REACTIVE LOCAL SEARCH FOR MAXIMUM CLIQUE: A NEW IMPLEMENTATION

Roberto Battiti and Franco Mascia

May 2007

Technical Report # DIT-07-018
Reactive local search for maximum clique: A new implementation

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

May 2007

Abstract

This paper presents algorithmic and implementation enhancements of Reactive Local Search algorithm for the Maximum Clique problem [3]. In addition, we build an empirical complexity model for the CPU time required for a single iteration, and we show that with a careful implementation of the data-structures one can achieve a speedup of at least an order of magnitude difference for large size graphs.

1 Introduction

The motivation of this work is to assess how different implementations of the supporting data structures of RLS for the Maximum Clique Problem affect the CPU times.

Let us briefly summarize the context and define our terminology.

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 [1, 3]. 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)

The new experimental results motivated also two qualitative changes of the algorithm: the fact that the entire search history is kept for the whole run duration (instead the memory was cleared at each restart in the previous version) and the fact that a new upper bound has been placed on the prohibition value \( T \).

\[ \text{http://dimacs.rutgers.edu/Challenges/} \]
The following sections present the main algorithmic changes, some optimizations of the C++ code, and finally we develop an empirical complexity model of the CPU time for a single iteration.

2 Algorithmic changes

We refer to [3] for details about the algorithm and for bibliography about different approaches for solving the maximum clique problem.

Briefly, the RLS algorithm alternates between expansion and plateau phases, selecting the nodes among the non-prohibited ones which have the highest degree in $\text{PossibleAdd}$. 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 specified number of iterations.

![RLS algorithm, from [3].](image1)

![Best-Neighbor routine, from [3].](image2)

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
Memory-Reaction (CurrentClique, T)
1. \(\triangleright\) search for clique CurrentClique in the memory, get a reference Z
2. \(Z \leftarrow\) Hash-Search(CurrentClique)
3. if \(Z \neq \text{Null}\) then
4. \(\triangleright\) find the cycle length, update last visit time:
5. \(R \leftarrow t - Z.\)LastVisit
6. \(Z.\)LastVisit \(\leftarrow t\)
7. if \(R < 2(n - 1)\) then
8. \(t_{R} \leftarrow t\)
9. return \(\max(\text{Increase}(T), \text{MAX}(T))\)
10. else
11. if the clique is not found, install it:
12. Hash-Insert(CurrentClique, t)
13. if \((t - t_{R}) > B\) then
14. \(t_{R} \leftarrow t\)
15. return \(\text{Decrease}(T)\)
16. return \(T\)

Figure 3: Memory-Reaction routine, from [3].

Restart
1. \(T = 1 ; t_{R} = t\)
2. \(\triangleright\) search for the “seed” vertex \(v\)
3. SomeAbsent \(\leftarrow\) true iff \(\exists v \in V \) with LastMoved[v] = \(-\infty\)
4. if SomeAbsent then
5. \(L \leftarrow\{w \in V : \text{LastMoved}[w] = -\infty\}\)
6. \(v \leftarrow\) random vertex with maximum \(\text{deg}_{G(V)}(v)\) in \(L\)
7. else
8. \(v \leftarrow\) random vertex \(\in V\)
9. PossibleAdd \(\leftarrow V\)
10. OneMissing \(\leftarrow \emptyset\)
11. forall \(v \in V\)
12. \(\text{MissingList}[v] \leftarrow \emptyset ; \text{Missing}[v] \leftarrow 0\)
13. \(\text{DeltaPA}[v] \leftarrow 0\)
14. CurrentClique \(\leftarrow\{v\}\)
15. Incremental-Update(v, AddMove)

Figure 4: Restart routine.

trajectory, while the second ones refer 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. Now, 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 \(\text{MAX}(T)\) equal to a proportion of the current estimate of the maximum clique. More specifically, the upper-bound \(\text{MAX}(T)\) is set to \(|\text{BestClique}|\), being, the size of the best configuration found during the search. Fig. 6 shows the explosion of the parameter \(T\) which decreases the average clique size, and Fig. 7 shows how the issue is addressed by putting an upper bound to the prohibition parameter.
Incremental-Update \( (v, \text{type}) \)
1. \( \triangleright \) Comment: \( v \) is the vertex acted upon by the last move.
2. \( \triangleright \) type is a flag to differentiate between AddMove and DropMove.
3. \( \text{LastMoved}[v] \leftarrow t \)
4. \( \text{if type=AddMove then} \)
5. \( \forall j \in \mathcal{N}(v) \)
6. \( \text{MissingList}[j].\text{Insert}(v) ; \text{Missing}[j] \leftarrow \text{Missing}[j] + 1 \)
7. \( \text{if Missing}[j] = 1 \)
8. \( \text{PossibleAdd.Del}(j) \)
9. \( \text{OneMissing.Insert}(j, v) ; \text{DeltaPA}[\ell] \leftarrow \text{DeltaPA}[\ell] + 1 \)
10. \( \text{else if Missing}[j] = 2 \)
11. \( x \leftarrow \text{OneMissing.Del}(j) ; \text{DeltaPA}[x] \leftarrow \text{DeltaPA}[x] - 1 \)
12. \( \text{else} \)
13. \( \forall j \in \mathcal{N}(v) \)
14. \( \text{MissingList}[j].\text{Del}(v) ; \text{Missing}[j] \leftarrow \text{Missing}[j] - 1 \)
15. \( \text{if Missing}[j] = 0 \)
16. \( x \leftarrow \text{OneMissing.Del}(j) ; \text{DeltaPA}[x] \leftarrow \text{DeltaPA}[x] - 1 \)
17. \( \text{PossibleAdd.Insert}(j) \)
18. \( \text{else if Missing}[j] = 1 \text{ then} \)
19. \( x \leftarrow \text{the only vertex contained in MissingList}[j] \)
20. \( \text{OneMissing.Insert}(j, x) ; \text{DeltaPA}[x] \leftarrow \text{DeltaPA}[x] + 1 \)

Figure 5: Incremental-Update routine.

