Reactive Local Search for the Maximum Clique Problem

Roberto Battiti
Dipartimento di Matematica, Univ. of Trento
Via Sommarive 14, 38050 Povo (Trento) - Italy
battiti@science.unitn.it

Marco Protasi *
Dipartimento di Matematica, Univ. of Roma "Tor Vergata"

Printed in: *Algorithmica , April 2001, vol. 29, no. 4, pag. 610–637*

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 (history-sensitive) 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.

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 [34].

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 $P \neq NP$, MC is not approximable within $n^{1/4-\epsilon}$ for any $\epsilon > 0$, n being

*Marco died on Feb 1, 1998.

the number of nodes in the graph [12], and it is not approximable within $n^{1-\epsilon}$ for any $\epsilon > 0$, unless $co - RP = NP$ [25].

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* [19] (TS for short) has been successfully applied to a growing number of problems, including MC [35]. 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. Similar ideas have been used in the past for the Traveling Salesman [36] and Graph Partitioning [31] problems.

Reactive schemes aim at obtaining algorithms with an internal feedback (*learning*) loop, so that the tuning is automated. In addition, the tuning acts *on-line*, while the algorithm runs on a single instance. Therefore properties of the *task* and *local* properties of the portion of the configuration space that is being visited can be taken into account. Reactive schemes need *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 [9]. *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. 37 significant MC instances were selected by the organizers to provide a "snapshot" of the algorithm's effectiveness, and the results obtained by the participants on a benchmark containing a wide spectrum of graphs were presented at a DIMACS workshop [30].

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 **36** 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 single cases where the current best value is not obtained, the difference is of one vertex.

As a comparison, when the same set of 37 graphs is considered, the best four competitors obtained the best value in a range from 23 to 27 instances (in a range from **21** to **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 Maximum Clique based on Tabu Search (Sec. 2) we discuss the motivation for *reactive* schemes (Sec. 3) and the adaptation to Maximum Clique (Sec. 3.1). We analyze a series of preliminary experiments executed to motivate the basic algorithm choices in Sec. 4 and present the *RLS* algorithm top-level view in Sec. 5. Then we study the realization of *RLS* with data structures with minimal computational complexity in Sec. 6. Finally we present the experimental results obtained on a series of tasks recently proposed in the DIMACS challenge [30] in Sec. 7. Two variants studied during the development of *RLS* are discussed in Sec. 8.

2 Tabu Search heuristics for Maximum Clique

As a recent bibliography about max-clique can be found in [34]. 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 [19] and Hansen and Jaumard [23], 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 history-sensitive schemes with algorithms based on Markov (i.e., memory-less) processes like Simulated Annealing [32], where the next configuration during the search is chosen with a probability that depends only on the current configuration.

A problem related to the Maximum Clique is the Maximum Stable Set, where one is to find a set of pairwise nonadjacent nodes of maximum cardinality. A maximum stable set of G is a maximum clique of $\overline{G}$, the graph-theoretic complement of G. Tabu Search has been used in Frieden et al. [17] for finding large stable sets (STABULUS). The size s_b of the stable 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. [20] 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 used in [35].

3 Reactive search: the framework

One of the frequently raised criticisms is that it is difficult to judge the *intrinsic quality* of schemes that contain many possible choices and free parameters [5]. As an example, Genetic Algorithms [26, 3] and advanced Simulated Annealing [29] versions with about five free parameters are not unusual, and one finds versions in the literature with up to about ten parameters.

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 history-sensitive: 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*) [9] adapts T during the search so that its value is appropriate to the local structure of a specific instance of a 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 [10, 11, 6, 8]. Solutions of better quality are obtained by using comparable and often less running times.

3.1 Adaptation to Maximum 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 (the set of visited cliques with visit times).

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 [35].

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 $\Delta(K, K')$ the symmetric difference of sets K and K'. In other words, $\Delta(K, K')$ 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. 5.1). In the assumption that the above requirement is valid and that only allowed moves are executed, the relationship between T and the diversification [11] is as follows:

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

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

- Let us define as a *repetition* the fact that the same configuration is found multiple times along the search trajectory, i.e., one has a repetition at time $t + R$ (for $R > 0$) if $X^{(t+R)} = X^{(t)}$. The repetition interval R along the trajectory is lower-bounded by $2(T + 1)$.

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

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 3.1 *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:*

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

4 A preliminary experimental study: local search with prohibitions

After starting from the general *Reactive Search* framework, we executed many implementation decisions to obtain the final algorithm. While some readers may be interested in the final "horse-race" results, we think that it is important to motivate the different choices through focussed experiments and to judge the relative impact of the different choices on the final results and the robustness of the algorithm with respect to variations of its parameters.

We aim at a compromise: while it would be too long and tedious to illustrate in detail all experiments that have been executed to obtain the final algorithm, we concentrate on a subset of the experiments and on a subset of the graphs. These experiments motivate the three basic choices that have been executed after starting from the basic local search method.

The first choice deals with the criterion used to judge potential candidates in the neighborhood. In particular, because the function to be optimized (the number of nodes in a clique) changes by only two possible values (plus or minus one) in many cases there will be more candidates leading to the same change. We show experimentally that a partially deterministic *tie-breaking* citerion for selecting among the neighbors is preferable to a completely random selection among the tied best neighbors.

The second choice is that of *continuing* the search in the vicinity of a local optimum after the point is reached by the basic local search technique. The alternative choice in this case is that of restarting from a new initial configuration immediately after reaching the local optimum.

The third choice is that of introducing *prohibitions*: some neighbors are prohibited to enforce diversification. The alternative is of course that of allowing all neighbors when the search is continued beyond a locally optimal point.

Additional choices have been executed to determine the values of the parameters used in the routines Increase and Decrease, and the values of the parameters A and B, see Sec. 5.2. In this case a preliminary choice has been executed based on previous experiments with reactive schemes, see for example [6, 8, 9]. Then a number of robustness tests have been executed: the parameters were changed in both directions by a factor of two and four without observing significant and consistent performance changes for all graph kinds.

But we did *not* optimize these parameters on the given tasks, both because the detailed optimization would require excessive CPU times and because we wanted to show that a simple

reactive scheme is capable of reaching state-of-the-art results even in the absence of a detailed optimization on the given tasks. Furthermore, it is not scientifically correct to tune the parameters on the same tasks that are used in the final tests. One should consider different tasks extracted from the same random generator. Of course, the implication is that, in principle, better results could be obtained by additional performance tuning, especially if one considers only specific problem classes. For example, one could consider only random graphs of different densities and study the optimal parameter settings as a function of the graph density. This is an open research area for future investigations.

