Reactive and dynamic local search for Max-Clique, an experimental comparison.

Roberto Battiti and Franco Mascia
Dept. of Computer Science and Telecommunications
University of Trento, Via Sommarive, 14 - 38050 Povo (Trento) - Italy
e-mail of the corresponding author: mascia@dit.unitn.it

March 29, 2007

Abstract

This paper presents results of an ongoing investigation about how different algorithmic building blocks contribute to solving the Maximum Clique problem. We consider greedy constructions, plateau searches, and more complex schemes based on dynamic penalties and/or prohibitions, in particular the recently proposed technique of Dynamic Local Search and the previously proposed Reactive Local Search. In addition we consider in detail the effect of the low-level implementation choices on the CPU time per iteration. We present experimental results on randomly generated graphs with different statistical properties, showing the crucial effects of the implementation, the robustness of different techniques, and their empirical scalability.

1 Reactive search for maximum clique

Reactive Search, see [4, 5] for seminal papers, advocates the use of *machine learning* to automate the parameter tuning process and make it an integral and fully documented part of the algorithm. Learning is performed on-line, and therefore *task-dependent and local properties* of the configuration space can be used. In this way a single algorithmic framework maintains the flexibility to deal with related problems through an internal feedback loop that considers the previous history of the search, see also [7] for a proposal for learning domain-specific backtracking-based algorithms.

The Maximum Clique problem in graphs (MC for short) is a paradigmatic combinatorial optimization problem with relevant applications [13], including information retrieval, computer vision, and social network analysis. Recent interest includes computational biochemistry, bio-informatics and genomics, see for example [12, 9]. The problem is NP-hard and strong negative results have been shown about its approximability [10], making it an ideal testbed for search heuristics.

Let $G = (V, E)$ be an 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 problem asks for a clique of maximum cardinality.

A Reactive Local Search (RLS) algorithm for the solution of the Maximum-Clique problem is proposed in [3, 6]. 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. The performance obtained in computational tests appears to be significantly better with respect to all algorithms tested at the the second DIMACS implementation challenge (1992/93)[1].

Recently, a stochastic local search algorithm (DLS-MC) is developed in [15]. It is based on a clique expansion phase followed by a plateau search after a maximal clique is encountered. Diversification uses vertex *penalties* which are dynamically adjusted during the search, a "forgetting" mechanism decreasing the penalties is added, and vertex degrees are not considered in the selection. The authors report a very good performance on the DIMACS instances after preliminary extensive optimization phase to determine the optimal penalty delay parameter for each instance. While the number of iterations (additions or deletions of nodes to the current clique) is in some cases larger than that of competing techniques, the small complexity of each iteration when the algorithm is realized through efficient supporting data structures leads to smaller overall CPU times.

The initial motivation of this work is threefold. First, we want to investigate how the different algorithmic building blocks contribute to effectively solving max-clique instances corresponding to random graphs with different statistical properties. In particular, the investigation considers the effects of using the vertex degree information during the search, starting from simple to more complex techniques.

Second, we want to assess how different implementations of the supporting data structures affect CPU times. For example, it may be the case that larger CPU times are caused by using a high-level language implementation w.r.t. low-level "pointer arithmetic". Having available the original software simplified the starting point for this analysis.

Third, the DIMACS benchmark set (developed in 1992) has been around for more than a decade and there is a growing risk that the desire to get better and better results on

[1] http://dimacs.rutgers.edu/Challenges/

the same benchmark will bias the search of algorithms in an unnatural way. Furthermore many of the contained instances are not sufficiently challenging for current machines. We therefore decided to concentrate the experimental part on two classes of random graphs, chosen to assess the effect of degree variability on the effectiveness of different techniques.

2 Algorithmic building blocks of increasing complexity

In local search algorithms for MC, the basic moves consist of the addition to or removal of single nodes from the current clique. A swap of nodes can be trivially decomposed into two separate moves. The local changes generate a search trajectory $X^{\{t\}}$, the current clique at different iterations t. Two sets are involved in the execution of basic moves: the set of the *improving neighbors* PossibleAdd which contains nodes connected to all the elements of the clique, and the set of the *level neighbors* OneMissing containing the nodes connected to all *but* one element of the clique, see Fig. 1. The various simple building blocks considered are named following the *BasicScheme* - *CandidateSelection* structure. The *BasicScheme* describes how the greedy expansion and plateau search strategies are combined, possibly with prohibitions or penalties. The *CandidateSelection* specifies whether the vertex degree information is used during the selection of the next candidate move. If it is used, there are two possibilities: of using the static node degree in G or the dynamic degree in the subgraph induced by the PossibleAdd set.

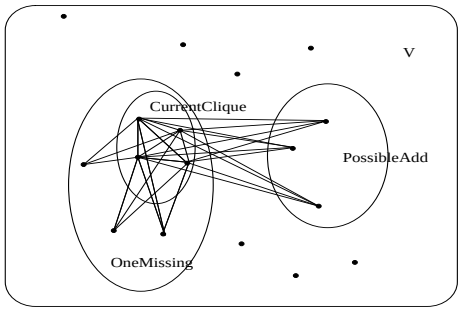

Figure 1: Neighborhood of current clique.

