Journal of Heuristics, 10: 135–152, 2004
© 2004 Kluwer Academic Publishers. Manufactured in The Netherlands.

Combining Swaps and Node Weights in an Adaptive Greedy Approach for the Maximum Clique Problem

A. GROSSO*
M. LOCATELLI†
D.I., Università di Torino, Torino, Italy
email: grosso@di.unito.it
email: locatell@di.unito.it

F. DELLA CROCE
D.A.I., Politecnico di Torino, Torino, Italy
email: federico.dellacroce@polito.it

Submitted in September 2003 and accepted by Edmund Burke in January 2004 after 1 revision

Abstract

In this work, the NP-hard maximum clique problem on graphs is considered. Starting from basic greedy heuristics, modifications and improvements are proposed and combined in a two-phase heuristic procedure. In the first phase an improved greedy procedure is applied starting from each node of the graph; on the basis of the results of this phase a reduced subset of nodes is selected and an adaptive greedy algorithm is repeatedly started to build cliques around such nodes. In each restart the selection of nodes is biased by the maximal clique generated in the previous execution. Computational results are reported on the DIMACS benchmarks suite. Remarkably, the two-phase procedure successfully solves the difficult Brockington-Culberson instances, and is generally competitive with state-of-the-art much more complex heuristics.

Key Words: maximum clique problem, adaptive search, greedy algorithm

1. Introduction

The maximum clique problem (MC) calls for finding in a given undirected graph a subset of pairwise adjacent nodes—called a *clique*—whose cardinality is as large as possible. MC is known to be NP-hard (Garey and Johnson, 1977) and hard to approximate (Bellare, Goldreich, and Sudan, 1998).

A wide range of optimization techniques has been applied to MC; for a comprehensive survey, we refer to Bomze et al. (1999). Exact branch and bound algorithms can handle up to 1000 nodes graphs but are limited to 200 nodes for difficult instances (see Östergard, 2002; Wood, 1997). Among the heuristic techniques, the so-called greedy algorithms offer great simplicity and high speed; they usually build a clique by iteratively adding nodes selected

*Author to whom all correspondence should be addressed.
†M. Locatelli acknowledges support from Progetto FIRB "Ottimizzazione non Lineare su Larga Scala".

by quick-and-dirty rules until no further additions are possible. Unfortunately, greedy algorithms are also myopic in nature and often fail to generate good solutions. However, in view of such appealing characteristics of simplicity, significant research efforts have been devoted to improve accuracy of greedy algorithms while adding minimal complexities. We consider two approaches in this line of work. Feo, Resende, and Smith (1994) and Resende, Feo, and Smith (1998) pursued the GRASP (Greedy Randomized Adaptive Search Procedure) approach. GRASP procedures alternate between a greedy phase where cliques are constructed quickly and an improvement phase where local search is applied to improve the generated solutions; in order to generate many different solutions, the selection rules are formulated such that random tie-breaking drives different runs of the greedy phase towards significantly different cliques. The neighborhood structure for MC is usually based on k-exchange moves (k nodes of a clique are replaced by k external nodes): such neighborhoods may have a quite "flat" structure for small k (i.e. very few improving solutions) whereas for large k they become very large and contain many unfeasible neighbors (possibly requiring the local search phase to cope with unfeasibilities, a complicating issue). Hence in a GRASP procedure a substantial amount of time is usually spent in the local search phase. Jagota and Sanchis (2001) recently proposed a class of algorithms called "restart-adaptive": the core of a restart-adaptive algorithm for MC is a greedy algorithm which generates cliques by means of a selection rule which is probabilistic in nature; such a greedy algorithm is repeatedly called, updating the probability distributions used by the selection rule through a learning-like mechanism. Several variants are discussed in the cited paper.

This research meets the spirit of these two approaches, aiming to develop effective procedures by combining simple greedy algorithms while keeping the overall structure as simple as possible. The work proceeds as follows.

(a) We start by considering a very basic greedy algorithm and its natural multistart application called SM[1] (see Brockington and Culberson, 1996). To overcome its major drawback, namely the construction of a clique through a "straight" unrecoverable selection of nodes, we augment it with a simple and less myopic selection rule which in many cases considerably improves performances.
(b) We consider another possible modification incorporating restarts and an adaptive mechanism (different from that of Jagota and Sanchis (2001)) aimed to progressively diversify the search. We combine this with the improved greedy algorithm of (a) in a two-phase procedure which—differently from GRASPs—incorporates no local search at all.

We test the proposed algorithms on several benchmarks from the second DIMACS challenge (Johnson and Trick, 1996); the two-phase procedure is able to successfully solve the very hard Brockington-Culberson graphs (Brockington and Culberson, 1996), and is generally competitive with state-of-the art, much more complex heuristics on all other graphs.

The paper is structured as follows. In Section 2 we develop our approach; in Section 3 we analyze the behaviour of the algorithms on "difficult" examples; in Section 4 we develop the computational study.

2. Notation and algorithms

We use the following notation. $G(V, E)$—an undirected graph with node set $V = \{1, 2, \ldots, n\}$ and edges $E \subseteq \{\{i, j\} : i, j \in V, i \neq j\}$. $N(i)$—the neighbors of node $i \in V$, $N(i) = \{j \in V : \{i, j\} \in E\}$. $D = 2|E|/[n(n-1)]$ is the density of the graph. For any subset $K \subseteq V$ of pairwise adjacent nodes, we define sets

$$C_p(K) = \{i \in V : |K \setminus N(i)| = p\} \quad p = 0, 1,$$

i.e. the set of those nodes that are non-adjacent to exactly p nodes of K. The subgraph of G induced by $C_0(K)$ contains all the nodes that can expand K to a larger clique. Such subgraph is also called the *residual graph* in literature. Finally, $\omega(G)$ will denote the size of the largest clique of G.

The very basic greedy algorithm shown below—called SM^0 in Brockington and Culberson (1996)—expands a clique K by adding a node $i \in C_0(K)$ having maximum degree in the residual graph, until no further addition is possible.

Note: in all the algorithms described in the paper, ties are broken randomly.

Procedure Greedy(K_0 : initial clique)

```
1. Set $K := K_0$;
2. **while** $C_0(K) \neq \emptyset$ **do**
3.     Select $i \in C_0(K)$ such that $|N(i) \cap C_0(K)|$ is maximum;
4.     Set $K := K \cup \{i\}$;
5. **end while**;
6. **return** $K$;
```

Procedure Greedy can be implemented with a running time bounded by $O(n^2)$: steps 3 and 4 have linear complexity and are performed at most $\omega(G)$ times; updating the residual graph implies removing one or more nodes and all their incident edges. With suitable data structures, deleting an edge from the residual graph requires constant time: since no more than $|E| = O(n^2)$ edge deletions can occur, the time bound follows. The most natural multistart applications of Procedure Greedy—called SM^k— work by running Procedure Greedy from all k-tuples of pairwise adjacent nodes. The running time of SM^k is $O(n^{2+k})$ for dense graphs; accuracy of these heuristics grows with k, and so do the running times. SM^1 runs Greedy($\{i\}$) for all $i \in V$, and is quite fast; SM^2 runs Greedy($\{i, j\}$) for all $\{i, j\} \in E$ and though it delivers better solutions, it can already require a substantial amount of CPU time on dense graphs.