All tests have been executed on a Pentium II personal computer, with 450 MHz clock, 384 Mb RAM, Linux operating system and GNU g++ compiler (with optimizaion flag -O2).

4.1 Local search plus tie-breaking

Let us now describe the different variations of local search that have been tested on the Maximum Clique problem. The basic local search scheme that we consider is based on the neighborhood introduced in Sec 3.1. The algorithm starts from an empty clique. At each iteration, it consider the nodes that are connected to all nodes of the current clique, i.e., the set PossibleAdd of possible additions, see also Sec. 5.1. It selects a candidate in PossibleAdd and adds it to the current clique. When a maximal clique is reached, such that the set PossibleAdd is empty, the algorithm is restarted from a clique consisting of a single node, by calling the Restart routine that will be explained in Sec. 5.2.

The two versions differ by the manner in which a winning candidate is selected from PossibleAdd. In the first option (labeled "simple") one picks a random candidate, distributed with uniform probability. In the second option (labeled "tie-break") one picks a random node in the subset of PossibleAdd containing the nodes with the largest degree in the subgraph G(PossibleAdd), the graph induced by the set PossibleAdd.

The motivation for the second option is that a larger degree implies a larger size of PossibleAdd after the node is added to the current clique, and therefore a larger freedom of choice for subsequent additions.

Each run is stopped when either the "best" value listed in Table 9 (in the column with label "BR") is reached or a maximum number of $1000 \times n$ iteration is reached. The "best" value is proved to be optimal in some cases, it is the best heuristic value available for the other graphs, see Sec. 7.1 for details. A limit on the number of iterations is needed because some runs do not reach the best value even if a day of CPU time is allocated.

A total of 100 runs with different random number generator seeds is executed to obtain the average results listed in Table 1 and Table 2.

Table 1 and Table 2 show the average CPU time in seconds to reach the best solution during a single run (with standard deviation), the average number of iterations executed to reach the best value, the average number of iterations per second, the average clique size obtained in each run (with standard deviation) and the best heuristic result found in the DIMACS challenge, see Sec. 7.1. for details.

The superiority of the "tie-break" version is evident. The "simple" local search method reaches the best value only in one case (`p_hat300-1`) but with a much larger number of iterations (4686 versus 77). For all other graphs the "tie-break" version obtains significantly better average clique sizes, already coincident with the best values in six cases. In the remaining three cases the values obtained are less than 2% less than the best values.

Name	Time to Best Avg (S.Dev.)	Avg Iter.	Iter./Sec.	Clique Size Avg (S.Dev.)	BR
p_hat300-1	1.563 (1.597)	4686.579	2938.412	8 (0)	8
p_hat300-2	23.758 (18.730)	103597.184	4359.642	23.184 (0.652)	25
p_hat300-3	21.138 (13.596)	116616.132	5514.462	31.447 (0.686)	36
p_hat700-1	263.090 (195.378)	189996.605	722.186	10.132 (0.475)	11
p_hat700-2	233.901 (166.783)	315505.447	1348.921	37.184 (1.159)	44
p_hat700-3	194.174 (120.140)	328366.158	1690.783	49.895 (1.034)	62
p_hat1500-1	1919.542 (1859.729)	353498.297	184.258	11.054 (0.229)	12
p_hat1500-2	1661.893 (1060.736)	669910.784	403.541	49.189 (1.309)	65
p_hat1500-3	1469.637 (906.762)	742078.568	505.131	66.730 (1.866)	94

Table 1: Results of "simple" repeated local search algorithm.

Name	Time to Best Avg (S.Dev.)	Avg Iter.	Iter./Sec.	Clique Size Avg (S.Dev.)	BR
p_hat300-1	0.027 (0.017)	77.480	2857.045	8 (0)	8
p_hat300-2	0.412 (0.425)	2298.940	5579.51	25 (0)	25
p_hat300-3	17.435 (14.199)	114242.710	6551.584	35.780 (0.416)	36
p_hat700-1	12.560 (12.349)	10943.720	835.539	11 (0)	11
p_hat700-2	10.706 (5.328)	20909.580	1922.487	44 (0)	44
p_hat700-3	17.503 (8.822)	41771.550	2381.675	62 (0)	62
p_hat1500-1	8.323 (0.230)	1537.740	184.751	12 (0)	12
p_hat1500-2	827.396 (651.565)	531504.530	643.149	64.830 (0.378)	65
p_hat1500-3	626.933 (481.898)	508699.230	810.888	92.420 (0.496)	94

Table 2: Results of "tie-break" repeated local search algorithm.

4.2 Local search plus plateau search

For some problems like Maximum Satisfiability one finds that an increased performance can be obtained by spending some additional iterations after one reaches the local optimum and before restarting, see for example [8] and the references therein. In some cases the first local optimum reached is close to better configurations and , on average, more iterations are required to reach configurations of comparable quality after restarting from a new initial point.

For Maximum Satisfiability this phase is called "plateau search" because, when no improving move is available, one accepts moves such that the function to be optimized does not change.

For The Maximum Clique problem each move changes the function value by either plus or minus one and the term "plateau search" has a different meaning. In detail, we consider the "tie-break" version of local search but do not restart immediately after reaching a local optimum. When this happens, an additional number of iterations are executed with the same criterion to select neighbors. Of course, the difference is now that the chosen best neighbor is accepted even if the size of the clique decreases. This happens if no node can be added to the current clique.

Table 3 shows the results obtained for an experiment designed in the same manner as the one with the "tie-break" repeated local search. In particular, the same number of total

Name	Time to Best Avg (S.Dev.)	Avg Iter.	Iter./Sec.	Clique Size Avg (S.Dev.)	BR
p_hat300-1	0.018 (0.012)	99.320	5420.283	8 (0)	8
p_hat300-2	0.014 (0.019)	108.970	7783.571	25 (0)	25
p_hat300-3	0.253 (0.406)	3003.550	11447.723	36 (0)	36
p_hat700-1	1.444 (0.970)	2421.110	1593.724	11 (0)	11
p_hat700-2	0.112 (0.105)	367.360	3206.504	44 (0)	44
p_hat700-3	0.276 (0.325)	1213.070	4118.682	62 (0)	62
p_hat1500-1	9.002 (10.618)	3757.860	398.635	12 (0)	12
p_hat1500-2	1.981 (2.011)	2306.880	1143.393	65 (0)	65
p_hat1500-3	66.479 (76.563)	105135.120	1570.087	94 (0)	94

Table 3: Results of "tie-break" repeated local search algorithm plus plateau search.

iterations is considered for each run. The only difference is that, if a local optimum is reached in t iterations after the last restart, an additional t iterations of local search are executed before the next restart.