2.1 Repeated expansions

The starting point for many schemes is given by expansion of a clique after starting from an initial seed vertex. At each iteration the next vertex to be added can be chosen from the PossibleAdd set through different levels of "greediness" when one considers the vertex degrees:

- **Exp-Rand.** The node is selected at random among the possible additions. When a maximal clique is is encountered one restarts from a random node. The pseudo-code is shown Fig. 2.

```
1. Exp-Rand (maxIterations)
2.   iterations ← 0
3.   while iterations < maxIterations do
4.     C ← random v ∈ V
5.     expand (C)

6. expand (C)
7.   while PossibleAdd ≠ ∅ do
8.     C ← C ∪ random v ∈ PossibleAdd
9.     iterations ← iterations + 1
```

Figure 2: Greedy expansion algorithm.

- **Exp-StatDegree.** At each iteration, a random node is chosen among the candidates having the highest degree in G. The expand sub-routine in Fig 2 is modified by substituting line 8 with:
  $C \leftarrow C \cup \{$ *random* $v \in$ PossibleAdd such that $deg_G(v)$ *is maximum*$\}$.

- **Exp-DynDegree.** In this version, the selection of the candidate is not based on the degree of the nodes in G, but on the degree in PossibleAdd. This greedy choice will maximize the number of nodes remaining in PossibleAdd after the last addition. Line 8 becomes:
  $C \leftarrow C \cup \{$ *random* $v \in$ PossibleAdd such that $deg_{\text{PossibleAdd}}(v)$ *is maximum*$\}$.

2.2 Expansion and plateau search

This algorithm alternates between a greedy expansion and a plateau phase, choosing between the possible candidate nodes with different ways to consider the vertex degrees:

- **ExpPlat-Rand.** During the *expansion* phase, new vertices are chosen randomly from PossibleAdd and moved to the current clique. When PossibleAdd is empty and therefore no further expansion is possible, the *plateau* phase starts. In this phase, a node belonging to the *level neighborhood* OneMissing is swapped with the only node not connected to it in the current clique. The *plateau* phase does not increment the size of the current clique and it terminates as soon as there is at least an element in the PossibleAdd set, or if no candidates are available in OneMissing. As it is done in [15], nodes cannot be selected twice in the same plateau phase. In order to avoid infinite loops, the number of plateau searches is limited to *maxPlateauSteps*.

  Starting from Exp-Rand, the base algorithm is adapted to deal with the alternation of the two phases, see Fig. 3. Let us note that, if plateau returns with PossibleAdd $\neq \emptyset$ then a new expansion is tried as described in line 5-7. The iterations are incremented by 2 during a swap because it is counted as a deletion followed by an addition.

- **ExpPlat-StatDegree.** This algorithm is a modified version of ExpPlat-Rand (Fig. 3) with the static degree selection during the expansion and the plateau.

```
1.  ExpPlat-Rand (maxIterations, maxPlateauSteps)
2.  ⌈ iterations ← 0
3.  | while iterations < maxIterations do
4.  | ⌈ C ← random v ∈ V
5.  | | while PossibleAdd ≠ ∅ do
6.  | | ⌈ expand (C)
7.  | ⌊ ⌊ plateau (C, maxPlateauSteps)

8.  plateau (C, maxPlateauSteps)
9.  ⌈ count ← 0
10. | while PossibleAdd = ∅ and OneMissing ≠ ∅
11. | and count < maxPlateauSteps do
12. | ⌈ C ← C ∪ random v ∈ OneMissing
13. | | remove from C the node not connected to v
14. | | iterations ← iterations + 2
15. | ⌊ ⌊ count ← count + 1
```

Figure 3: ExpPlat-Rand. algorithm, alternating between expand and plateau phases.

- **ExpPlat-DynDegree.** This algorithm is the same of ExpPlat-StatDegree, apart from the selection based on the dynamic degree during the expansion phase.

2.3 Algorithms based on penalties or prohibitions

More complex schemes can be obtained by using diversification strategies to encourage the search trajectories to visit unexplored regions of the search space. These methods are particularly effective for "deceptive" instances [8], where the sub-optimal solutions attract the search trajectories.

- **ExpPlatProhibition-Rand.** A simple diversification strategy can be obtained by prohibiting selected moves in the neighborhood. In detail, after a node is added or deleted from the current clique, the algorithm prohibits moving it for the next T iterations. Prohibited nodes cannot be considered among the candidates of expansion and plateau phases. When all the moves are prohibited a restart is performed.

- **DLS-MC.** To achieve diversification during the search, penalties are assigned to vertices of the graph [15]. The algorithm alternates between expansion and plateau phases. Selection is done by choosing the best candidate among the set of the nodes in the neighborhood having minimum penalty.

