.8	5409.5	16	16.3 (0.5)	17	17 *
brock400_2	1.1 (1.1)	7061.2	6258.9		25 (0)		29 *
brock400_4	2.4 (2.5)	14733.3	6235.0	25	26.6 (3.3)	33	33 *
brock800_2	13.8 (12.0)	50225.6	3951.2		21 (0)		21
brock800_4	23.7 (28.3)	87625.4	3986.0		21 (0)		21
gen200_p0.9_44	0.4 (0.1)	1751.3	18264.8		44 (0)		44 *
gen200_p0.9_55	0.4 (0)	543.0	12788.2		55 (0)		55 *
gen400_p0.9_55	5.0 (4.3)	41643.8	10495.4		55 (0)		55
gen400_p0.9_65	0.6 (0.1)	2053.8	11011.6		65 (0)		65
gen400_p0.9_75	0.6 (0.1)	1835.4	11922.3		75 (0)		75
hamming8-4	0.4 (0)	16.0	1600.0		16 (0)		16 *
hamming10-4	1.6 (0.1)	944.8	3655.2		40 (0)		40
keller4	0.3 (0)	99.6	5712.5		11 (0)		11 *
keller5	1.6 (0.8)	3185.0	4821.8		27 (0)		27
keller6	566.0 (298.2)	1061222.2	1928.3		59 (0)		59
p_hat300-1	0.4 (0)	154.0	5345.0		8 (0)		8 *
p_hat300-2	0.4 (0)	37.2	3127.8		25 (0)		25 *
p_hat300-3	0.5 (0.1)	951.4	9612.9		36 (0)		36 *
p_hat700-1	1.2 (0.5)	1021.1	2751.0		11 (0)		11 *
p_hat700-2	0.9 (0)	158.6	1983.5		44 (0)		44 *
p_hat700-3	0.9 (0)	346.2	2848.5		62 (0)		62
p_hat1500-1	39.5 (46.5)	61754.8	1613.3	11	11.6 (0.5)	12	12 *
p_hat1500-2	3.1 (0.1)	578.6	1349.3		65 (0)		65
p_hat1500-3	3.3 (0.2)	1696.8	2680.2		94 (0)		94

Table 3: Results on DIMACS Benchmark Instances, average time (with standard deviation) and iterations to find the best solution, average number of iterations per second, obtained clique size (with std. dev.). BR is the best result of all DIMACS workshop participants (* if optimality is proved).

The results obtained on the DIMACS instances not included in the "snapshot" suite are listed in Table 4. For these instances we have only the results obtained by Soriano and Gendreau [21] by using three different versions of Tabu Search. The range of results obtained by their algorithms is listed in the column labeled "SG."