When one compares the results of Table 3 with those of Table 2 one observes a striking performance jump. The best values are reached in all runs. This improves the previous results for the p_hat300-3, p_hat1500-2 and p_hat1500-3 graphs. For the other graphs, in two cases the total number of iterations increases (from 77 to 99 for p_hat300-1, from 1,537 to 3,757 for p_hat1500-1). In all other cases a dramatic reduction in the number of iterations is observed (from 2,298 to 108 for p_hat300-2, from 10,943 to 2,421 for p_hat700-1, from 20,909 to 367 for p_hat700-2, etc.).

To summarize, for the simple tasks such that the plateau search does not help, on average the iterations are at most doubled (with our choice, the plateau search doubles the number of iterations in comparison with the previous "tie-break" local search). For the more difficult tasks, where the first local optima found after restarting tend to be of low quality, local optima of better quality are found in a very efficient way during the plateau search phases.

Our experiments confirm the previous results for the Maximum Satisfiability problem. We think that the results are very intriguing and a theoretical explanation of these results is an open research issue.

4.3 Local search plus prohibitions

In the previous section we have seen that, if one continues the search for a certain number of iterations after reaching a local optimum, one obtains a dramatic reduction in the total number of iterations needed to solve certain tasks.

Nonetheless, the previous technique can still be improved by adding the prohibition mechanism described in Sec. 3.1. In fact, what can happen is that the plateau search phase remains confined in a limited portion of the search space, a portion at a limited Hamming distance from the current configuration. This is the case if the algorithm enters a cycle (e.g., the algorithm keeps adding and removing the same node from a clique). While cycles are discouraged by the randomized selection of a winning candidate, a more frequent case is that the algorithm keeps adding and removing a small set of nodes.

To see the effect of the *prohibition period* T on the average number of iterations needed

to solve the benchmark tasks, we executed a series of tests in the same conditions as those of Sec. 4.2 with the only difference that the prohibition mechanism is now activated during the search. The tests are on the `p_hat` tasks and on the `C125.9` and `C250.9` random graphs, see [30] for details.

Prohibition	p_hat300-1	p_hat300-2	p_hat300-3
T = 0	116.1 (8.3)	127.4 (18.2)	2357.5 (373.7)
T = 1	119.5 (9.7)	58.1 (6.4)	1701.4 (193.0)
T = 2	**95.0** (6.7)	61.3 (7.2)	1490.2 (220.2)
T = 4	149.9 (11.7)	69.7 (10.6)	**1413.5** (152.5)
T = 8	143.4 (12.3)	**55.5** (7.5)	1643.9 (223.1)
T = 16	184.4 (16.4)	203.0 (33.4)	2624.9 (263.9)
T = 32	316.9 (51.6)	18890.3 (2569.9)	92753.8 (7477.8)

Table 4: Average number of iterations to reach the best value for the "tie-break" repeated local search algorithm plus plateau search on the `p_hat300` tasks. Results for different values of the prohibition T.

Prohibition	p_hat700-1	p_hat700-2	p_hat700-3
T = 0	2418.1 (172.9)	334.2 (28.3)	1175.3 (122.7)
T = 1	1719.7 (142.9)	176.1 (13.7)	512.0 (54.8)
T = 2	**1690.8** (154.1)	172.2 (13.5)	539.5 (60.7)
T = 4	1893.8 (164.4)	171.0 (11.1)	498.4 (55.9)
T = 8	10575.0 (1278.4)	**166.7** (13.8)	**457.5** (52.0)
T = 16	16230.2 (1644.5)	204.3 (19.8)	599.3 (58.9)
T = 32	20473.8 (2104.2)	2830.5 (396.3)	4752.0 (490.5)

Table 5: Average number of iterations to reach the best value for the "tie-break" repeated local search algorithm plus plateau search on the `p_hat700` tasks. Results for different values of the prohibition T.

The average number of iterations needed to reach the best value (that is reached in all 100 runs) is listed in Table 4, Table 5, and Table 6 for the different values considered for the prohibition ($T = 0, 1, 2, 4, 8, 16, 32$). The random seed used is the same for runs differing only in the value of the prohibition. Of course, it changes after all prohibitions have been tested and a new series of runs is initiated.

In general, an appropriate value of the prohibition significantly reduces the number of iterations for most tasks. For example, for the `p_hat1500-3` task, the iterations are reduced from 96,700 ($T = 0$) to 14,111 ($T = 16$).

Qualitatively similar results have been found in experiments on different graphs of the DIMACS benchmark suite. As an example, Table 7 shows the results for two random graphs.

Let us note that the optimal value of the prohibition depends on the task. For example, among the considered values, the optimal value is $T = 0$ (no prohibition) for `p_hat1500-1`, $T = 16$ for `p_hat1500-2` and `p_hat1500-3`.

Furthermore, the most appropriate value does not depend only on the task but also on the *local properties* of the particular region that the search is visiting.

Prohibition	p_hat1500-1	p_hat1500-2	p_hat1500-3
T = 0	**3488.1** (456.7)	2513.9 (355.4)	96700.0 (10607.1)
T = 1	5820.8 (1318.5)	948.3 (105.4)	30857.1 (4087.3)
T = 2	11646.4 (2396.1)	867.5 (81.9)	31385.5 (4220.7)
T = 4	12940.6 (2600.5)	823.5 (90.6)	26756.6 (3288.2)
T = 8	28588.6 (3447.0)	658.6 (70.0)	21755.0 (2191.9)
T = 16	23189.7 (2843.6)	**635.2** (58.5)	**14111.6** (1731.6)
T = 32	35686.4 (3680.6)	3031.5 (334.2)	28998.5 (3607.0)

Table 6: Average number of iterations to reach the best value for the "tie-break" repeated local search algorithm plus plateau search on the `p_hat1500` tasks. Results for different values of the prohibition T.

Prohibition	C125.9	C250.9
T = 0	451.73 (50.40)	6099.73 (499.03)
T = 1	243.44 (27.93)	2794.43 (293.65)
T = 2	245.36 (28.28)	2789.46 (267.21)
T = 4	177.22 (21.56)	2912.66 (257.43)
T = 8	135.32 (18.00)	**2525.08** (272.66)
T = 16	**133.20** (17.03)	4538.52 (539.52)
T = 32	1861.18 (124.49)	21523.32 (1768.97)

Table 7: Average number of iterations to reach the best value for the "tie-break" repeated local search algorithm plus plateau search on the `C125.9` and `C250.9` tasks. Results for different values of the prohibition T.