  When the algorithm starts, the penalty value of every node is initialized to 0 and when no further expansion or plateau moves are possible, the penalties of nodes belonging to the clique are incremented by one. All penalties are decremented by one after *pd* (*penalty delay*) restarts, see [15] for additional details and results.

- **RLS.** This algorithm alternates between expansion and plateau phases, like DLS-MC, but it selects the nodes among the non-prohibited ones which have the highest degree in PossibleAdd. The prohibition time is adjusted reactively depending on the search history. In the "history" a fingerprint of each configuration is saved in a hash-table. Restarts are executed only when the algorithm cannot improve the current configuration within a fixed number of iterations, see [6] for details.

  To allow for a comparison between different amount of "greediness" in the node selection, a modification of RLS is introduced (called RLS-StatDegree) which uses the static degree instead of the dynamic degree selection.

The research presented in this paper considers two kinds of changes to the original RLS version. The first changes are algorithmic and influence the search trajectory, while the second one refers only to the more efficient implementation of the supporting data structures, with no effect on the dynamics.

The algorithmic changes are the following ones. In the previous version the search history was cleared at each restart, now, in order to allow for a more efficient diversification, the entire search history is kept in memory.

Having a longer memory caused the parameter T to explode on some specific instances characterized by many repeated configuration during the search. If the prohibition becomes much larger than the current clique size, after a maximal clique is encountered and one node has to be extracted from the clique, all other nodes will be forced to leave the clique before the first node is allowed to enter again. This may cause spurious oscillations in the clique membership which may prevent discovering the globally optimal clique.

An effective way to avoid the above problem is to put an upper-bound MAX_T equal to a proportion of the current estimate of the maximum clique. More specifically, the upper-bound is set to $|\text{Clique}| * 0.5$.

3 Computational experiments

The computational experiments, presented in this paper are on two classes of random graphs, and are aimed at comparing the different algorithmic building blocks and their impact on average number of steps required to find the maximum clique, as well as the cost per single iteration.

To ensure that hard instances are considered in the test, a preliminary study investigates the empirical hardness as a function of the graph dimension.

3.1 Benchmark graphs

Performance and scalability tests are made on two different classes of random graphs:

Binomial random graphs A binomial graph $GIL(n, p)$, belonging to Gilbert's model $\mathcal{G}(n, p)$ is constructed by starting from n nodes and adding up to $\frac{n(n-1)}{2}$ undirected edges, independently with probability $0 < p < 1$. See [2] for generation details.

Preferential Attachment Model A graph instance $PAT(n, d)$, of the preferential attachment model,

introduced in [1] is built by starting from a single node and adding successively the remaining nodes. The edges of the the newly added nodes are connected to d existing nodes, with preferential attachment to nodes having a higher degree, i.e., with probability proportional to the number of edges present between the existing nodes.

In binomial graphs, the degree distribution for the different nodes will be peaked on the average value, while in the preferential attachment model, the probability that a node is connected to k other nodes decreases following a power-law i.e. $P(k) \sim k^{-\gamma}$ with $\gamma > 1$.

In the experiments, the graphs are generated using the NetworkX library [11], for a number of nodes ranging from 100 to 1500. Because of the hardness of MC, the optimal solutions of large instances cannot be computed and one must resort to the *empirical maximum*. The *empirical maximum* considered in the experiments is the best clique that RLS is able to find in 10 runs of 5 million steps each. In no case DLS-MC with *pd* equal to 1 is able to find bigger cliques for the same number of iterations. The sizes of the empirical maximum cliques in the various graphs are listed in Table 1.

Nodes	$GIL(n, 0.3)$	$PAT(n, n/3)$
100	6	13
200	7	19
300	8	25
400	8	31
500	8	37
600	8	42
700	9	48
800	9	54
900	9	57
1000	9	60
1100	10	64
1200	10	70
1300	10	74
1400	10	79
1500	10	86

Table 1: Best empirical maximum cliques in the graphs.

The algorithms are tested against our data set, to compute the empirical distribution function of the iterations needed to find the *empirical maximum*. The maximum number of steps per iteration is set to 10 million and each test is repeated on the same graph instance 100 times. For the algorithms having a plateau phase, *maxPlateauSteps* is set to 100.

We count as one iteration each add- or drop-move executed on the clique. DLS-MC code was modified to count the steps in this manner, to be able to make comparisons with the other algorithms. The CPU time spent by each iteration is measured on our reference machine, having one Xeon processor at 3.4 GHz and 6 GB RAM. The operating system is a Debian GNU/Linux 3.0 with kernel 2.6.15-26-686-smp. All the algorithms are compiled with g++ compiler with "-O3 -mcpu=pentium4".