Because of the very short-sighted selection rule, SM^1 and even SM^2 in many cases fail to generate optimal or near-optimal cliques. In order to design greedy-based algorithms able to deliver good results while keeping simple structure and limited CPU times, we considered two devices to be added to SM^1: swap moves and node weights.

2.1. Adding swap moves

The basic greedy selection rule—used in step 3 of Procedure Greedy—has the major drawback that it is completely myopic. Hence nodes with high residual degree but not belonging to a maximum clique can be brought into the solution, unrecoverably driving Procedure Greedy towards non-optimal cliques. In order to overcome this flaw, consider replacing step 3 with the following

3′. Select $i \in C_0(K) \cup C_1(K)$ such that $|N(i) \cap C_0(K)|$ is maximum.

The updating of K needs then to be modified as follows. If $i \in C_0(K)$ we set $K := K \cup \{i\}$ as before; if $i \in C_1(K)$ there exists a unique $l \in K$ such that $\{i, l\} \notin E$, and we set $K := K \cup \{i\} \backslash \{l\}$. The latter is termed a *swap* move; when such a move takes place K is updated to a new partial clique with equal size. Since this modified procedure is no longer monotonic, we avoid infinite loops by forcing the procedure to select $i \in C_0(K)$ by the usual greedy rule for a fixed number START_SWAP of initial iterations and also when the total number of performed swap moves reaches a threshold T. To reduce cases of useless swaps, the last node which has been swapped out from K is forbidden to reenter for the next swap move only (just like a minimal "tabu" rule).

Procedure Greedy-Swap(K_0 : initial clique)

```
 1. Set $K := K_0; n_{it} = 0$; N_SWAP = 0; LAST_SWAP = $-1$;
 2. **while** $C_0(K) \neq \emptyset$ **do**
 3.      **if** $n_{it} >$ START_SWAP **and** N_SWAP $< T$ **then**
 4.           Select $i \in [C_0(K) \cup C_1(K) \backslash \{\text{LAST\_SWAP}\}]$ such that
              $|N(i) \cap C_0(K)|$ is maximum;
 5.           **if** $i \in C_0(K)$ **then**
 6.                Set $K := K \cup \{i\}$;
 7.           **else**
 8.                Select the unique node $l \in K : \{i, l\} \notin E$;
 9.                Set $K := K \cup \{i\} \setminus \{l\}$; LAST_SWAP := $l$;
                   N_SWAP = N_SWAP + 1;
10.      **else**
11.                Select $i \in C_0(K)$ such that $|N(i) \cap C_0(K)|$ is maximum;
12.                Set $K := K \cup \{i\}$;
13.      Set $n_{it} = n_{it} + 1$;
14. **end while**;
15. **return** $K$;
```

START_SWAP is set to 5 in our experiments. The number of operations performed by Greedy-Swap is that of Greedy plus a number of operations required to execute at most T swap moves. In our implementation we used the "dynamic" threshold $T = 2|K|$ (see Section 4 for more comments about this choice). The number of iterations involving a swap is bounded by $O(n)$, as the number of Greedy-like iterations is. However, in contrast with Procedure Greedy where only (at most n) deletions from $C_0(K)$ occur, during a swap iteration both deletions and insertions in $C_0(K)$ take place. Each insertion or deletion in the residual graph has a linear cost, and the running time for Greedy-Swap depends on their total number. It does not seem to be possible inferring an $O(n)$ bound on that number: this makes one unable to invoke an $O(n^2)$ bound on the total running time for Procedure Greedy-Swap. Hence we also tested a variant of Greedy-Swap where we explicitly imposed such $O(n)$ bound, by forcing the procedure to the pure greedy behaviour when the total number of deletions and insertions in $C_0(K)$ reaches the limit value $10n$: this allows one to keep an $O(n^2)$ bound for the running time of Greedy-Swap. However, such modification had no practical impact since the number of insertion/deletion operations is usually largely below the $10n$ bound.

The introduction of the modified selection rule has important consequences on the behaviour of the algorithm: as each swap replaces a node in K, previously taken decisions can be retreated. Executing swaps can be considered as a recovering phase where the current partial solution can be replaced by an equal-sized but more promising one.

Although the swap moves may have some flavor of neighborhood search, we remark that Greedy-Swap is still greedy in nature: note that the recovering phase may not even be triggered for several iterations. Moreover the selected value of T strongly limits the number of swap moves. Nevertheless the algorithm becomes considerably less myopic, being able to "look around", intensifying the search and adjusting the trajectory it follows towards the best clique.

We call $\mathrm{SM}^k_{\mathrm{SWAP}}$ the natural multistart application of Greedy-Swap (i.e. running it from each completely connected k-tuple of nodes): only $\mathrm{SM}^1_{\mathrm{SWAP}}$ will be considered here because it is the fastest one. In computational experiments $\mathrm{SM}^1_{\mathrm{SWAP}}$ exhibits a considerably improved accuracy compared to SM^1 and even—in some cases—to SM^2, with good computation times. Still, there are hard instances on which $\mathrm{SM}^1_{\mathrm{SWAP}}$ is unable to compute optimal or even near-optimal cliques. On such instances, it is extremely difficult to grow large cliques using only information about the degree; the consequences of wrong decisions are much deeper and the swap mechanism alone is not able to drive back the heuristic along the correct trajectory.

2.2. Weighting nodes

Investigating the failures of SM^1 and $\mathrm{SM}^1_{\mathrm{SWAP}}$ on the most difficult instances we noted that, even when starting from a node i belonging to a maximum clique, other nodes not included in any maximum clique get selected by the heuristics already in the very early iterations. Even restarting the heuristic a number of times does not change much the chances of avoiding such wrong choices and discovering a good clique.

In order to insert relevant perturbations in the selection process, we formulate the following strategy. Introduce a weight w_i for all $i \in V$ and let

$$z_i = \sum_{j \in N(i) \cap C_0(K)} w_j .$$

Then replace the selection rule of Procedure Greedy by

3″. Select $i \in C_0(K)$ such that z_i is maximum.

Call Weight-Greedy the resulting modified procedure. Given a node $v \in V$, restart Weight-Greedy($\{v\}$) for a prefixed number MAXITER of times, perturbing the weights after each restart. By this mechanism we aim to progressively diversify the search, allowing different choices to be made since from the very first iterations of Weight-Greedy.

Proper computing and updating of the weights are crucial for the success of this approach. We propose the following scheme. Given the initial node v, set initally $w_i = 1$ for all $i \in V$. With the t-th restart of Weight-Greedy($\{v\}$), we get a clique $K^{[t]}$; before the $(t+1)$-th restart, update the weights by

$$w_i := \begin{cases} w_i, & \text{If } i \notin K^{[t]}, \quad \text{else} \\ w_i/2, & \text{If } i \in K^{[t]} \quad \text{and } w_i > \dfrac{1}{2^{\text{MAXWEIGHT}-1}}, \quad \text{else} \\ 0, & \text{If } i \in K^{[t]} \quad \text{and } w_i = \dfrac{1}{2^{\text{MAXWEIGHT}-1}}. \end{cases}$$

MAXWEIGHT is a predefined integer parameter.