The task and local dependencies motivate the adoption of simple mechanisms to adapt the prohibition value while the search runs. Additional experiments on the effect of the prohibition have been executed for different problems, see for example [6, 8, 9].

5 Reactive Local Search for Max Clique: top-level view

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

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

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 $obj.field$, etc.). Descriptive long variable names are used.

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 CURRENTCLIQUE, the largest clique BESTCLIQUE found so far with its size MAXSIZE, and the iteration t_b at which it is found. 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 $100 \cdot$ MAXSIZE, as explained in Sec 5.2.

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 CURRENTCLIQUE (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. 3.1 into account and that the restart is executed after a suitably long search period and not after the first local optimum is encountered (Sec 5.2). In addition, *RLS* is characterized by the way it uses randomization.

5.1 Choice of the best neighbor

```
BEST-NEIGHBOR (CURRENTCLIQUE)
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 ∈ POSSIBLEADD} ≠ ∅)
6        if ALLOWEDFOUND then
7            type ← ADDMOVE
8            MAXDEG ← max_{allowed j∈POSSIBLEADD} {deg_{G(POSSIBLEADD)}(j)}
9            v ← random allowed w ∈ POSSIBLEADD with deg_{G(POSSIBLEADD)}(w) = MAXDEG
10   if type = NOTFOUND then
11       ▷adding an allowed vertex was impossible: drop
12       type ← DROPMOVE
13       if ({ allowed v ∈ CURRENTCLIQUE} ≠ ∅) then
14           MAXDELTAPA ← max_{allowed j∈CURRENTCLIQUE} DELTAPA[j]
15           v ← random allowed w ∈ CURRENTCLIQUE with DELTAPA[w] = MAXDELTAPA
16       else
17           v ← random w ∈ CURRENTCLIQUE
18   INCREMENTAL-UPDATE(v, type)
19   if type = ADDMOVE then return CURRENTCLIQUE ∪ {v}
20   else return CURRENTCLIQUE \ {v}
```

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

Let us define the set $\text{POSSIBLEADD}^{(t)}$ as follows:

Definition 5.1 *Let* $\text{CURRENTCLIQUE}^{(t)}$ *be the current clique at iteration* t. $\text{POSSIBLEADD}^{(t)}$ *is the vertex set of possible additions, i.e., the vertices that are connected to all* $\text{CURRENTCLIQUE}^{(t)}$ *nodes:*

$$\text{POSSIBLEADD}^{(t)} = \{v : v \in (V \setminus \text{CURRENTCLIQUE}^{(t)}), (v, j) \in E, \ \forall\, j \in \text{CURRENTCLIQUE}^{(t)}\}$$

Fig. 2 shows the selection algorithm (let us note that the t in iteration-dependent items like $\text{POSSIBLEADD}^{(t)}$ is dropped in the corresponding variable, like POSSIBLEADD). The choice of the best neighbor is influenced by the prohibition rule of Def. 3.1. The selection is executed in stages with this overall scheme: first an allowed vertex that can be added to the current clique is sought (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 is picked with uniform probability on $\text{CURRENTCLIQUE}^{(t)}$ and it is dropped (line 17). Let us note that CURRENTCLIQUE $\neq \emptyset$ at line 17 (if CURRENTCLIQUE $= \emptyset$, then POSSIBLEADD $= V$, |POSSIBLEADD| > 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 in the subgraph $G(\text{POSSIBLEADD}^{(t)})$ induced by the set $\text{POSSIBLEADD}^{(t)}$. 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 ($|\text{PossibleAdd}^{(t+1)}| - |\text{PossibleAdd}^{(t)}|$). 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 OneMissing and the quantities DeltaPA$[v]$.

Definition 5.2 *Let* CurrentClique$^{(t)}$ *be the current clique at iteration* t. OneMissing$^{(t)}$ *is the set of ordered pairs* (v, x) *such that vertex* v *has exactly one edge missing to the nodes of* CurrentClique$^{(t)}$*, the edge* (v, x)*:*
OneMissing$^{(t)}$ $= \{(v, x) : v \in V, x \in \text{CurrentClique}^{(t)}, (v, x) \notin E, (v, x') \in E \quad \forall x' \in \text{CurrentClique}^{(t)}, x' \neq x\}$
A vertex v *is such that* (v, x) *is in* OneMissing$^{(t)}$ *if and only if the number of edges in* $G(V)$ *incident to* v *and to* CurrentClique$^{(t)}$ *nodes is* $(|\text{CurrentClique}^{(t)}| - 1)$.

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

Definition 5.3 *If a vertex* $v \in \text{CurrentClique}^{(t)}$ *is dropped in passing from* CurrentClique$^{(t)}$ *to* CurrentClique$^{(t+1)}$, PossibleAdd$^{(t+1)}$ *receives all nodes that were lacking the edge to* v *but had all other edges to member of* CurrentClique$^{(t)}$*. For each* $v \in \text{CurrentClique}^{(t)}$*, let us define:*
DeltaPA$[v] = |\{w : w \in (V \backslash \text{PossibleAdd}^{(t)}), (w, v) \notin E, (w, v') \in E \; \forall \; v' \in \text{CurrentClique}^{(t)}, v' \neq v\}|$

Clearly, if CurrentClique$^{(t+1)}$ = CurrentClique$^{(t)} \backslash \{v\}$, DeltaPA$[v] = |\text{PossibleAdd}^{(t+1)}| - |\text{PossibleAdd}^{(t)}|$.

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. 6.2 (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 CurrentClique are present in OneMissing (or, better, in its projection), in fact each vertex $x \in$ CurrentClique is not connected to itself.

5.2 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 [11]): therefore, T is reduced as soon as frequent repetitions disappear.