Fig. 4 and Fig. 5 summarize with standard box-and-whisker plots the medians, the quartiles, and the outliers of the iterations by ExpPlat-Rand. Fig. 4 shows that there are some instances which are significantly harder than others. The sawtooth trend of the plot is due to the fact that ExpPlat-Rand needs on average more iterations to solve instances of the Gilbert model corresponding to the increase of the expected clique size in Table 1. Instances become then easier when the number of nodes increases and the maximum clique remains of the same size while the number of optimal cliques increases. This is confirmed also by all other algorithms considered.

The sawtooth behavior is hardly visible in Fig 5 because of the different granularity of the cliques dimension with respect to the graph sizes considered.

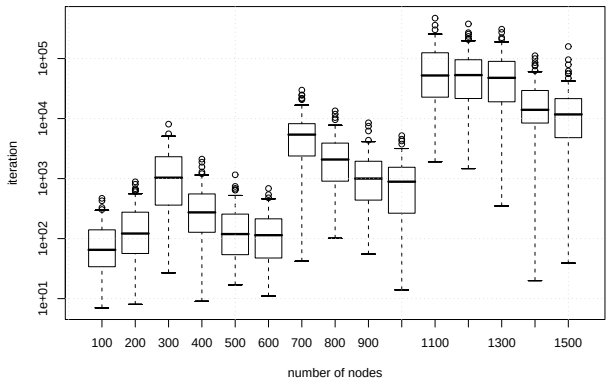

Figure 4: Iterations of ExpPlat-Rand to find the *empirical maximum* clique in $GIL(n, 0.3)$. Y axis is logarithmic.

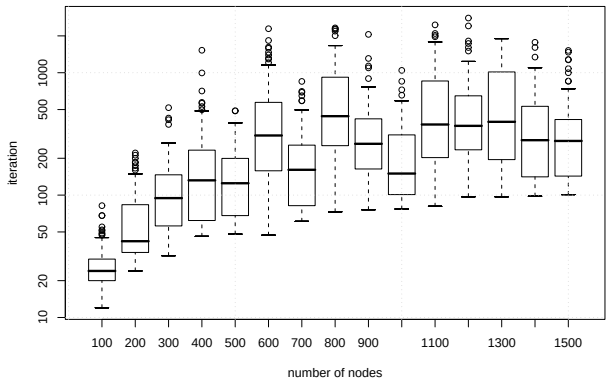

Figure 5: Iterations of ExpPlat-Rand to find the *empirical maximum* clique in $PAT(n, n/3)$. Y axis is logarithmic.

3.2 Implementation details and cost per iteration of RLS

The total computational cost for solving a problem is of course the product of the number of iterations times the cost of each iteration. More complex algorithms like RLS risk that the higher cost per iteration is not compensated by a sufficient reduction in the total number of iterations. This section is dedicated to exploring this issue.

The original implementation [6] focused on the algorithm and the appropriate data structures but did not optimize low-level implementation details.

The implementation of the supporting data structures of the new version has many improvements: i) the management of the dynamic memory, used for storing the config-

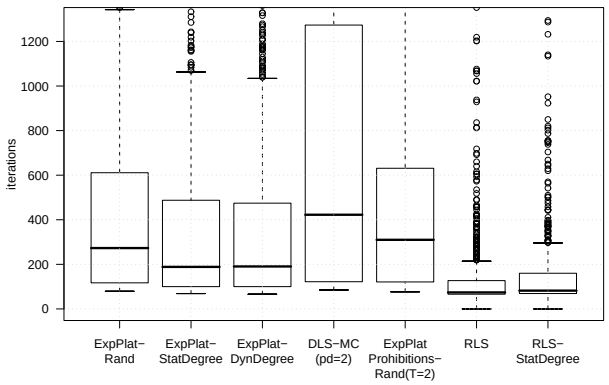

Figure 6: Iterations to find the *empirical maximum* clique in $PAT(1100, 366)$. Results for the most significant algorithms are reported.

uration fingerprints in the history, which is not allocated when needed as in the first version, but rather pre-initialized and shared among the steps of the actual run; ii) the usage of a dynamic hash table where the size is adapted to the load factor; iii) the substitution of all dynamic allocations in the functions with allocations executed at the beginning and reused throughout the run.

The speedup results are reported in Table 2. They show the improvement in the steps per seconds achieved by the new version, for two random graphs and some representative DIMACS instances. Let us note that the obtained speedup is substantial. For example the improvement for large random graphs increases with the graph dimension reaching a factor of 22 for graphs with thousand nodes (C4000.5).

Let us now consider a simple model to capture the time spent by the RLS algorithm on each iteration. Most the cost is spent on updating the data structures after each addition or deletion. After a node deletion the complexity for updating the data structures is $O(deg_{\overline{G}}(v))$, $deg_{\overline{G}}(v)$ being the degree of the just moved node v in the *complementary* graph $\overline{G}$. After a node addition the complexity is $O(deg_{\overline{G}}(v) \cdot |\text{PossibleAdd}|)$, see [6] for more details. Now, because the algorithm alternates between expansions and plateau moves, for most of the run |PossibleAdd| oscillates between 0 and 1. We can therefore make the strong assumption that |PossibleAdd| is substituted with a small constant. In both cases the dominant factor is therefore $O(deg_{\overline{G}}(v))$.