Name	Time to Best Avg (S.Dev.)	Avg Iter.	Iter./Sec.	Cique Size Min	Avg(S.Dev.)	Max	SG
c-fat200-1	0.3 (0)	12.0	1200.0		12 (0)		12 *
c-fat200-2	0.3 (0)	24.0	2400.0		24 (0)		24 *
c-fat200-5	0.4 (0)	58.0	4350.0		58 (0)		58 *
c-fat500-1	0.6 (0)	14.0	1400.0		14 (0)		14 *
c-fat500-2	0.6 (0)	26.0	2022.2		26 (0)		26 *
c-fat500-5	0.6 (0)	64.0	1973.3		64 (0)		64 *
c-fat500-10	0.6 (0)	126.0	1939.5		126 (0)		126 *
johnson8-2-4	0.3 (0)	4.0	400.0		4 (0)		4 *
johnson8-4-4	0.3 (0)	14.0	1400.0		14 (0)		14 *
johnson16-2-4	0.3 (0)	8.0	800.0		8 (0)		8 *
johnson32-2-4	0 (0)	16.0	3078.6		16 (0)		16
hamming6-2	0.3 (0)	32.0	3200.0		32 (0)		32 *
hamming6-4	0.3 (0)	4.0	N/A		4 (0)		4 *
hamming8-2	0.4 (0)	128.0	3285.3		128 (0)		128 *
hamming10-2	2 (0)	512.0	1043.6		512 (0)		512 *
san200_0.7_1	1.7 (0.4)	7432.6	5431.2		30 (0)		16 ÷ 30 *
san200_0.7_2	3.5 (2.9)	16330.7	5194.6		18 (0)		15 ÷ 18 *
san200_0.9_1	1.9 (0.4)	16360.9	10504.6		70 (0)		47 ÷ 70 *
san200_0.9_2	0.8 (0.4)	6490.3	13566.9		60 (0)		41 ÷ 60 *
san200_0.9_3	0.6 (0.3)	3563.0	15169.5		44 (0)		36 ÷ 44 *
san400_0.5_1	3.4 (2.2)	7728.9	2880.9		13 (0)		8 ÷ 13 *
san400_0.7_1	2.5 (0.8)	10494.0	5168.8		40 (0)		21 ÷ 40 *
san400_0.7_2	5.6 (5.4)	24796.1	4940.1		30 (0)		18 ÷ 30 *
san400_0.7_3	9.7 (6.3)	45183.1	4805.6	18	21.6 (1.3)	22	17 ÷ 22 *
san400_0.9_1	0.5 (0)	104.2	2401.0		100 (0)		100 *
sanr200_0.7	0.4 (0)	366.0	10067.9		18 (0)		18 *
sanr200_0.9	0.6 (0.2)	3270.2	16752.8		42 (0)		41 ÷ 42 *
sanr400_0.5	2.2 (1.4)	6605.0	3799.7		13 (0)		12 ÷ 13 *
sanr400_0.7	0.4 (0.3)	2399.6	5469.8		21 (0)		20 ÷ 21
san1000	87.7 (73.7)	144690.0	1676.2		15 (0)		10
brock200_1	0.6 (0.3)	2446.8	12082.6		21 (0)		20 ÷ 21 *
brock200_3	3.1 (8.9)	14652.6	4865.0	14	14.1 (0.3)	15	14
brock400_1	2.9 (3.1)	18064.4	6199.7		25 (0)		24 ÷ 25
brock400_3	10.5 (19.2)	65658.9	6285.5	25	26.2 (2.5)	31	24 ÷ 25
brock800_1	9.2 (10.0)	36099.1	3926.4		21 (0)		20 ÷ 21
brock800_3	67.4 (60.8)	267502.7	3986.4		22 (0)		20 ÷ 21
p_hat500-1	0.6 (0)	100.8	2861.2		9 (0)		9 *
p_hat500-2	0.7 (0.2)	485.2	3066.1		36 (0)		36 *
p_hat500-3	0.5 (0.4)	2947.4	6273.1		50 (0)		49 ÷ 50
p_hat1000-1	0.1 (0.1)	243.8	2027.1		10 (0)		10
p_hat1000-2	0 (0)	222.6	4688.6		46 (0)		46
p_hat1000-3	0.6 (0.3)	3287.4	5527.3		**68** (0)		66

Table 4: Results on DIMACS Benchmark Instances, instances not considered in the "snapshot". SG is the range of results obtained by the three algorithms of Soriano and Gendreau, starred if the largest value is the global optimum. N/A (not available) means that the time is too short to be measured with the timing routine.

7 Variants of RLS

Different algorithmic choices have been investigated while developing the above presented *RLS* scheme. Both the computational complexity and the actual results obtained on the DIMACS benchmark have been considered before defining the presented algorithm.

In particular, the algorithm obtained if the degrees in the original graph are used instead of the $G(S)$ degrees in the BEST-NEIGHBOR routine is discussed in Sec. 7.1. The relaxation of prohibitions for selected "promising" moves (*aspriration criterion*) is considered in Sec. 7.2.