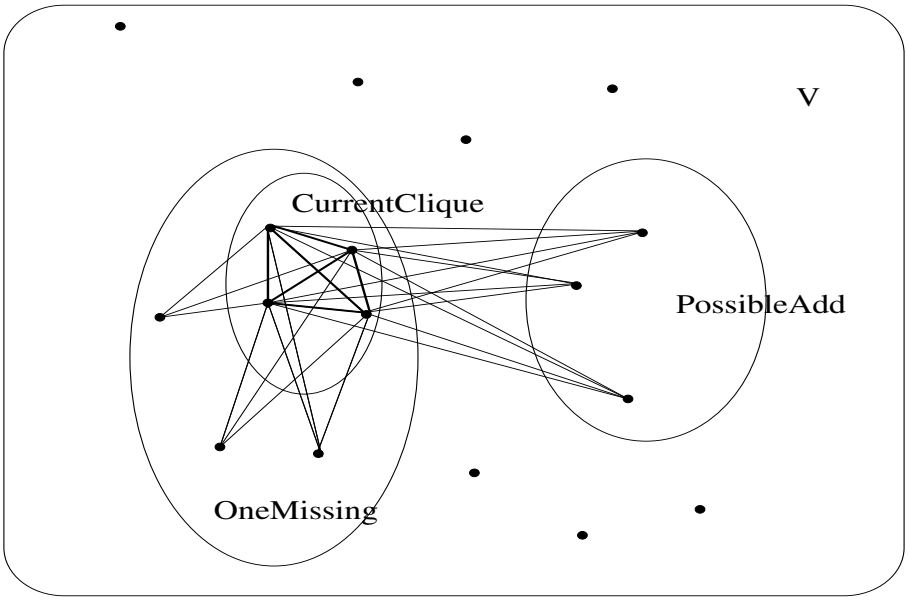

Figure 3: Subsets of V corresponding to CURRENTCLIQUE$^{(t)}$, POSSIBLEADD$^{(t)}$, and ONEMISSING$^{(t)}$.

The MEMORY-REACTION algorithm is illustrated in Fig. 4. The current clique is searched in memory. If CURRENTCLIQUE 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 [11]), cycles are discouraged by increasing T (lines 7–9).

If CURRENTCLIQUE 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 MAXSIZE, 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 MAXSIZE during the search). The value used in our tests is $B = 10 \cdot$ MAXSIZE. Increases and decreases (with a minimal change of one unit, plus upper and lower bounds) are realized by the two following functions:

$$\begin{aligned} \text{INCREASE}(T) &= \min\{\max\{T \cdot 1.1, T+1\}, n-2\} \\ \text{DECREASE}(T) &= \max\{\min\{T \cdot 0.9, T-1\}, 1\} \end{aligned}$$

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 $A = 10 \cdot B = 100$ MAXSIZE 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 [35]. First the prohibition parameter T is reset and the hashing memory structure is cleared (lines 1–2). If there are vertices that have

```
MEMORY-REACTION (CURRENTCLIQUE, T)
1     ▷search for clique CURRENTCLIQUE in the memory, get a reference Z
2     Z ← HASH-SEARCH(CURRENTCLIQUE)
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(CURRENTCLIQUE, 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 .

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 CURRENTCLIQUE in the past, a random vertex in V is selected (line 9). Data structures are updated to reflect the situation of CURRENTCLIQUE $= \emptyset$, see lines 10–14 (MISSING and MISSINGLIST are introduced in Sec. 6.2), then the selected vertex is added and the incremental update applied (lines 15–16).

6 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 $O(|N(\text{CURRENTCLIQUE})|)$, 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 a suitable *hashing* function is applied (like a sum of contributions of the different bits, where the single contribution of the updated bit has to be added or subtracted at every iteration). If the entire clique is stored with the *radix tree* method [15] 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 $|N(\text{CURRENTCLIQUE})|$ different function values. As an example, an incremental evaluation is used to update POSSIBLEADD during successive **add** moves in [20], while POSSIBLEADD is recomputed from scratch after a **drop** move, with a worst-case complexity of $O(n^2)$. Now, after a transient phase of successive

```
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    POSSIBLEADD ← V
11    ONEMISSING ← ∅
12    forall  v ∈ V
13        MISSINGLIST[v] ← ∅ ; MISSING[v] ← 0
14        DELTAPA[v] ← 0
15    CURRENTCLIQUE ← {v}
16    INCREMENTAL-UPDATE(v, ADDMOVE)
```

Figure 5: *RLS* Algorithm: routine RESTART .

add moves if CURRENTCLIQUE 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 CURRENTCLIQUE, the set POSSIBLEADD and ONEMISSING are represented with an *indicator set*, see Sec. 6.1, DELTAPA[v] with a n-dimensional array.

6.1 Indicator set

To realize some of the needed data structures with the best worst-case 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" where all contained elements are examined. Some elements can be deleted during the examination and 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*.

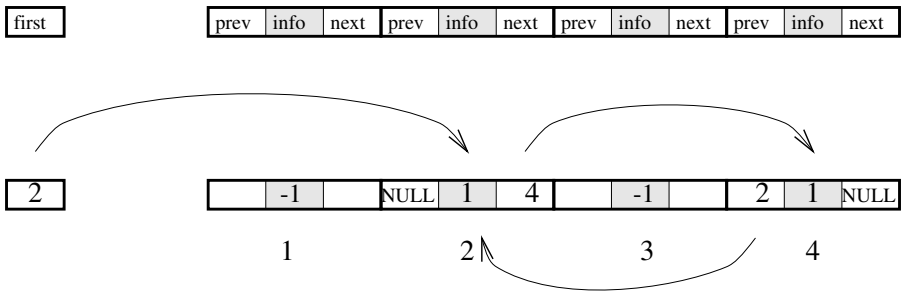

Figure 6: Indicator set structure (above) and example with n=4 and *length*=2 (below).

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 ONEMISSING (see Def. 5.2). The items to be stored are ordered pairs of integers from 1 to n (a pair is present at most once). Now, for each v there is at most one x such that $(v, x) \in$ ONEMISSING, and therefore the associated x can be stored in the above described *info* variable of the indicator set. Removal of vertex v from ONEMISSING requires decrementing DELTAPA$[x]$. After removing vertex v, the x value returned by ONEMISSING.DEL(v) is used to know which DELTAPA$[x]$ is to be decremented.

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

6.2 The INCREMENTAL-UPDATE algorithm

The data structures described in Sec. 6 are adopted to realize the sets CURRENTCLIQUE, POSSIBLEADD, and ONEMISSING. An n-dimensional array MISSING is used to record the number of missing connection to set CURRENTCLIQUE for each vertex,

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

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

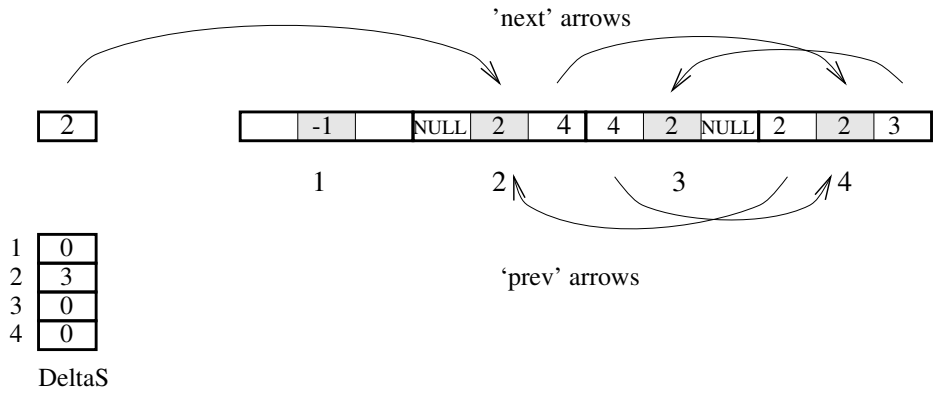

Figure 7: Data structure for OneMissing, and array DeltaPA, an example with n=4 and three vertices (2,3,4) not connected to vertex 2 in CurrentClique.

When an element is added to or dropped from CurrentClique, the data structures PossibleAdd, OneMissing, Missing and MissingList are updated through the algorithm of Fig. 8.

Let us prove 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 PossibleAdd. Clearly, this property is not satisfied by v because $(v, v) \notin E$.

The membership of $w \in$ PossibleAdd 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 CurrentClique, w will have all edges after the move. Similarly, PossibleAdd membership does not change after **drop** moves: if w was in PossibleAdd it will remain there (trivial), if w was not in PossibleAdd, then some other edge beyond (w, v) must be missing to CurrentClique 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 OneMissing membership, while the fact that MissingList$[w]$ and Missing$[w]$ 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 CurrentClique (lines 4–11). For all non-connected j, v is added to the list of missing connections and the number of missing connections to CurrentClique increases (line 6). If the number of missing connections from j to CurrentClique is one, j was in PossibleAdd before the addition and now enters OneMissing (lines 8–9). It could be added to CurrentClique if v is dropped, therefore DeltaPA$[v]$ increases. If the number of missing connections from j to CurrentClique is two, j was in OneMissing and has now to be deleted from it (lines 10–11), the value DeltaPA$[x]$ is decreased for the single vertex x to which j was not connected.

The case when a vertex is dropped is easily proved with analogous arguments. In particular, if Missing$[j]$ is zero, j transfers from OneMissing to PossibleAdd (lines 15–17), if Missing$[j]$ is one, the vertex in CurrentClique corresponding to the single missing edge is extracted from MissingList$[j]$ and j enters OneMissing (lines 19–20).

If the lists of missing connections for each vertex are not available in the structure defining