The computational complexity for using the history data-structure can be amortized to a $O(1)$ complexity per iteration. The restart operation cannot be amortized: its complexity is $O(n)$ but it is not performed regularly. On the contrary, the number of restarts highly depends on the search dynamics and on the hardness of the instance.

Under the above assumption we decided to propose an empirical model for the time per iteration which is linear in the number of node and the degree:

$$T(n, deg_{\overline{G}}) = \alpha\ n + \beta\ deg_{\overline{G}} + \gamma \qquad (1)$$

The last simplification is given by substituting the *average* node degree instead of the actual degree.

Instance	Steps per second		Speedup
	RLS [6]	*New RLS*	
gilbert_1100_0.3	11201.97	107526.88	9.598
pa_1100_366	24838.55	168350.17	6.777
C125.9	371747.21	1162790.70	3.127
C250.9	281690.14	943396.23	3.349
C500.9	165289.26	714285.71	4.321
C1000.9	80450.52	471698.11	5.863
C2000.9	27285.13	265957.45	9.747
DSJC500_5	43290.04	295857.99	6.834
DSJC1000_5	17421.60	160000.00	9.184
C2000.5	5573.20	78125.00	14.017
C4000.5	1536.78	34965.03	22.752
MANN_a27	485436.89	909090.91	1.872
MANN_a45	293255.13	425531.91	1.451
MANN_a81	14285.71	16666.67	1.166
brock200_2	109769.48	543478.26	4.951
brock200_4	147492.63	699300.70	4.741
brock400_2	103412.62	555555.56	5.372
brock400_4	105374.08	552486.19	5.243
brock800_2	33715.44	264550.26	7.846
brock800_4	33311.13	262467.19	7.879
gen200_p0.9_44	321543.41	1000000.00	3.109
gen200_p0.9_55	273224.04	943396.23	3.452
gen400_p0.9_55	210084.03	800000.00	3.808
gen400_p0.9_65	204498.98	740740.74	3.622
gen400_p0.9_75	205761.32	724637.68	3.521
hamming8-4	113122.17	568181.82	5.022
hamming10-4	46339.20	316455.70	6.829
keller4	140646.98	546448.09	3.885
keller5	55035.77	296735.91	5.391
keller6	7011.15	101626.02	14.494
p_hat300-1	57870.37	308641.98	5.333
p_hat300-2	112233.45	558659.22	4.977
p_hat300-3	171821.31	729927.01	4.248
p_hat700-1	21758.05	168350.17	7.737
p_hat700-2	49358.34	337837.84	6.844
p_hat700-3	88417.33	478468.90	5.411
p_hat1500-1	7344.84	85470.09	11.636
p_hat1500-2	13504.39	184842.88	13.687
p_hat1500-3	30459.95	282485.88	9.274

Table 2: Speed improvement on random graphs and selected DIMACS benchmark instances of the new RLS implementation.

Let us note that the above model is not precise if the size of the PossibleAdd set remains large for a sizable fraction of the iterations. For example this is the case when a large graph is extremely dense, and the clique is very large. In this case the size of the PossibleAdd set is a non-negligible factor which multiplies $deg_{\overline{G}}(v)$, impacting significantly the overall algorithm performance. This happens for the MANN instances in the DIMACS benchmark set which are not considered when fitting the above model.

The fitted model for our specific testing machine is the following:

$$T(n, deg_{\overline{G}}) = 0.0010\ n + 0.0107\ deg_{\overline{G}} + 0.0494 \qquad (2)$$

The fit residual standard errors for α, β and γ are 0.0004, 0.0009 and 0.2765 respectively.

Let us note that the cost for using the history data-structure, which is approximately included in the constant term in the above expression, becomes rapidly negligible as soon as the graph dimension and density are not very small. In fact the memory access costs approximately less than 50 nanoseconds per iteration while the total cost reaches rapidly tens of microseconds in the above instances.

3.3 Results summary

Because of lack of space, Table 3 presents the results on two specific instances of random graphs: $GIL(1100, 0.3)$ and $PAT(1100, 366)$. The choice of $GIL(1100, 0.3)$ is determined by the fact that it is empirically the most difficult instance of our data set, while $PAT(1100, 366)$ is chosen with the same number of nodes. The results are for 100 runs on 10 different instances.