The aim of restarting Weight-Greedy with the described weight adjustment is to stimulate diversification between consecutive restarts, allowing several different cliques being generated from the same starting node. This diversification is *soft*, in the sense that nodes repeatedly appearing in the generated cliques are discouraged to be selected again because of the weight reduction they suffer. Nodes with a large number of neighbors belonging to already generated cliques are also discouraged to be selected. However no one of such nodes is forbidden: indeed, they might be selected at later steps, when however the residual graph is small and making wrong decisions is less likely.

The trajectory followed by the different restarts of Weight-Greedy will move farther and farther from the initial one, still keeping a chance of making the correct choices (in some restart) and discarding the wrong ones. This way the correct trajectory to the maximum clique, can gradually emerge after some repeated restarts from the same initial node even if it was initially "hidden" and strongly adversed by the basic greedy rule with respect to other trajectories.

Parameters MAXITER and MAXWEIGHT should be properly calibrated. For small values of MAXWEIGHT a consistent number of weights will be quickly driven to zero, whereas a high value of MAXWEIGHT induces excessively smooth weights' refinement. Preliminary experiments showed that a reasonable choice is to set MAXWEIGHT in the range $1, \ldots, 4$ and MAXITER to a fraction of n (from $n/8$ to $n/4$).

Procedure Weight-Greedy can be implemented with $O(n^2)$ running time provided that suitable data structures are used (as for Procedure Greedy).

2.3. *The two-phase procedure*

Several restarts of Weight-Greedy are necessary for the weighting approach of the previous section to be effective, hence a substantial computational effort must be spent on each node v

selected for restarting. Using all $v \in V$ would be impractical resulting in heavy computation times; to get a faster algorithm we combine $\mathrm{SM}^1_{\mathrm{SWAP}}$ and the weighting approach into a two-phase procedure which we call Deep Adaptive Greedy Search (DAGS). We first define the algorithm formally; MAXITER and MAXWEIGHT are exogenous integer parameters already introduced in Section 2.2, δ is a real parameter in the interval (0, 1].

Procedure DAGS

```
    (Phase 1: apply $\mathrm{SM}^1_{\mathrm{SWAP}}$ and rank nodes)
 1. Set $K^{\mathrm{BEST}} := \emptyset$;
 2. for all $i \in V$ do
 3.     $K_i$ := Greedy-Swap($\{i\}$);
 4.     if $|K_i| > |K^{\mathrm{BEST}}|$ then set $K^{\mathrm{BEST}} := K_i$;
 5. end for;
 6. Compute $s_j$ = number of cliques in $K_1, \ldots, K_n$ containing $j, \forall j \in V$;
 7. Select the first $\lfloor \delta n \rfloor$ nodes with smallest $s_i$, call this set $U$;
    (End of Phase 1)
 8. for all $i \in U$ do
 9.     Set $w_1 = w_2 = \cdots = w_n = 1$;
11.     for $t := 1, \ldots,$ MAXITER do
12.         $K^{[t]}$ := Weight-Greedy($\{i\}$);
12.         if $|K^{[t]}| > |K^{\mathrm{BEST}}|$ then set $K^{\mathrm{BEST}} := K^{[t]}$;
13.         for all $j \in K^{[t]}$ do
14.             Set $w_j := w_j/2$;
15.             if $w_j < \frac{1}{2^{\mathrm{MAXWEIGHT}-1}}$ then set $w_j := 0$;
16.         end for
17.     end for
18. end for
19. return $K^{\mathrm{BEST}}$;
```

The purpose of Phase 1 in DAGS is twofold: first to generate a number of good cliques by running $\mathrm{SM}^1_{\mathrm{SWAP}}$; then to score the nodes by the number of generated cliques they belong to.

In the second phase we execute MAXITER restarts of Weight-Greedy on a subset $U \subseteq V$ of nodes, perturbing weights as in Section 2.2. Subset U is made by the $\lfloor \delta n \rfloor$ nodes with least scores, because we want to focus this phase of the search on nodes which have been (maybe erroneously) frequently discarded by the greedy rules, and skip “basins of attraction” of already heavily explored regions of the graph. Again, a sensible tuning of the parameters is required; our tests showed that choosing an excessively high value of δ does not result in improvements of solution quality, but only in marginal improvements in robustness, i.e. the best clique is detected a larger number of times, at the expense of increased computation times. We usually found that the best values are in the interval [0.1, 0.3]. We limited our experiments to the described scoring criterion; other rules can be developed, and are open to investigation.

There are both similarities and significant differences between DAGS and the approaches of Feo, Resende, and Smith (1994) and Jagota and Sanchis (2001). Analogously to GRASPs, DAGS is structured in two phases: the first phase is aimed to perform a preliminary search as well as to gather some global information to be passed to the second phase. The second phase—which performs the "deep" search—is completely greedy and no local search is involved at all. Our second phase is different in spirit from the Adaptive-Restart approach of Jagota and Sanchis (2001). In Jagota and Sanchis (2001) weights w_i are assigned to nodes and an $i \in C_0(K)$ is selected to be added to the partial clique K with probability $w_i / \sum_{j \in C_0(K)} w_j$, and all nodes in $C_0(K)$ have a chance to be selected; the weights are subjected to an additive modification and may be lowered or raised. Here in DAGS, only nodes with maximum z_i in $C_0(K)$ are considered for selection at each step; weights w_i are decreased by an exponential rule, with a different impact on the values of z_i.

The running time of DAGS is given by the sum of the two phases running times. The running time of phase 1 is $O(n^3)$ due to the application of $\text{SM}^1_{\text{SWAP}}$ (by the $O(n^2)$ version of Greedy-Swap, see Section 2.1). The running time of phase 2 is given by $\lfloor \delta n \rfloor \cdot$ MAXITER $\cdot O(n^2)$ (where $O(n^2)$ is the running time of Weight-Greedy). For MAXITER set to a fraction of n, the resulting time bound for DAGS is then $O(n^4)$.

3. Analysis on some "difficult" instances

In order to better understand the behaviour of the proposed heuristics, we analyze the following examples, based on a class of instances suggested by Busygin (2002b). Let $H(V_H, E_H)$ be a graph defined by

$$\begin{aligned} V_H &= \{a, b, c, d, e, f, g\}, \\ E_H &= \{\{a, b\}, \{a, c\}, \{a, d\}, \{e, f\}, \{e, g\}, \{f, g\}\} \end{aligned}$$

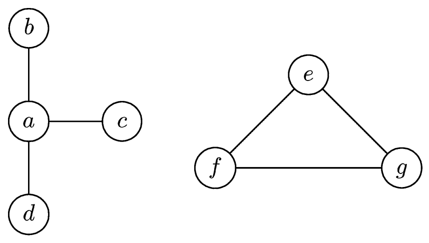
(see figure 1(a)).