```
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             MISSINGLIST[j].INSERT(v) ; MISSING[j] ← MISSING[j] + 1
7             if MISSING[j] = 1 then
8                 POSSIBLEADD.DEL(j)
9                 ONEMISSING.INSERT(j, v) ; DELTAPA[v] ← DELTAPA[v] + 1
10            else if MISSING[j] = 2 then
11                x ← ONEMISSING.DEL(j) ; DELTAPA[x] ← DELTAPA[x] − 1
12    else
13        forall j ∈ N_Ḡ(v)
14            MISSINGLIST[j].DEL(v) ; MISSING[j] ← MISSING[j] − 1
15            if MISSING[j] = 0 then
16                x ← ONEMISSING.DEL(j) ; DELTAPA[x] ← DELTAPA[x] − 1
17                POSSIBLEADD.INSERT(j)
18            else if MISSING[j] = 1 then
19                x ← the only vertex contained in MISSINGLIST[j]
20                ONEMISSING.INSERT(j, x) ; DELTAPA[x] ← DELTAPA[x] + 1
```

Figure 8: INCREMENTAL-UPDATE routine.

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 6.1 *The incremental algorithm for updating* CURRENTCLIQUE, POSSIBLEADD *and* ONEMISSING *during each iteration of RLS has a worst case complexity of* $O(n)$. *In particular, if vertex* v *is added to or deleted from* POSSIBLEADD, *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 $deg_{\overline{G}}(v) << n$).

6.3 Update of G(POSSIBLEADD) degrees

The INCREMENTAL-UPDATE algorithm is used to assure that the *sets* POSSIBLEADD and ONEMISSING 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(POSSIBLEADD) (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 POSSIBLEADD, the corresponding degrees are incremented by 1.

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

Corollary 6.1 *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 PossibleAdd$^{(t)}$ 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 $O(deg_{\overline{G}}(v) \cdot |\text{PossibleAdd}^{(t+1)}|)$. In actual runs, because the search aims at maximizing the clique CurrentClique$^{(t)}$, the set PossibleAdd$^{(t)}$ 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 6.1 (Sec. 6.2).

6.4 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 6.2 *The memory required by the RLS algorithm is* $O(n^2 + t_{max})$.

The proof 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 MissingList 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 $(i, j) \notin E$).

The linear growth of the memory required as a function of t_{max} can be avoided with an implementation such that the hashing values of the older configurations stored, the configuration older than $2(n-1)$, are removed and the space is re-used for the more recent configurations.

7 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 MissingList is avoided and line 19 in Fig. 8 is substituted with a loop over all CurrentClique 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 6.1). 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 [30].

7.1 DIMACS challenge

An international Implementation Challenge organized by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) took 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 were presented at a DIMACS workshop in Fall of 1993, with proceedings published by the AMS [30].

A small number of instances were 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 available from the DIMACS archive by using ftp to `dimacs.rutgers.edu` (directory `pub/challenge`) or at the ULR `http://dimacs.rutgers.edu/Challenges/`. Most of the graphs are generated in [33]. Different generators can be found in [24]. An exact algorithm that motivated in part the DIMACS implementation challenge is described in [14]. A more recent branch-and-bound algorithm with state of the art results in the area of exact algorithms is presented in [37]. Of course, exact algorithms can be applied only to graphs of limited size. The 37 tasks include the following graphs (see [30] 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 [30], the user times in seconds obtained by our workstation are listed in Table 8.

r100.5	r200.5	r300.5	r400.5	r500.5
0.02	0.38	3.06	13.04	57.42

Table 8: User times for DIMACS machine benchmarks instances (Pentium II, 450 MHz clock, 384 Mb RAM, compiler: g++ -O2)

In order to assess the statistical variation, 100 runs were performed for each task, with different sequences of pseudo-random numbers (used to break ties and during restarts). The number of iterations executed on each instance was fixed to $20,000 \times n$. The experiments

on the instances with known optimal clique sizes demonstrate that most instances are solved correctly within the given limit.

The computational results are shown in Table 9. 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 [30]:

- 1) AtA (Grossman) [22]
- 2) SA plus greedy (Homer and Peinado) [27]
- 3) VHP (Gibbons, Hearn, Pardalos) [18]
- 4) SM1 (Brockington and Culberson) [13]
- 5) CLIQUEMERGE (Balas and Niehaus) [2]
- 6) ST (Soriano and Gendreau) [35]
- 7) DT (Soriano and Gendreau) [35]
- 8) PT (Soriano and Gendreau) [35]
- 9) GSD($\emptyset$) (Jagota, Sanchis, Ganesan) [28]
- 10) SSD($\emptyset$) (Jagota, Sanchis, Ganesan) [28]
- 11) $SSD_{RL}(\emptyset)$ (Jagota, Sanchis, Ganesan) [28]
- 12) RB-clique (Goldberg and Rivenburgh) [21]
- 13) tabu search and genetic hybrids (Fleurent and Ferland) [16]
- 14) A (Bar-Yehuda, Dabholkar, Govindarajan, Sivakumar) [4]
- 15) C (Bar-Yehuda, Dabholkar, Govindarajan, Sivakumar) [4]

Given the size and difficulty of some tasks, only 20 values in Table 8 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 **36** 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 case where the current best value is not obtained, the difference is of one vertex (`DSJC500.5`).

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.

8 Variants of RLS

Different algorithmic choices were investigated while developing the above presented *RLS* scheme. The variants presented in this section present less dramatic performance changes than those presented in Sec. 4. Both the computational complexity and the actual results obtained on the DIMACS benchmark were considered before defining the presented algorithm.