Algorithm	$GIL(1100, 0.3)$		$PAT(1100, 366)$	
	Iter.	μs	*Iter.*	μs
Exp-Rand	**[92%]***	8.90	**[0%]***	3.20
Exp-StatDegree	**[0%]***	8.30	**[40%]***	3.10
Exp-DynDegree	**[10%]***	104.00	**[0%]***	10.3
ExpPlat-Rand	74697	5.80	273	3.10
ExpPlat-StatDegree	**[60%]***	5.70	189	3.10
ExpPlat-DynDegree	75577	27.20	191	7.55
DLS-MC(pd=2)	75943	5.90	423	3.20
DLS-MC(pd=4)	63467	5.90	**[99%]***	3.20
DLS-MC(pd=8)	73831	5.90	**[85%]***	3.20
ExpPlatPro.-Rand(T=2)	65994	5.80	310	3.10
ExpPlatPro.-Rand(T=4)	67082	5.90	333	3.10
ExpPlatPro.-Rand(T=8)	67329	5.80	329	3.15
RLS	47442	9.40	75	5.50
RLS-StatDegree	45259	7.30	84	4.50

Table 3: Results summary with the medians of the empirical steps distribution and the average time per iteration. (*) The algorithm is not always able to find the maximum clique; the percent of successes is reported.

From Table 3, it is clear that algorithms based only on expansions are not always able to find the maximum clique in the given iteration bound, especially on *hard instances*. The plateau phase increases dramatically the success rate.

Degree consideration is effective in the Preferential Attachment model, while in Gilbert's graphs, where the nodes tend to have similar degrees, penalty- or prohibition-based algorithms win. For example the reduction in iterations achieved by ExpPlat-StatDegree over ExpPlat-Rand on $PAT(1100, 366)$ is about 31%. On the contrary, algorithms using degree information have poorer performances on Gilbert's graphs, if compared with their completely random counterparts. For example ExpPlat-StatDegree finds the maximum clique in $GIL(1100, 0.3)$ only in the 60% of the runs.

The cost per iteration changes significantly among different instances and it also depends on the directions taken in the search-space by the algorithms so that simple model like that derived for RLS is not applicable. For example, the plateau phase does not only decrease the average number of iterations needed to find the maximum clique, but also the time spent by each single iteration. With a plateau phase, in fact, the less frequent restarts have a reduced impact on the average cost per iteration.

Table 3 shows that ExpPlat-DynDegree spends less time per iteration (factor of 3.8) than Exp-DynDegree in $PAT(1100, 366)$. The improvement is even bigger in $GIL(1100, 0.3)$ where degree-based selections are less appropriate.

In case of dynamic degree selection, the incremental update routine complexity depends also on the size of the PossibleAdd set. With plateau phases the search is longer and the PossibleAdd set is on average smaller.

RLS, which has a different and less frequent restart policy, alternates between short expansions and plateaus. Therefore the PossibleAdd set remains on average smaller than in Exp-DynDegree or ExpPlat-DynDegree and the cost per iteration is smaller.

Fig. 7 shows the average CPU time per iteration on Gilbert's graphs in log-log scale. The regression lines have a slope of 2.41, 0.92 and 0.95 for Exp-DynDegree RLS and DLS-MC(pd=2) respectively, confirming an approximate cost per iteration of ExpPlat-DynDegree growing faster than n^2, while RLS cost, even if the candidate selection is based on the dynamic degree, grows approximately linearly.

By multiplying the values in Table 3, the CPU time needed on average by RLS-StatDegree to find the maximum clique in Gilbert's hard instance is 88% of the time required by DLS-MC(pd=4). RLS-StatDegree needs 330 milliseconds while DLS-MC 374 milliseconds.

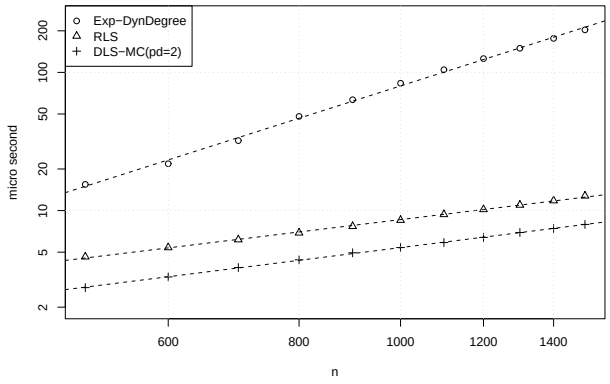

Figure 7: Empirical cost per iteration in μseconds on Gilbert's graphs. Log-log scale.

For the sake of completeness, Appendix A reports the comparison between RLS and DLS-MC on the DIMACS benchmark instances.

3.4 Penalties versus Prohibitions

As shown in Table 3, DLS-MC is not always able to find the best clique on PAT graphs while prohibition-based heuristic is always successful. Our results confirm that the penalty heuristic tends to be less robust than the prohibition-based heuristic. A significant dependency between DLS-MC performance and the choice of the *penalty delay* parameter is also discussed in [15]. Further investigations, summarized in Fig. 8, show the success rate of DLS-MC compared with that of ExpPlatProhibition-Rand for different values of the *penalty delay* and *prohibition time* parameters. The tests are on all instances of the PAT graphs of our data set.