7.1 RLS without using $G(S)$ degrees

The values of the vertex degrees in the induced subgraph $G(S)$ are needed in the BEST-NEIGHBOR routine to break ties when more allowed vertices can be added to the current clique (Fig. 2 lines 8–9). The question whether those degrees are really needed in the heuristic is worth considering. In fact, a modified algorithm that does not make use of the updated degrees in $G(S)$ would have a better worst-case complexity per iteration of $O(n)$ because the $O(m)$ term caused by the updating illustrated in Sec. 5.3 would not be present.

To investigate the option, a variant has been considered in which the degrees in the complete graph $G = (V, E)$ are used instead of the degrees in the induces subgraph $G(S)$. In detail, the term $deg_{G(S)}$ in Fig. 2 (lines 8–9) has been substituted with $deg_{G(V)}$, whose values are calculated in the initialization part of the *RLS* algorithm. The computational results are listed in Table 5. As it was expected, the number of iterations per second always increases with respect to those obtained in Table 3. For some problems the iterations are up to two–three times faster. Unfortunately the best clique sizes obtained are in four cases inferior (see the graphs `MANN_a27, MANN_a45, MANN_a81`, and `keller6`) and the average time-to-best in the other cases (considering also the initialization phase) is not significantly better, and therefore the option was rejected. Nonetheless, the fact that comparable results were obtained in most tasks, implies that the use of $G(S)$ degrees in the move choice is crucial only for a limited subset of the considered benchmark tasks.

7.2 RLS with aspiration

An important element of traditional Tabu Search [10] is the incorporation of an *aspiration level criterion*. Its role is "to provide added flexibility to choose good moves by allowing the tabu status of a move to be overridden if the aspiration level is obtained." In particular, a simple aspiration criterion that is often used is that the prohibition of a move is relaxed if the cost function value that can be obtained by applying it is better than the "best so far" value.

A version of the BEST-NEIGHBOR function with an *aspiration* scheme has been tested. The details are illustrated in Fig. 9. In this version, a prohibited vertex can be added if an *aspiration* criterion is met (line 11) and if the degree of this vertex in $G(S)$ is larger than the maximum degree of allowed vertices. The *aspiration* criterion is that a clique larger than the current best is obtainable or, better, not excluded *a priori*.

The computational results obtained are listed in Table 6. The increased algorithm complexity is not justified by the obtained performance: the maximal clique sizes obtained

```
BEST-NEIGHBOR (X)
1     ▷v is the moved vertex , type is ADDMOVE or DROPMOVE
2     if |S| > 0 then
3         type ← ADDMOVE
4         MAXDEG ← maximum degree in G(S)
5         ALLOWEDFOUND ← ({ allowed v ∈ S} ≠ ∅)
6         if ALLOWEDFOUND = false then
7            MAXDEGALLOWED ← −1
8         else
9            MAXDEGALLOWED ← max. degree in G(S) for allowed vertices
10        PROHIBITEDNOTBETTER ← (MAXDEGALLOWED = MAXDEG)
11        ASPIRATION ← (|X| + MAXDEG + 1 > k_b)
12        if ALLOWEDFOUND and (PROHIBITEDNOTBETTER or not ASPIRATION) then
13            v ← random allowed v ∈ S with deg_{G(S)}(v) = MAXDEGALLOWED
14        else
15            v ← random v ∈ S with deg_{G(S)}(v) = MAXDEG
16    else
17        type ← DROPMOVE
18        if ({ allowed v ∈ X} ≠ ∅) then
19            MAXDELTAS ← max_j DELTAS[j]
20            v ← random allowed v ∈ X with DELTAS[v] = MAXDELTAS
21        else
22            v ← random v ∈ X
23    INCREMENTAL-UPDATE(v, type)
24    if type = ADDMOVE then return X ∪ {v}
25    else return X \ {v}
```

Figure 9: *RLS* Algorithm: function BEST-NEIGHBOR with *aspiration*.

are not larger with respect to the version without the aspiration criterion and the CPU times are statistically comparable.