A $7m$-nodes instance $H[m]$ with a unique $3m$-nodes maximum clique is composed of m copies $H_1, \dots, H_m$ of H completely connected to each other. Figure 1(b) shows the adjacency matrix of $H[3]$. Such instances are claimed to be difficult in Busygin (2002b). As a matter of fact they are difficult mainly for SM^k-like heuristics which iteratively add nodes to the clique, whereas they are quite vulnerable to "drop" heuristics where low-degree nodes are removed from the original graph. Anyway, these instances may be useful in order to see how an SM^k-like heuristic is likely to fail. The Brockington-Culberson instances (Brockington and Culberson, 1996) are much harder, but obtaining compact examples is not easy.

Note that H contains a unique 3-nodes clique $\{e, f, g\}$, but node a has the highest degree. Such a node is able to defeat the greedy strategy. Let i_r be node i in H_r. The unique maximum clique in $H[m]$ is

$$K^* = \{e_r, f_r, g_r : 1 \le r \le m\}.$$

Note that, if a partial clique K is built by nodes in $H_1, H_2, \dots, H_r$, the nodes of $H_{r+1}, \dots, H_m$ will stay in $C_0(K)$.



(a)

$$\begin{array}{c|ccccccccccccccccccccc}
 & a_1 & b_1 & c_1 & d_1 & e_1 & f_1 & g_1 & a_2 & b_2 & c_2 & d_2 & e_2 & f_2 & g_2 & a_3 & b_3 & c_3 & d_3 & e_3 & f_3 & g_3 \\
\hline
a_1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
b_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
c_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
d_1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
e_1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
f_1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
g_1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
b_2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
c_2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
d_2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
e_2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
f_2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
g_2 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
a_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
b_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
c_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
d_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
e_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
f_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
g_3 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0
\end{array}$$

(b)

Figure 1. (a) Structure of H. (b) Adjacency matrix for $H[3]$.

We first analyze the behaviour of SM^1. Due to the symmetry of the graph we consider the application of Procedure Greedy starting from nodes $a_1, \ldots, g_1$ only (the same consideration holds for Greedy-Swap further on). Procedure Greedy obviously fails in detecting the maximum clique when starting from one of a_1, b_1, c_1, d_1. Now consider the behaviour of Greedy($\{v\}$) where $v \in \{e_1, f_1, g_1\}$. We analyze the case $v = e_1$ as the other two cases behave in the same way. Greedy will correctly select the partial clique $\{e_1, f_1, g_1\}$, since node f_1 and g_1 will exhibit maximum degree in the residual graph. But then only nodes $a_r, r = 2, \ldots, m$, will exhibit maximum degree in the new residual graph, and this will drive Greedy to select the wrong node. As a consequence SM^1 is guaranteed to fail on the $H[m]$ graphs; also it can be proved that SM^k will fail on any $H[m]$ with $m \geq k$.

Now consider how Procedure $\mathrm{SM}^1_{\mathrm{SWAP}}$ works on $H[m]$. Suppose Greedy-Swap($\{v\}$) starts with $v \in \{a_1, b_1, c_1, d_1\}$: then, as no swap is allowed in the first iteration, a_1 and one of b_1, c_1, d_1 will be in the clique after the first iteration because of the high residual degree and will never be removed, hence Greedy-Swap($\{v\}$) will terminate with a suboptimal clique.

Now consider Greedy-Swap($\{v\}$) starting with $v \in \{e_1, f_1, g_1\}$. Greedy-Swap is clearly able to build the partial clique $\{e_1, f_1, g_1\}$. Then all nodes of $H_2, \dots, H_m$ will belong to the residual graph, and nodes $a_2, \dots, a_m$ will exhibit maximum degree. Hence a wrong node (say a_2) will be brought into the clique. Now, note that

$$b_2, c_2, d_2 \in C_0(K), \quad e_2, f_2, g_2 \in C_1(K),$$

and, for all such nodes, $|N(v) \cap C_0(K)|$ will have the strictly maximum value. Hence nodes e_2, f_2, g_2 have a chance to be swapped into the clique. Let p be the probability of such a swap to occur (in this case $p = 0.5$). If the swap occurs, then e_2, f_2, g_2 will be brought into K since they will have largest degree in the residual graph and the next wrong node selection will occur only when processing the nodes of another copy H_r. By iterating the analysis on each H_r, we can establish that the maximum clique can be retrieved by Greedy-Swap with probability p^{m-1} (recover must occur on each of the $m - 1$ copies involved). The latter being the probability for Greedy-Swap of succeeding when starting from a correct node, Procedure $\text{SM}^1_{\text{SWAP}}$ has a failure probability of $p_f = (1 - p^{m-1})^{3m}$. For $p = 0.5$ and $m = 10$, this gives $p_f \approx 0.94$; hence $\text{SM}^1_{\text{SWAP}}$ has a chance to solve $H[m]$, but it becomes harder and harder as m increases. Also, one can worsen things for $\text{SM}^1_{\text{SWAP}}$ on similar examples by adding nodes that rise the degree of a in H.

Now we illustrate the behaviour of Procedure DAGS on graph $H[3]$, with the adjacency matrix of figure 1. We assume $\delta = 0.15$ and MAXWEIGHT $= 1$ (hence possible weights $\{0, 1\}$). The unique maximum clique is known to be $K^* = \{e_1, f_1, g_1, e_2, f_2, g_2, e_3, f_3, g_3\}$. After phase 1, the sequence of ranked nodes is

$$(b_1, g_3, c_1, f_3, e_1, f_1, g_1, e_3, g_2, f_2, e_2, c_2, b_2, d_3, b_3, c_3, d_2, d_1, a_3, a_2, a_1).$$

With $\delta = 0.15$, phase 2 considers nodes b_1 and g_3. Iterations starting from node b_1 will fail to find the maximum clique. Iterations starting from node g_3 will evolve like depicted in figure 2.

Note that, for $t = 1$, Weight-Greedy is defeated at the third iteration, where node a_2 is selected instead of g_1. The weight adjustment allows to lower the z_i's of those nodes leading the algorithm away from the optimum. For $t = 2$, we still have a probability of failure, for example at the third step (where a_1 or a_2 may also be selected), but chances of making the optimal choice are sensibly improved by the weight update. For instance, in a 100-runs test on $H[10]$, DAGS gave a fraction of failures equal to 0.38 in contrast with the 0.94 expected for $\text{SM}^1_{\text{SWAP}}$.