ExpPlatProhibition-Rand is always able to find the maximum clique within 100,000 iterations, while DLS-MC fails for several *penalty delay* values even incrementing the maximum number of iterations by a factor of 10 or 100.

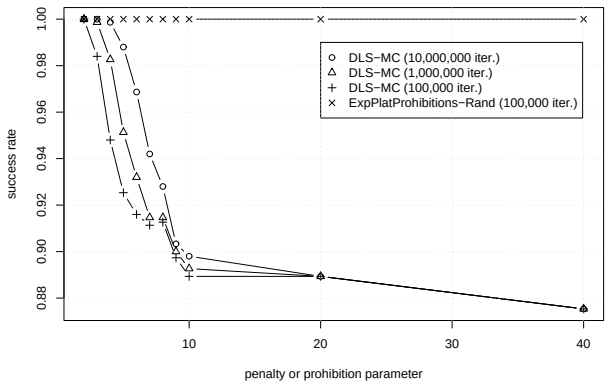

Figure 8: Success ratio of penalty- and prohibition-based algorithms on instances of the Preferential Attachment model.

4 Conclusions

The results of the current investigation show that a careful implementation of the data-structures considering also operating system services like memory allocation achieves a significant reduction of the CPU time per iteration. Because in certain cases one obtains an order of magnitude difference, this aspect is very crucial when comparing two algorithms.

The results of the tests on the two graph classes show clearly that the plateau search is necessary to find the maximum clique in hard instances and in any case to reduce the average number of iterations. The complexity added to the algorithms by the plateau search does not increase the cost per iteration. On the contrary, especially for the algorithms using the dynamic degree for candidate selections, it reduces the CPU time per iteration.

On Gilbert's graphs, where the nodes have the same degree on average, prohibition- or penalty-based algorithms perform better than pure random selections. On instances of the Preferential Attachment model, algorithms selecting the nodes using information about the degree are faster.

On the contrary, degree-based algorithms have poorer performance than random-selection algorithms in Gilbert's graphs, while prohibition- and penalty-based algorithms are disadvantageous in the Preferential Attachment model. The penalty heuristic is less robust than the prohibition heuristic, depending on the appropriate selection of the penalty value.

RLS and RLS-StatDegree, always perform better then the other algorithms. The cost per iteration of RLS-StatDegree is bigger than the one of DLS-MC, but the fewer steps needed on average to find the best cliques make it the best choice for the two graph models considered in this paper.

The software corresponding to the algorithm, benchmark graphs and the heuristically optimal values are available at request.

Acknowledgment

We thank Holger Hoos and co-authors for making available the software corresponding to the DLS-MC algorithm.

References

[1] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286:509 – 512, Oct 1999.

[2] Vladimir Batagelj and Ulrik Brandes. Efficient Generation of Large Random Networks. *Physical Review E*, 71(3):036113, Mar 2005.

[3] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. Technical Report TR-95-052, ICSI, 1947 Center St.- Suite 600 - Berkeley, California, Sep 1995.

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

[5] Roberto Battiti and Alan Albert Bertossi. Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, Apr 1999.

[6] Roberto Battiti and Marco Protasi. Reactive Local Search for the Maximum Clique Problem. *Algorithmica*, 29(4):610–637, 2001.

[7] Eric Breimer, Mark Goldberg, David Hollinger, and Darren Lim. Discovering optimization algorithms through automated learning. In *Graphs and Discovery*, volume 69 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 7–27. American Mathematical Society, 2005.

[8] Mark Brockington and Joseph C. Culberson. Camouflaging independent sets in quasi-random graphs. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26, pages 75–88. American Mathematical Society, 1996.

[9] S. Butenko and W. Wilhelm. Clique-detection models in computational biochemistry and genomics. *European Journal of Operational Research*, 2006. to appear.

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

[11] Aric Hagberg, Dan Schult, and Pieter Swart, 2004. NetworkX Library developed at the Los Alamos National Laboratory Labs Library (DOE) by the University of California. Code available at `https://networkx.lanl.gov/`.

[12] Y. Ji, X. Xu, and G.D. Stormo. A graph theoretical approach to predict common rna secondary structure motifs including pseudoknots in unaligned sequences. *Bioinformatics*, 20(10):1591–1602, 2004.

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

[14] Wayne Pullan. Phased Local Search for the Maximum Clique Problem. *Journal of Combinatorial Optimization*, 12(3):303 – 323, November 2006.

[15] Wayne Pullan and Holger H. Hoos. Dynamic Local Search for the Maximum Clique Problem. *Journal of Artificial Intelligence Research*, 25:159 – 185, Feb 2006.

A DIMACS benchmark set

The following table compares DLS-MC(PD=OPT) and RLS on a selected "snapshot" of the DIMACS benchmark set. The results presented are averages on 100 runs of 100,000,000 maximum steps each. DLS-MC(PD=OPT) is DLS-MC with the *pd* parameter set to the optimal value for each single instance as suggested in [15].