In particular, the algorithm obtained if the degrees in the original graph are used instead of the $G(\text{PossibleAdd})$ degrees in the Best-Neighbor routine is discussed in Sec. 8.1. The relaxation of prohibitions for selected "promising" moves (*aspiration criterion*) is considered in Sec. 8.2.

8.1 RLS without using $G(\text{PossibleAdd})$ degrees

The values of the vertex degrees in the induced subgraph $G(\text{PossibleAdd})$ 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 $G(\text{PossibleAdd})$ would have a better worst-case complexity per iteration of $O(n)$ because the $O(m)$ term caused by the updating illustrated in Sec. 6.3 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 induced subgraph $G(\text{PossibleAdd})$. In detail, the term $deg_{G(\text{PossibleAdd})}$ in Fig. 2 (lines 8–9) was replaced with $deg_{G(V)}$, whose values are calculated in the initialization part of the *RLS* algorithm.

For brevity, the computational results are not listed in this paper (they are available in the preprint [7]). As expected, the number of iterations per second always increases with respect to those obtained in Table 9. 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 $G(\text{PossibleAdd})$ degrees in the move choice is crucial only for a limited subset of the considered benchmark tasks.

8.2 RLS with aspiration

An important element of traditional Tabu Search [19] 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 $G(\text{PossibleAdd})$ 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*.

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

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

The computational results are listed in [7]. The increased algorithm complexity is not justified by the obtained performance: the maximal clique sizes obtained are not larger with respect to the version without the aspiration criterion and the CPU times are statistically comparable.

9 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.

We executed a series of experiments in order to motivate the criterion used to evaluate neighbors, the continuation of the search in the vicinity of the first local optimum found,

and the use of prohibition to enforce diversification. But we did *not* optimize some of the algorithm parameters on the given tasks, also to show what could be obtained by a simple automated on-line tuning mechanism. Of course, better results could in principle be obtained by additional performance tuning, especially if one considers only specific problem classes. For example, one could consider only random graphs of different densities and study the optimal parameter settings as a function of the graph density. This is an open research area for future investigations. An additional finding that waits for a theoretical analysis is the presence of strong correlations in the positions of good locally optimal points: searching in the vicinity of a local optimum is in many cases much more efficient than restarting from a new initial point. Exceptionally good solutions are found more easily when the search is continued from a given local optimum, while they tend to elude a direct search that starts from a random initial point.

The computational complexity per iteration of *RLS* has been analyzed in the worst case and extensive computational tests were 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 [30]), *RLS* appears to provide a significantly better performance, considering both the obtained clique sizes and the scaled CPU times utilized. For a limited time the C++ code for the *RLS* algorithm will be available for research purposes at `http://rtm.science.unitn.it/` .

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. 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). The work of the second author was partially done while visiting the International Computer Science Institute, Berkeley, CA.

On February 1st, 1998 Marco Protasi's life has been defeated by a long and terrible illness. Marco, I will miss my co-author and friend.

References

[1] G. Ausiello, P. Crescenzi, and M. Protasi, Approximate Solution of NP Optimization Problems, *Theoretical Computer Science*, **150**(1995), 1–55.

[2] E. Balas and W. Niehaus, Finding large cliques in arbitrary graphs by bipartite matching, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26. American Mathematical Society, Providence, RI, 1996, pp. 29–52.

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

[4] R. Bar-Yehuda, V. Dabholkar, K. Govindarajan, and D. Sivakumar, Randomized Local Approximations with applications to the MAX-CLIQUE problem, Technical Report, Department of Computer Science, SUNY at Buffalo, August 10, 1993.

[5] 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, *Journal of Heuristics*, **1(1)**(1995), 9–32.

[6] R. Battiti and A. Bertossi, Greedy, Prohibition, and Reactive Heuristics for Graph-Partitioning, *IEEE Transactions on Computers*, **48(4)**(1999), 361–385.

[7] R. Battiti and M. Protasi, Reactive Local Search for the Maximum Clique Problem, Technical Report TR-95-052, ICSI, Berkeley, 1995.

[8] R. Battiti and M. Protasi, Reactive Search, a history-sensitive heuristic for MAX-SAT, *ACM Journal of Experimental Algorithmics*, **2** (1997). (electronic journal available at http://www.jea.acm.org/).

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

[10] 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.

[11] R. Battiti and G. Tecchiolli, Local search with memory: Benchmarking RTS, *Operations Research Spectrum*, **17(2/3)**(1995), 67–86.

[12] M. Bellare, O. Goldreich, and M. Sudan, Free bits, PCPs and non-approximability – Toward tight results, in *Proc. 36-th Ann. IEEE Symp. on Foundations of Computer Science*, IEEE Computer Society, 1995, pp. 422-431.

[13] M. Brockington and J. C. Culberson, Camouflaging independent sets in quasi-random graphs, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26, American Mathematical Society, Providence, RI, 1996, pp. 75–88.