Name	Time to Best Avg (S.Dev.)	Avg Iter.	Iter./Sec.	Clique Size Min	Clique Size Avg(S.Dev.)	Clique Size Max	BR
C125.9	0.3 (0)	91.6	10714.3		34 (0)		34 *
C250.9	0.4 (0)	740.6	15983.4		44 (0)		44 *
C500.9	15.2 (16.7)	134545.1	9667.6		57 (0)		57
C1000.9	100.7 (110.0)	780989.5	8001.1		68 (0)		67
C2000.9	484.0 (502.7)	2973148.2	6166.4	76	77.1 (0.7)	78	75
DSJC500.5	0.4 (0.5)	1660.0	3730.8		13 (0)		14 *
DSJC1000.5	16.5 (8.6)	45130.0	2982.3		15 (0)		15 *
C2000.5	20.1 (14.2)	30229.6	2047.8		16 (0)		16
C4000.5	538.2 (644.2)	632615.4	1209.7	17	17.6 (0.5)	18	18
MANN_a27	48.2 (26.4)	512507.5	10736.5		125 (0)		126 *
MANN_a45	288.6 (219.8)	2092805.7	7289.5	337	337.9 (0.6)	338	345 *
MANN_a81	544.1 (584.4)	1690646.1	3131.6	1082	1082.1 (0.3)	1083	1098
brock200_2	16.3 (11.5)	65376.1	4141.6	11	11.9 (0.3)	12	12 *
brock200_4	10.0 (8.9)	54146.0	5580.2	16	16.7 (0.5)	17	17 *
brock400_2	1.4 (1.4)	9127.3	6516.2		25 (0)		29 *
brock400_4	6.9 (16.5)	45157.7	6573.0	25	25.8 (2.5)	33	33 *
brock800_2	10.3 (8.3)	41115.4	4525.7		21 (0)		21
brock800_4	11.7 (5.8)	47074.8	4414.2		21 (0)		21
gen200_p0.9_44	0.5 (0)	2596.8	18469.8		44 (0)		44 *
gen200_p0.9_55	0.4 (0)	936.8	16375.5		55 (0)		55 *
gen400_p0.9_55	7.1 (6.0)	63678.2	10389.7		55 (0)		55
gen400_p0.9_65	0.6 (0.1)	1618.8	13748.8		65 (0)		65
gen400_p0.9_75	0.6 (0)	1402.6	13846.6		75 (0)		75
hamming8-4	0.4 (0)	25.2	2533.3		16 (0)		16 *
hamming10-4	1.6 (0.1)	869.8	6574.7		40 (0)		40
keller4	0.3 (0)	85.4	10513.3		11 (0)		11 *
keller5	6.4 (3.9)	29198.5	5764.4		27 (0)		27
keller6	1545.8 (1014.3)	4698501.9	3053.2	53	54.7 (1.1)	57	59
p_hat300-1	0.5 (0.2)	385.8	6270.7		8 (0)		8 *
p_hat300-2	0.4 (0)	29.0	2900.0		25 (0)		25 *
p_hat300-3	0.5 (0.1)	1175.0	14307.1		36 (0)		36 *
p_hat700-1	1.4 (0.4)	1619.1	3289.0		11 (0)		11 *
p_hat700-2	0.9 (0)	271.8	6183.7		44 (0)		44 *
p_hat700-3	0.9 (0.1)	414.4	7240.9		62 (0)		62
p_hat1500-1	35.0 (40.6)	61448.3	1888.7	11	11.7 (0.5)	12	12 *
p_hat1500-2	2.8 (0.1)	337.2	3440.9		65 (0)		65
p_hat1500-3	3.0 (0.2)	1262.8	5308.4		94 (0)		94

Table 5: Results on DIMACS "snapshot" instances, best move choice based on vertex degree in original graph, not on degree in $G(S)$.