For each instance and both algorithms the table shows the average clique size with standard deviation. The CPU time's average and standard deviation and and steps' average, are computed only on successful runs. If the time is $< \epsilon$ it means that the time spent is less than 0.001 seconds thus not precisely measurable.

Let us note again that the comparison below is not fair, because in one case (DLS-MC) one reports only the time corresponding to the optimal setting of an individual *pd* parameter for each instance, while in the second case this extensive tuning phase is absent.

In most cases, a part from the "camouflaged" Brockington graphs [8], the optimal values obtained are the same. For the CPU times, in many cases the graph dimension is so small that the measure becomes difficult, in some other cases RLS has times which are larger but of the same order of magnitude. For other instances RLS CPU time is shorter, which is quite unexpected given the absence of the tuning phase.

The extended version of this paper will consider additional tests with more recent evolution of DLS-MC [14].

Instance	**DLS-MC(pd=opt)**			**RLS**		
	Clique size	*CPU(s)*	*Steps*	*Clique size*	*CPU(s)*	*Steps*
C125.9	34	$< \epsilon$	156.28	34	$< \epsilon$	176.10
C250.9	44	$< \epsilon$	939.17	44	0.002(0.004)	1351.50
C500.9	57	0.040(0.053)	41406.66	57	0.207(0.222)	149759.14
C1000.9	68	0.555(0.529)	314001.81	68	2.094(2.243)	998206.67
C2000.9	77.75(0.44)	64.703(41.958)	18947684.00	77.76(0.43)	143.736(105.392)	34891829.67
DSJC1000_5	15	0.525(0.555)	88244.35	15	0.346(0.359)	54903.68
DSJC500_5	13	0.008(0.009)	2463.03	13	0.006(0.008)	1824.12
C2000.5	16	0.581(0.535)	46208.56	16	0.566(0.492)	43260.01
C4000.5	18	163.826(163.577)	6667380.50	18	156.146(155.734)	5378762.59
MANN_a27	126	0.026(0.012)	43228.71	126	0.108(0.134)	98985.32
MANN_a45	343.96(0.20)	19.680(17.376)	13350905.00	343.98(0.14)	58.514(54.845)	22073511.88
MANN_a81	1097.92(0.27)	121.394(90.903)	23287140.00	1098	125.053(132.555)	14829085.96
brock200_2	12	0.021(0.024)	14843.86	12	0.155(0.179)	84653.65
brock200_4	17	0.036(0.036)	34364.36	17	0.395(0.419)	280505.31
brock400_2	29	0.328(0.322)	226598.48	28.89(0.63)	57.345(48.598)	27560202.26
brock400_4	33	0.040(0.040)	27968.34	33	4.537(5.017)	2477471.58
brock800_2	24	10.799(9.684)	3172619.00	21.09(0.51)	133.517(53.025)	31998039.67
brock800_4	26	4.645(4.586)	1362838.12	21.65(1.68)	160.922(121.850)	38804405.38
gen200_p0.9_44	44	0.001(0.003)	1831.32	44	0.002(0.004)	1943.39
gen200_p0.9_55	55	$< \epsilon$	452.01	55	$< \epsilon$	720.90
gen400_p0.9_55	55	0.020(0.021)	24865.05	55	0.041(0.045)	33270.25
gen400_p0.9_65	65	$< \epsilon$	792.38	65	0.002(0.004)	1586.58
gen400_p0.9_75	75	$< \epsilon$	462.97	75	0.002(0.004)	1752.26
hamming8-4	16	$< \epsilon$	26.31	16	$< \epsilon$	16.00
hamming10-4	40	0.006(0.006)	1879.89	40	0.004(0.005)	891.50
keller4	11	$< \epsilon$	32.23	11	$< \epsilon$	104.34
keller5	27	0.012(0.010)	3949.94	27	0.017(0.020)	4975.65
keller6	58.94(0.34)	113.263(93.421)	13705481.00	59	10.522(11.321)	1066758.19
p_hat300-1	8	$< \epsilon$	121.92	8	$< \epsilon$	211.90
p_hat300-2	25	$< \epsilon$	83.85	25	$< \epsilon$	58.52
p_hat300-3	36	$< \epsilon$	425.83	36	0.002(0.004)	1006.00
p_hat700-1	11	0.010(0.011)	1620.92	11	0.013(0.015)	2084.82
p_hat700-2	44	$< \epsilon$	206.12	44	0.001(0.003)	144.84
p_hat700-3	62	$< \epsilon$	384.92	62	0.002(0.004)	356.18
p_hat1500-1	12	1.554(1.356)	121763.33	12	2.123(1.986)	181487.99
p_hat1500-2	65	0.003(0.005)	596.03	65	0.007(0.008)	888.91
p_hat1500-3	94	0.004(0.006)	1355.84	94	0.008(0.007)	1589.82

Table 4: Algorithm comparison on a selected sub-set of the DIMACS benchmark instances.