[14] R. Carraghan and P.M. Pardalos, An exact algorithm for the Maximum Clique Problem, *Operations Research Letters*, **9** (1990), 375–382.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[16] C. Fleurent and J. A. Ferland, Object-oriented implementation of heuristics search methods for Graph Coloring, Maximum Clique, and Satisfiability, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26, American Mathematical Society, Providence, RI, 1996, pp. 619–652.

[17] C. Friden, A. Hertz, and D. de Werra, STABULUS: A Technique for Finding Stable Sets in Large Graphs with Tabu Search, *Computing*, **42** (1989), 35–44.

[18] L. E. Gibbons, D. W. Hearn, and P. M. Pardalos, A continuous based heuristic for the maximum clique problem, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26, American Mathematical Society, Providence, RI, 1996, pp. 103–124.

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

[20] 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.

[21] M. K. Goldberg and R. D. Rivenburgh, Constructing cliques using restricted backtracking, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26, American Mathematical Society, Providence, RI, 1996, pp. 89–102.

[22] T. Grossman, Applying the INN model to the Maximum Clique problem, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26, American Mathematical Society, Providence, RI, 1996, pp. 125–146.

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

[24] J. Hasselberg, P. M. Pardalos, and G. Vairaktarakis, Test case generators and computational results for the maximum clique problem, *Journal of Global Optimization*, **3** (1993), 463–482.

[25] J. Hastad, Clique is hard to approximate within $n^{1-\epsilon}$, *Proc. 37th Ann. IEEE Symp. on Foundations of Computer Science*, IEEE Computer Society, 1996, pp. 627–636.

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

[27] S. Homer and M. Peinado, Experiments with polynomial-time approximation algorithms on very large graphs, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26, American Mathematical Society, Providence, RI, 1996, pp. 147–168.

[28] A. Jagota, L. Sanchis, and R. Ganesan, Approximately solving Maximum Clique using neural network and related heuristics, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26, American Mathematical Society, Providence, RI, 1996, pp. 169–204.

[29] 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.

[30] D. Johnson and M. Trick (Editors), *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v 26. American Mathematical Society, Providence, RI, 1996.

[31] B. Kernighan and S. Lin, An Efficient Heuristic Procedure for Partitioning Graphs, *Bell Systems Technical J.*, **49** (1970), 291–307.

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

[33] P.M. Pardalos, Construction of test problems in quadratic bivalent programming, *ACM Transactions on Mathematical Software*, **17(1)** (1991), 74–87.

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

[35] P. Soriano and M. Gendreau, Tabu search algorithms for the maximum clique problem, in *Cliques, Coloring, and Satisfiability* (D. S. Johnson and M. Trick, eds.), DIMACS Series in Discrete Mathematics and Theoretical Computer Science, v. 26, American Mathematical Society, Providence, RI, 1996, pp. 221–244.

[36] K. Steiglitz and P. Weiner, Algorithms for Computer Solution of the Traveling Salesman Problem, *Proceedings of the Sixth Allerton Conf. on Circuit and System Theory, Urbana, Illinois*, 1968, pp. 814–821.

[37] D. R. Wood, An algorithm for finding a maximum clique in a graph, *Operations Research Letters*, **21** (1997), 211–217.

Name	Time to Best Avg (S.Dev.)	Avg Iter.	Iter./Sec.	Clique Size Min	Avg (S.Dev.)	Max	BR
C125.9	0.004 (0.005)	151.805	37850.320		34 (0)		34 *
C250.9	0.029 (0.031)	1339.929	46172.413		44 (0)		44 *
C500.9	3.124 (2.993)	83303.133	27671.708		57 (0)		57
C1000.9	41.660 (45.987)	890773.239	21628.085		**68 (0)**		67
C2000.9	823.358 (844.611)	10766833.460	13099.206	77	77.575 (0.497)	**78**	75
DSJC500.5	0.194 (0.200)	2134.832	10991.432		13 (0)		14 *
DSJC1000.5	6.453 (6.937)	51146.292	8145.442		15 (0)		15 *
C2000.5	9.976 (9.021)	42372.655	4227.030		16 (0)		16
C4000.5	2183.089 (1921.273)	3891500.274	1778.785		18 (0)		18
MANN_a27	3.116 (3.224)	86846.761	24743.495		126 (0)		126 *
MANN_a45	398.770 (441.584)	5432022.407	13628.973	343	343.602 (0.510)	345	345 *
MANN_a81	2830.820 (2774.694)	11921798.495	4220.428		1098 (0)		1098
brock200_2	9.605 (10.869)	116387.712	12414.936		12 (0)		12 *
brock200_4	19.491 (22.290)	319903.360	16753.570		17 (0)		17 *
brock400_2	42.091 (91.274)	823523.324	19601.841	25	26.063 (1.760)	29	29 *
brock400_4	108.638 (104.315)	2146140.595	19700.513	25	32.423 (2.078)	33	33 *
brock800_2	4.739 (4.484)	58090.387	12652.807		21 (0)		21
brock800_4	6.696 (5.914)	81744.081	12494.204		21 (0)		21
gen200_p0.9_44	0.037 (0.045)	1865.847	50405.405		44 (0)		44 *
gen200_p0.9_55	0.016 (0.009)	772.423	47990.649		55 (0)		55 *
gen400_p0.9_55	1.204 (1.327)	35017.631	32543.732		55 (0)		55
gen400_p0.9_65	0.050 (0.032)	1869.550	35378.370		65 (0)		65
gen400_p0.9_75	0.051 (0.022)	1858.640	35281.747		75 (0)		75
hamming8-4	0.003 (0.005)	16	5340.431		16 (0)		16 *
hamming10-4	0.078 (0.051)	928.757	9310.314		40 (0)		40
keller4	0.002 (0.004)	31.126	16250.250		11 (0)		11 *
keller5	0.171 (0.167)	2990.036	16065.350		27 (0)		27
keller6	189.814 (196.006)	1173480.964	6312.307		59 (0)		59
p_hat300-1	0.018 (0.036)	245.928	13611.111		8 (0)		8 *
p_hat300-2	0.006 (0.005)	44.514	7418.333		25 (0)		25 *
p_hat300-3	0.021 (0.014)	835.117	39761.90		36 (0)		36 *
p_hat700-1	0.186 (0.210)	1707.793	10965.527		11 (0)		11 *
p_hat700-2	0.028 (0.005)	140.054	4901.682		44 (0)		44 *
p_hat700-3	0.035 (0.012)	333.297	7844.232		62 (0)		62
p_hat1500-1	30.274 (30.440)	157613.757	5113.824		12 (0)		12 *
p_hat1500-2	0.158 (0.057)	943.955	4819.865		65 (0)		65
p_hat1500-3	0.192 (0.066)	1807.351	8190.056		94 (0)		94

Table 9: 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).