4. Computational experience

Implementation details

Procedures SM^1, SM^2, $\text{SM}^1_{\text{SWAP}}$ and DAGS were implemented in *C* and run on a Pentium IV processor (1.4 GHz clock, 256 MB RAM) with Linux operating system. For each considered instance of MC we allowed 20 runs of the procedure, recording the best result (and average

$$t = 1, \qquad w_{a_1}, \ldots, w_{g_3} = 1.$$

$$\begin{array}{lll}
K = \{g_3\}, & z_{e_3} = z_{f_3} = 15 & \Rightarrow K := K \cup \{e_3\} \\
K = \{g_3, e_3\} & z_{f_3} = 14 & \Rightarrow K := K \cup \{f_3\} \\
K = \{g_3, e_3, f_3\} & z_{a_1} = z_{a_2} = 10 & \Rightarrow K := K \cup \{a_2\} \\
K = \{g_3, e_3, f_3, a_2\} & z_{b_2} = z_{c_2} = z_{d_2} = 7 & \Rightarrow K := K \cup \{d_2\} \\
K = \{g_3, e_3, f_3, a_2, d_2\} & z_{a_1} = 3 & \Rightarrow K := K \cup \{a_1\} \\
K = \{g_3, e_3, f_3, a_2, d_2, a_1\} & z_{b_1} = z_{c_2} = z_{d_1} = 0 & \Rightarrow K := K \cup \{d_1\} \\
K = \{g_3, e_3, f_3, a_2, d_2, a_1, d_1\} & &
\end{array}$$

$$t = 2, \qquad \begin{array}{l} w_{a_1}, w_{d_1}, w_{a_2}, w_{d_2}, w_{e_3}, w_{f_3}, w_{g_3} = 0, \\ \text{all other } w_i = 1. \end{array}$$

$$\begin{array}{lll}
K = \{g_3\}, & z_{e_3} = z_{f_3} = 11 & \Rightarrow K := K \cup \{e_3\} \\
K = \{g_3, e_3\} & z_{f_3} = 10 & \Rightarrow K := K \cup \{f_3\} \\
K = \{g_3, e_3, f_3\} & z_{a_1} = z_{e_1} = z_{f_1} = z_{g_1} = z_{a_2} = 7 & \Rightarrow K := K \cup \{f_1\} \\
K = \{g_3, e_3, f_3, f_1\} & z_{e_1} = z_{g_1} = 6 & \Rightarrow K := K \cup \{g_1\} \\
K = \{g_3, e_3, f_3, f_1, g_1\} & z_{e_1} = 5 & \Rightarrow K := K \cup \{e_1\} \\
K = \{g_3, e_3, f_3, f_1, g_1, e_1\} & z_{a_1} = z_{e_2} = z_{f_2} = z_{g_2} = 2 & \Rightarrow K := K \cup \{e_2\} \\
K = \{g_3, e_3, f_3, f_1, g_1, e_1, e_2\} & z_{f_2} = z_{g_2} = 1 & \Rightarrow K := K \cup \{f_2\} \\
K = \{g_3, e_3, f_3, f_1, g_1, e_1, e_2, f_2\} & z_{g_2} = 0 & \Rightarrow K := K \cup \{g_2\} \\
K = \underline{\{g_3, e_3, f_3, f_1, g_1, e_1, e_2, f_2, g_2\}} & &
\end{array}$$

Figure 2. Solution of the example.

and worst when different) and the average CPU times (in all the tests only small deviations from average times were observed).

We implemented the graph structure by storing each $N(v)$, $v \in V$ as bitstrings; this is commonly used in order to save storage and allowed us to handle large, dense graphs. The residual node set $C_0(K)$ is represented as a bitstring as well. The residual degree of a node $v \in C_0(K)$ is recomputed from scratch by $|N(v) \cap C_0(K)|$ in Procedure Greedy. The weighting scheme used in DAGS is implemented as follows: since only a discrete set of MAXWEIGHT weights $1, 1/2, \ldots, 1/2^{\text{MAXWEIGHT}-1}, 0$, is available for each node, we partition V in weight classes W_q, $q = 0, \ldots,$ MAXWEIGHT and compute the z_i's by

$$z_i = \sum_{r=0}^{\text{MAXWEIGHT}-1} \frac{|N(i) \cap C_0(K) \cap W_r|}{2^r},$$

where the set intersections are computed by fast bit-string operations; updating a weight w_j simply implies moving j from one class to another.

Admittedly, this simple implementation makes Procedures Greedy and Weight-Greedy run in $O(n^3)$ time, missing the best theoretical time bounds because of the (weighted) degree recomputation. This also makes SM^k and DAGS $O(n^{3+k})$ and $O(n^5)$ algorithms, respectively. In practice however bitstring manipulation is highly optimized, and in most cases allows to get very good computation times. The following experiment justifies our implementation choice. We ran our best $O(n^3)$ implementation of SM^1 on three dense ($D = 0.9$) graphs with sizes $n = 500$, 1000 and 2000 respectively; to further speed up this SM^1, we arranged it to work on the sparse complement graphs instead of the original ones. Our bitstring-based $O(n^4)$ SM^1 ran about 8 times faster in all three cases; hence this implementation can be recommended on most instances. A different situation is observed for large and extremely dense ($n > 1000$, $D > 0.99$) graphs (like the so-called `MANN-a45` instance, see below): in such cases the complement graph is exceptionally sparse, and the above mentioned $O(n^3)$ implementation runs much faster. Hence we decided to keep the bitstring structures, just remarking that there is room for improving running times on very dense graphs.

Evaluation of the heuristics

We tested the algorithms on several families from the benchmarks proposed in the second DIMACS challenge (Johnson and Trick, 1996): `p-hat`, `san`, `sanr`, `C`, `brock`, `gen`, `MANN`, `hamming`, `keller`. The `C` graphs are random graphs, the `brock` and `gen` graphs are developed in Brockington and Culberson (1996) and Sanchis and Jagota (1996) respectively, and have a known maximum clique embodied in a randomly generated (with different techniques) graph; the `MANN` instances are Steiner Triple graphs, and the `keller` graphs are from Keller's conjecture on tiling graphs.

We first discuss testing of $\text{SM}^1_{\text{SWAP}}$ compared to SM^1, SM^2. The results in Tables 1 and 2 illustrate the superiority of $\text{SM}^1_{\text{SWAP}}$ over the two other procedures.

As expected, the fast but myopic SM^1 heuristic fails to reach the optimal (or best known) value on 25 of the 58 instances; SM^2 improves accuracy and reliability but at high computational prices, especially on large, dense graphs. The test of SM^2 on `MANN-a45` was aborted after a single run required more than 5 hours; similarly, we abandoned testing of SM^2 on `keller6` and `C2000.9`. The $\text{SM}^1_{\text{SWAP}}$ heuristic, incorporating the modified greedy rule of Section 2.1, strongly outperforms SM^1 and SM^2: on most instances it is as accurate and reliable as SM^2 is, with much smaller computation times. Note, however, that, in a few cases (`C2000.5`, `brock200-1`, `brock200-2`, `brock800-1`), SM^1 works on average slightly better than $\text{SM}^1_{\text{SWAP}}$, namely no theoretical dominance of the modified greedy rule can be inferred. All best values were reached by $\text{SM}^1_{\text{SWAP}}$ on the `p-hat`, `san` and `sanr` families within a hundred of seconds; failures are reported on `keller6` (a very large graph, $n = 3361$) and `MANN-a45` (large and dense: $n = 1035$, $D > 0.99$). The dense random graphs `C`n`.9` present difficulties, but $\text{SM}^1_{\text{SWAP}}$ still improves its best result over SM^2. Note that for instances `C1000.9` and `p-hat-1000-3` the respective best-known value (68 for both) was only recently established in Battiti and Protasi (2001) and Burer, Monteiro, and Zhang (2002).