Name	Time to Best Avg (S.Dev.)	Avg Iter.	Iter./Sec.	Clique Size Min	Clique Size Avg(S.Dev.)	Clique Size Max	BR
C125.9	0.3 (0)	120.8	8262.5		34 (0)		34 *
C250.9	0.5 (0.2)	1724.6	13752.7		44 (0)		44 *
C500.9	15.4 (16.6)	126332.2	9262.9		57 (0)		57
C1000.9	142.1 (133.5)	958708.0	6764.7		**68** (0)		67
C2000.9	444.6 (362.2)	1901291.4	4289.9	77	77.2 (0.4)	**78**	75
DSJC500.5	0.4 (0.4)	1419.6	3492.0		13 (0)		14 *
DSJC1000.5	20.6 (18.8)	48644.1	2507.9		15 (0)		15 *
C2000.5	35.0 (20.5)	42623.8	1417.2		16 (0)		16
C4000.5	59.2 (30.4)	22908.1	635.6		17 (0)		18
MANN_a27	10.5 (9.8)	104024.4	8589.2		126 (0)		126 *
MANN_a45	110.0 (95.4)	746743.0	6768.8	343	343.3 (0.5)	344	345 *
MANN_a81	497.7 (417.0)	1407070.9	2852.1	1097	1097.5 (0.5)	1098	1098
brock200_2	10.0 (8.6)	38745.5	4673.4	11	11.9 (0.3)	12	12 *
brock200_4	8.4 (10.4)	43900.0	5513.2	16	16.6 (0.5)	17	17 *
brock400_2	2.2 (1.5)	13263.1	6102.4		25 (0)		29 *
brock400_4	25.6 (29.3)	153550.9	6121.6	25	29.8 (4.1)	33	33 *
brock800_2	14.4 (9.7)	51216.6	3892.1		21 (0)		21
brock800_4	16.3 (8.5)	57980.1	3843.7		21 (0)		21
gen200_p0.9_44	0.4 (0.1)	1459.6	16086.1		44 (0)		44 *
gen200_p0.9_55	0.4 (0)	653.8	13913.5		55 (0)		55 *
gen400_p0.9_55	6.1 (5.5)	51667.9	9440.1		55 (0)		55
gen400_p0.9_65	0.6 (0)	1133.6	10209.1		65 (0)		65
gen400_p0.9_75	0.6 (0.1)	1808.0	11088.2		75 (0)		75
hamming8-4	0.4 (0)	16.0	1600.0		16 (0)		16 *
hamming10-4	1.7 (0.2)	508.0	1928.2		40 (0)		40
keller4	0.3 (0)	21.0	3100.0		11 (0)		11 *
keller5	2.1 (0.8)	5517.1	4418.1		27 (0)		27
keller6	1258.9 (996.4)	2318536.0	1875.7		59 (0)		59
p_hat300-1	0.4 (0.1)	188.2	5500.0		8 (0)		8 *
p_hat300-2	0.4 (0)	35.4	2720.0		25 (0)		25 *
p_hat300-3	0.5 (0.1)	1092.2	10437.1		36 (0)		36 *
p_hat700-1	1.3 (0.3)	1284.9	3240.5		11 (0)		11 *
p_hat700-2	0.9 (0)	114.0	1496.8		44 (0)		44 *
p_hat700-3	0.9 (0.1)	363.2	2723.3		62 (0)		62
p_hat1500-1	23.7 (26.9)	33662.6	1578.5	11	11.7 (0.5)	12	12 *
p_hat1500-2	3.2 (0.1)	712.2	1515.8		65 (0)		65
p_hat1500-3	3.4 (0.3)	1869.2	2810.7		94 (0)		94

Table 6: Results on DIMACS "snapshot" instances, with *aspiration*.

8 Summary and conclusions

A new heuristic algorithm based on local (neighborhood) search (RLS) has been proposed for the solution of the Maximum-Clique problem. The RLS algorithm is characterized by an internal feedback loop that determines the value of a prohibition parameter related to the amount of diversification. Through this mechanism a degree of flexibility is present that is appropriate for dealing with tasks with greatly different characteristics, but the user intervention in the tuning of parameters is avoided. The computational complexity per iteration of RLS has been analyzed in the worst case and extensive computational tests have been executed. In particular, when the experimental results are compared with those obtained by competing heuristics on the second DIMACS implementation challenge (at least, those presented at the 1994 workshop [17]), RLS appears to provide a significantly better performance, considering both the obtained clique sizes and the CPU times utilized.