On some instances—mainly from `gen` and `C` families—we observed that increasing the swap threshold T up to $10|K|$ does not improve the best results (with the only exception of `C2000.9`, for which a 77-nodes clique was found) but makes $\text{SM}^1_{\text{SWAP}}$ more reliable, improving the average values at the expense of linearly proportional higher computation times.

Table 1. Performance of SM^1, SM^2, SM^1_{SWAP}, `p-hat`, `san` and `sanr` families.

Test				SM^1		SM^2		SM^1_{SWAP}	
Name	n	D	$\omega(G)$	CLIQUE	CPU(s)	CLIQUE	CPU(s)	CLIQUE	CPU(s)
`p-hat300-1`	300	0.244	8	8	0.1	8	0.3	8	0.1
`p-hat300-2`	300	0.489	25	25	0.1	25	2.9	25	0.6
`p-hat300-3`	300	0.744	36	36 (34.95, 34)	0.1	36	9.1	36	0.9
`p-hat500-1`	500	0.253	9	9	0.1	9	1.7	9	0.3
`p-hat500-2`	300	0.489	36	36	0.2	36	21.4	36	2.5
`p-hat500-3`	500	0.752	≥50	49 (48.95, 48)	0.4	50 (49.95, 49)	68.2	50	3.9
`p-hat700-1`	700	0.748	11	11 (10.50, 9)	0.6	11	5.3	11	0.8
`p-hat700-2`	700	0.498	44	44 (43.95, 43)	0.6	44	82.9	44	6.5
`p-hat700-3`	700	0.748	≥62	62 (61.75, 61)	1.3	62	278.9	62	10.1
`p-hat1000-1`	1000	0.245	≥10	10	0.2	10	16.5	10	1.9
`p-hat1000-2`	1000	0.490	≥46	46 (45.95, 46)	1.4	46	266.8	46	14.1
`p-hat1000-3`	1000	0.744	≥68	65 (64.30, 64)	3.2	67 (66.45, 66)	967.1	68 (67.85, 67)	24.6
`p-hat1500-1`	1500	0.253	12	12	0.6	12	68.9	12 (11.75, 11)	5.7
`p-hat1500-2`	1500	0.506	≥65	64	5.4	65	1519.2	65	56.8
`p-hat1500-3`	1500	0.754	≥94	92 (91.40, 91)	12.2	94 (93.35, 93)	5453.2	94	97.8
`san200-0.7-1`	200	0.700	30	30	0.1	30	1.5	30	0.2
`san200-0.7-2`	200	0.700	18	18 (17.40, 17)	0.1	18	1.2	18 (17.90, 17)	0.2
`san200-0.9-1`	200	0.900	70	70	0.1	70	7.4	70	0.6
`san200-0.9-2`	200	0.900	60	60 (59.90, 58)	0.1	60	5.7	60	0.5
`san200-0.9-3`	200	0.900	44	44 (42.00, 37)	0.1	44	4.7	44	0.5
`san400-0.5-1`	400	0.500	13	13	0.1	13	3.3	13	0.4
`san400-0.7-1`	400	0.700	40	40	0.2	40	17.2	40	1.3
`san400-0.7-2`	400	0.700	30	30	0.2	30	13.6	30	1.0
`san400-0.7-3`	400	0.700	22	19 (18.00, 17)	0.1	22	11.1	22 (21.55, 17)	0.8
`san400-0.9-1`	400	0.900	100	100	0.5	100	79.3	100	3.0
`san1000`	1000	0.502	15	15	0.5	15	92.3	15	4.3
`sanr200-0.7`	200	0.697	18	18	0.1	18	1.1	18	0.2
`sanr200-0.9`	200	0.898	42	42 (41.20, 41)	0.1	42	5.2	42 (41.80, 41)	0.5
`sanr400-0.5`	400	0.501	13	13 (12.65, 12)	0.1	13	3.5	13 (12.95, 12)	0.4
`sanr400-0.7`	400	0.700	21	21 (20.15, 20)	0.1	21	11.7	21	0.9

CLIQUE = best(average, worst) solution, CPU = average CPU time in seconds.

Finally, accordingly with the literature, random graphs with density about 0.5 are relatively easy, and even SM^1 is regularly successful on large instances (see `C2000.5`).

We attacked the hard instances with the DAGS procedure; a set of preliminary experiments led us to set $\delta = 0.15$, MAXITER $= n/8$, MAXWEIGHT $= 2$. DAGS was compared with three state of the art heuristics: QUALEX-MS by Busygin (2002), RLS by Battiti and Protasi

Table 2. Performance of SM^1, SM^2, SM$^1_{\text{SWAP}}$, other families.

Test				SM^1		SM^2		SM$^1_{\text{SWAP}}$	
Name	n	D	$\omega(G)$	CLIQUE	CPU(s)	CLIQUE	CPU(s)	CLIQUE	CPU(s)
gen200-p0.9-44	200	0.900	44	40 (39.70, 39)	0.1	43 (41.20, 40)	5.0	44 (41.10, 40)	0.5
gen200-p0.9-55	200	0.900	55	55 (51.35, 47)	0.1	55	5.3	55	0.5
gen400-p0.9-55	400	0.900	55	51 (50.50, 50)	0.4	52 (51.50, 51)	58.2	53 (51.80, 51)	2.8
gen400-p0.9-65	400	0.900	65	53 (50.85, 59)	0.3	64 (61.80, 58)	58.5	65 (57.85, 50)	2.7
gen400-p0.9-75	400	0.900	75	53 (50.75, 49)	0.3	66 (63.30, 59)	60.9	75 (55.20, 51)	2.9
C125.9	125	0.898	$\geq$34	34	0.1	34	0.9	34	0.2
C250.9	250	0.899	$\geq$44	44 (43.15, 42)	0.1	44	9.8	44	0.8
C500.9	500	0.900	$\geq$57	55 (53.85, 53)	0.6	56 (55.95, 55)	121.0	56 (55.95, 55)	4.5
C1000.9	1000	0.901	$\geq$68	65 (63.65, 63)	3.9	67 (65.75, 65)	1636.6	68 (65.95, 65)	27.6
C2000.5	2000	0.500	$\geq$16	16	3.5	16	895.0	16 (15.95, 15)	29.5
C2000.9	2000	0.900	$\geq$78	73 (72.10, 71)	30.1	–	–	76 (75.45, 74)	246.7
C4000.5	4000	0.500	$\geq$18	17 (16.95, 16)	24.3	–	–	18 (17.05, 17)	232.5
MANN-a27	378	0.990	126	126	0.9	126	176.1	126	7.3
MANN-a45	1035	0.996	345	344 (343.70, 343)	41.7	–	–	344 (343.75, 343)	293.2
hamming10-4	1024	0.829	$\geq$40	36	2.2	36	743.1	40	15.2
brock200-1	200	0.745	21	21 (20.80, 20)	0.1	21	1.5	21 (20.75, 20)	0.2
brock200-2	200	0.496	12	11	0.1	12	0.3	11 (10.85, 10)	0.1
brock200-3	200	0.605	15	14 (13.95, 13)	0.1	15	0.7	14	0.1
brock200-4	200	0.658	17	16	0.1	17	0.9	16	0.1
brock400-1	400	0.478	27	24	0.1	25	16.1	25 (24.45, 24)	1.1
brock400-2	400	0.749	29	24	0.1	29	16.0	25 (24.70, 24)	1.1
brock400-3	400	0.748	31	24	0.1	31	15.8	25	1.1
brock400-4	400	0.749	33	24	0.1	33	16.1	25 (24.65, 24)	1.1
brock800-1	800	0.649	23	21	0.5	21	92.8	21 (20.75, 20)	3.6
brock800-2	800	0.651	24	21 (20.05, 20)	0.5	21	94.5	21 (20.75, 20)	3.7
brock800-3	800	0.649	25	21 (20.40, 20)	0.5	22	92.9	22 (21.70, 20)	3.6
brock800-4	800	0.650	26	21 (20.15, 20)	0.5	21	93.4	21 (20.60, 20)	3.6
keller5	776	0.751	27	27	0.8	27	186.6	27	5.1
keller6	3361	0.818	$\geq$59	55	104.3	–	–	57 (55.70, 55)	792.0