Acknowledgments

We thank C. Mannino and A. Sassano for making available their C code for MC, P. Soriano and M. Gendreau for sending their Pascal code implementing Tabu Search. In addition we acknowledge useful discussions with them and with G. Di Caro. The present work has been partially funded by Special Project "Algorithms and Software for Optimization, and Models of Complex Systems" of the Univ. of Trento, MURST project "Efficienza di algoritmi e progetto di strutture informative," CNR grant "Strutture informative e teoria degli algoritmi" and Esprit Basic Research Action n.1741 (ALCOM II).

References

[1] G. Ausiello, P. Crescenzi, and M. Protasi, Approximate Solution of NP Optimization Problems, *Theoretical Computer Science*, to appear.

[2] T. Bäck and H. P. Schwefel, An Overview of Evolutionary Algorithms for Parameter Optimization, *Evolutionary Computation*, **1(1)** (1993), 1–23.

[3] E. Balas and C.S. Yu, Finding a Maximum Clique in an Arbitrary Graph, *SIAM J. Computing*, **14(4)** (1986), 1054–1068.

[4] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. Stewart, Designing and Reporting on Computational Experiments with Heuristic Methods, *Technical Report*, CS&E Dept., Southern Methodist University, Dallas, TX, 1995.

[5] R. Battiti and G. Tecchiolli, The reactive tabu search, *ORSA Journal on Computing*, **6(2)** (1994), 126–140.

[6] R. Battiti and G. Tecchiolli, Simulated annealing and tabu search in the long run: a comparison on QAP tasks, *Computer and Mathematics with Applications*, **28(6)** (1994), 1–8.

[7] R. Battiti and G. Tecchiolli, Local search with memory: Benchmarking RTS. *Operations Research Spectrum*, 1995, to appear.

[8] M. Bellare, O. Goldreich, and M. Sudan, Free bits, PCPs and non-approximability. Toward tight results, *Proc. 36-th Ann. Symp. on Foundations of Computer Science*, 1995, to appear.

[9] C. Friden, A. Hertz, and D. de Werra, STABULUS: A Technique for Finding Stable Sets in Large Graphs with Tabu Search, *Computing*, **42** (1989), 35–4.

[10] F. Glover, Tabu search - part I, *ORSA Journal on Computing*, **1(3)** (1989), 190–260.

[11] F. Glover, Tabu Search: Improved Solution Alternatives, in *Mathematical Programming, State of the Art 1994*, (J. R. Birge and K. G. Murty, eds.), The Univ. of Michigan Press, 1994, pp. 64–92.

[12] A. Gendreau, L. Salvail, and P. Soriano, Solving the Maximum Clique Problem Using a Tabu Search Approach, *Annals of Operations Research*, **41** (1993), 385–403.

[13] P. Hansen and B. Jaumard, Algorithms for the maximum satisfiability problem, *Computing*, **44** (1990), 279–303.

[14] J. Holland, *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI, 1975.

[15] D.S. Johnson, Approximation Algorithms for Combinatorial Problems, *J. Comput. and System Sciences*, **9** (1974), 256–278.

[16] D.S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning, *Operations Research*, **39** (1991), 378–406.

[17] D. Johnson and M. Trick (Eds.), *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, in press.

[18] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, Optimization by simulated annealing, *Science*, **220** (1983), 671–680.

[19] P.M. Pardalos and J. Xue, The maximum clique problem, *Journal of Global Optimization*, **4** (1994), 301–328.

[20] P. Soriano and M. Gendreau, Diversification Strategies in Tabu Search Algorithms for the Maximum Clique Problem, *Technical Report CRT-940*, CRT - Université de Montréal, Canada, 1993.

[21] P. Soriano and M. Gendreau, Tabu Search Algoritms for the Maximum Clique Problem, *Technical Report CRT-968*, CRT - Université de Montréal, Canada, 1994.