Tests with – were suspended because of excessive CPU time.

(2001) and GRASP (Resende, Feo, and Smith, 1998). In the following we briefly present relevant features of the algorithms and reasons for this choice.

QUALEX-MS is an improvement of the QSH algorithm presented in Busygin, Butenko, and Pardalos (2002). In QUALEX-MS, a greedy procedure constructs cliques by selecting nodes similarly to Procedure Weight-Greedy; node weights are computed on the basis of the coordinates of stationary points of nonlinear programs derived from the Motzkin-Straus nonlinear formulation of MC (Motzkin and Straus, 1965). Hence it is interesting to compare this approach with DAGS, where node weights are determined by a simple combinatorial, adaptive method. As a remarkable feature QUALEX-MS shows the best performance in the literature on the instances of the brock family, which are generally considered to be among the most difficult ones for combinatorial algorithms (indeed they were generated with a technique aimed to defeat greedy approaches, see Brockington and Culberson, 1996 for details): in Busygin (2002) they are classified as "definitely hard for existing combinatorial algorithms and possibly even intractable for any combinatorial approach".

RLS (Reactive Local Search) is a purely combinatorial algorithm which greedily constructs cliques, then evolve them by local search; local optima are also subjected to a so-called "plateau search", where a swap-based neighborhood is extensively searched. RLS also incorporates a number of features like prohibitions (tabu rules), restart (abandoning the current search and starting from a different point) and reactive memory (some search history is used to generate good restart points). The parameters involved in the procedure are self-tuned by a kind of feedback, hence saving extensive testing for calibration. Among the purely combinatorial algorithms RLS reaches very good performances, being also able to improve the best values produced by many algorithms of the second DIMACS challenge. Hence we are interested in comparing it with DAGS, which still keeps much of the structure of a simple greedy approach. Note, however, that only a subset (of significantly hard instances) was tested in Battiti and Protasi (2001).

The comparison with GRASP is justified by the similarities with the two-phases approach of DAGS. No results on the DIMACS benchmarks are reported in Feo, Resende, and Smith (1994) and Resende, Feo, and Smith (1998), but the recent code documented in Resende, Feo, and Smith (1998) is available on the web. We tested it with the same parameter setting discussed in that paper, but with the limit on the number of iterations raised to 100,000. As for DAGS, we allowed 20 runs on each tested instance, reporting best, average and worst solutions and the average computation times.

The results for QUALEX-MS and RLS are from the papers cited above. The CPU time for QUALEX-MS are directly comparable to ours, since the processors are practically identical. For RLS computation times, we have to estimate the CPU time as follows. According to the DIMACS timing benchmarks, our processor is about 4.5 faster than RLS' one; in Battiti and Protasi (2001) the average time to best (as usual for metaheuristics) and time per iteration (with $20000 \times n$ iterations in each run) are reported, together with the average number of iterations per second, for each tested instance. Being interested in the total running time, we estimated it by $(20000 \times n)/(4.5 \times \text{(Iter./sec)})$.

For the comparison we mainly focus on "hard" instances i.e. those which are not solved to the best known value by $\mathrm{SM}^1_{\mathrm{SWAP}}$ in all runs. Table 3 compares DAGS, QUALEX-MS, RLS and GRASP on such instances.

Table 3. Performance of DAGS, QUALEX-MS, RLS and GRASP (hard instances).

Test		DAGS		QUALEX-MS		RLS		GRASP	
Name	$\omega(G)$	CLIQUE	CPU(s)	CLIQUE	CPU(s)	CLIQUE*	CPU(s)*	CLIQUE	CPU(s)
`p-hat1000-3`	≥68	68 (67.85, 67)	112.3	65	32	–	–	68	371.2
`p-hat1500-1`	12	12 (11.75, 11)	31.1	12	95	12	1303	11	36.6
`san200-0.7-2`	18	18 (17.90, 17)	0.3	18	<1	–	–	18 (16.55, 15)	5.1
`san400-0.7-3`	22	22 (21.70, 19)	2.0	18	2	–	–	21 (18.80, 17)	15.4
`sanr200-0.9`	42	42 (41.85, 41)	0.9	41	<1	–	–	42	19.7
`gen200-p0.9-44`	44	44 (41.15, 40)	0.9	42	<1	44	18	44 (41.95, 41)	18.4
`gen400-p0.9-55`	55	53 (51.80, 51)	7.2	51	2	55	54	53 (52.25, 52)	55.1
`gen400-p0.9-65`	65	65 (55.40, 51)	7.3	65	2	65	50	65 (64.30, 63)	54.0
`gen400-p0.9-75`	75	75 (55.20, 52)	7.8	75	2	75	50	74 (72.30, 69)	56.5
`C500.9`	≥57	56 (55.85, 55)	13.5	55	4	57	80	56	126.4
`C1000.9`	≥68	68 (65.95, 65)	148.2	64	27	68	205	67 (66.10, 65)	241.2
`C2000.9`	≥78	76 (75.40, 74)	1824.0	72	215	78	678	75 (74.30, 73)	728.7
`C4000.5`	≥18	18 (17.50, 17)	3229.0	17	2345	18	9993	18 (17.75, 17)	729.6
`MANN-a45`	345	344 (343.95, 344)	1921.3	342	17	345	337	336 (334.50, 334)	471.7
`brock200-1`	21	21	0.4	21	1	–	–	21	7.8
`brock200-2`	12	12	0.1	12	<1	12	72	12	2.2
`brock200-3`	15	15	0.1	15	1	–	–	14	4.0
`brock200-4`	17	17 (16.80, 16)	0.3	17	<1	17	53	17	5.2
`brock400-1`	27	27 (25.35, 24)	2.8	27	2	–	–	25	23.4
`brock400-2`	29	29 (28.10, 24)	2.8	29	3	29	91	25	23.8
`brock400-3`	31	31 (30.70, 25)	2.8	31	2	–	–	31 (26.20, 25)	23.2
`brock400-4`	33	33	2.8	33	2	33	90	25	23.8
`brock800-1`	23	23 (20.95, 20)	16.6	23	18	–	–	21	50.0
`brock800-2`	24	24 (20.80, 20)	16.8	24	18	21	281	21	51.5
`brock800-3`	25	25 (22.20, 21)	17.0	25	18	–	–	22 (21.85, 21)	53.3
`brock800-4`	26	26 (22.60, 20)	16.9	26	18	21	284	21	51.8
`keller6`	≥59	57 (56.40, 56)	12326	53	1291	59	2366	55 (53.50, 53)	1677.8

*: for RLS, CLIQUE = best result over 100 runs, CPU = total running time estimated from Battiti and Protasi (2001), see text; intances with – not reported in Battiti and Protasi (2001).

Times for QUALEX-MS are taken directly from Busygin (2002).

Aside from the comparison, we note that in most cases DAGS improves accuracy over $\mathrm{SM}^1_{\mathrm{SWAP}}$, with a reasonable increase in the computation times. The improvement in quality is dramatic on the instances of the very difficult `brock` family. This validates the two stage approach and the proposed use of weights.

The comparison on the `brock` family shows that for all the 12 instances the optimal solutions were generated by DAGS within the 20 runs, each run taking on average about 15 seconds for the large $n = 800$ cases. Here, DAGS outperforms RLS, which seems unable to reach the optimal values within its much higher allowed running time; even taking into account all the 20 runs for DAGS, the total time does not exceed the average time of RLS. DAGS is also highly competitive with QUALEX-MS best results. This suggests that some other simple, combinatoric weighting rules might be worth to investigate in this framework. DAGS is significantly slower on large, dense graphs like `C2000.9. MANN-a45` and `keller6`—note however that `MANN-a45` is an extremely dense graph for which faster implementations of the algorithms can be developed, as we remarked previously. On such instances DAGS exhibits intermediate accuracy between RLS (which finds the best known values) and QUALEX-MS, which delivers low-quality results. On the `gen` graphs DAGS is able to find optimal values in 3 cases out of 4, outperforming QUALEX-MS (2 out of 4) but not RLS. GRASP seems to be competitive on some families, such as `gen` and `C`, but fails on instances from `brock`, `MANN` and `keller` families.

Although we do not show detailed tables, we mention that DAGS is clearly able to solve (already in the first phase, by $\mathrm{SM}^1_{\mathrm{SWAP}}$) all the "easy" instances in the `p-hat`, `san` and `sanr` families, outperforming RLS with respect to the running time and QUALEX-MS with respect to accuracy.

For what concerns the adaptive-restart approach by Jagota and Sanchis, no test on the DIMACS collection is reported in Jagota and Sanchis (2001), where instead many classes of instances were generated by the so-called SimParD(r) method—documented in Sanchis and Jagota (1996). In our tests, performed on analogous but independently generated instances, DAGS delivered much better results than those reported in Jagota and Sanchis (2001). Conservatively speaking, the parameter settings for the generator were identical to those in Jagota and Sanchis (2001), but not necessarily the random instances; hence this comparison may not be fully reliable. Still, we remark that some classes were extremely challenging for our algorithms: this is somewhat surprising, since most works about MC use the DIMACS collection as a well-established test bench. We do not know whether these classes of instances are definitely hard or they may be vulnerable to some other kind of attack (for example, local search); if the former holds, they should be considered as an interesting benchmark for future research. Detailed results on these tests are available upon request from the authors.

Acknowledgments

The authors wish to thank Dr. S. Busygin for fruitful discussion and for providing the code for generating the $H[m]$ instances, and Dr. R. Esposito for his valuable help during the development of the computer codes.

References

Battiti, R. and M. Protasi. (2001). "Reactive Local Search for the Maximum Clique Problem." *Algorithmica* 29(4), 610–637.

Bellare, M., O. Goldreich, and M. Sudan. (1998). "Free Bits, PCPs, and Nonapproximability—Towards Tight Results." *SIAM Journal on Computing* 27(3), 804–915.

Bomze, I.M., M. Budinich, P.M. Pardalos, and M. Pelillo. (1999). "The Maximum Clique Problem." In D.Z. Du and P.M. Pardalos (eds.), *Handbook of Combinatorial Optimization*, Suppl. vol. A, pp. 1–74.

Brockington, M. and J.C. Culberson. (1996). "Camouflaging Independent Sets in Quasi-Random Graph." In Johnson and Trick.

Burer, S., R.D.C. Monteiro, and Y. Zhang. (2002). "Maximum Stable Set Formulations and Heuristics Based on Continous Optimization." *Mathematical Programming* 94, 137–166.

Busygin, S. (2002a). "A New Trust Region Technique for the Maximum Clique Problem." Report submitted, available at http://www.busygin.dp.ua.

Busygin, S. (2002b). "A Simple Clique Camouflaging Against Greedy Maximum Clique Heuristics." Report available at http://www.busygin.dp.ua.

Busygin, S., S. Butenko, and P.M. Pardalos. (2002). "A Heuristic for the Maximum Independent Set Problem Based on Optimization of a Quadratic Over a Sphere." *Journal of Combinatorial Optimization* 21(4), 111–137.

Feo, T.A., M.G.C. Resende, and S.M. Smith. (1994) "A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set." *Operations Research* 46(5), 860–878.

Garey, M. and D.S. Johnson. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman & Co.

Jagota, A. and L.A. Sanchis. (2001). "Adaptive, Restart, Randomized Greedy Heuristics for Maximum Clique." *Journal of Heuristics* 7, 565–585.

Johnson, D.S. and M. Trick (eds.). (1996). "Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge." DIMACS Series, vol. 26, American Mathematical Society.

Motzkin, R.S. and E.G. Straus. (1965). "Maxima for Graphs and a New Poof of a Theorem of Turan." *Canadian Journal of Mathematics* 17(4), 533–540.

Östergård, P.R.J. (2002). "A Fast Algorithm for the Maximum Clique Problem." *Discrete Applied Mathematics* 120, 197–207.

Resende, M.G.C., T.A. Feo, and S.H. Smith. (1998). "Algorithm 786: FORTRAN Subroutines for Approximate Solution of the Maximum Independent Set Problem Using GRASP." *ACM Transactions on Mathematical Software* 24, 386–394.

Sanchis, L. and A. Jagota. (1996). "Some Experimental and Theoretical Results on Test Case Generators for the Maximum Clique Problem." *INFORMS Journal on Computing* 8(2), 87–102.

Wood, D.R. (1997). "An Algorithm for Finding a Maximum Clique in a Graph." *Operations Research Letters* 21(5), 211–217.