3 Implementation details

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 [3] focused on the algorithm and the appropriate data structures but did not optimize low-level implementation details. For example, every time a new configuration had to be inserted in the table, the memory needed to store the element was allocated dynamically. The new implementation moved from these on-demand allocations to the more efficient allocation of a single bigger chunk of memory; the memory is used as a pool of available locations to be assigned to the elements when needed. In this way, if one million different configurations are expected, instead of doing one million times expensive system calls, one single call is done at the beginning, and new elements are later added by means of fast pointer assignments.

Moreover, the hash table resolves key collisions by means of chaining. In order to keep the frequent access operation as close to a direct access as possible, these chains have to be kept as short as possible. This has been achieved doubling the size of the hash table when the number of elements inside the table exceeds a specified load factor. More in detail, the hash table is created at the beginning with a default size, along with the corresponding pool of available locations for the expected configuration which are a fraction of the size of the table:

\[
\text{table} = \text{new}\ \text{elem}[\text{hash}\_\text{size}];
\]
\[
\text{pre}\_\text{alloc} = \text{new}\ \text{elem}[(\text{int})\text{ceil} (\text{hash}\_\text{size} \times \text{fill}\_\text{factor})];
\]

The initial \text{fill}\_\text{factor} is set to 0.6. When there are no more allocations available in the pool, and an element has to be added, the size of the hash table is doubled. If there is not enough memory for doubling the hash table and
Figure 6: Evolution of the size of the current clique during the search, and prohibition parameter $T$ explosion at approximately 33000 iterations, impacting on the clique size. The run is on a gilbert\_1100\_0.3 instance [2]; x axis is in log scale.

consequently growing the allocation pool, the size of the hash table is kept constant and the fill factor increased by 0.2. In the latter case, the access to the elements in the table is less efficient because one resorts more frequently to chaining to resolve possible collisions.

4 Empirical model for cost per iteration

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 $O(\text{deg}_{\overline{G}}(v))$, $\text{deg}_{\overline{G}}(v)$ being the degree of the just moved node $v$ in the complementary graph $\overline{G}$. After a node addition the complexity is $O(\text{deg}_{\overline{G}}(v) \cdot |\text{POSSIBLEADD}|)$, see [3] for more details. Now, because the algorithm alternates between expansions and plateau moves, for most of the run $|\text{POSSIBLEADD}|$ oscillates between 0 and 1. We can therefore make the strong assumption that $|\text{POSSIBLEADD}|$ is substituted with a small constant.

In both cases the dominant factor is therefore $O(\text{deg}_{\overline{G}}(v))$. Moreover, since the algorithm selects the nodes having the highest degree among all the candidates, the cost of $O(\text{deg}_{\overline{G}}(v))$ is kept as small as possible.

The computational complexity for using the history data-structure can be
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:

\[ T(n, \text{deg}_G) = \alpha n + \beta \text{deg}_G + \gamma \] (1)

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

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 \( \text{deg}_G(v) \), 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:

\[ T(n, \text{deg}_G) = 0.0010 n + 0.0107 \text{deg}_G + 0.0494 \mu s \] (2)
The fit residual standard errors for $\alpha$, $\beta$ and $\gamma$ 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 of the complementary graph 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.

5 Experimental Results

The speedup results reported in Table 1 show the improvement in the steps per seconds achieved by the new version, considering both algorithmic and implementation improvements, for two random graphs [2] 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).

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.

Moreover, the empirical complexity model shows how the cost for keeping the history of the search trajectory is negligible for non trivial graph instances.

References


<table>
<thead>
<tr>
<th>Instance</th>
<th>Steps per second</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RLS</strong></td>
<td><strong>New RLS</strong></td>
<td></td>
</tr>
<tr>
<td>gilbert1100.9.3</td>
<td>11201.97</td>
<td>107526.88</td>
</tr>
<tr>
<td>pa14100.366</td>
<td>24838.55</td>
<td>168350.17</td>
</tr>
<tr>
<td>C125.9</td>
<td>371747.21</td>
<td>1162790.70</td>
</tr>
<tr>
<td>C250.9</td>
<td>281690.14</td>
<td>943296.23</td>
</tr>
<tr>
<td>C500.9</td>
<td>165289.26</td>
<td>714285.71</td>
</tr>
<tr>
<td>C1000.9</td>
<td>80450.52</td>
<td>471698.11</td>
</tr>
<tr>
<td>C2000.9</td>
<td>27265.13</td>
<td>265557.45</td>
</tr>
<tr>
<td>DSIC500.5</td>
<td>43290.04</td>
<td>295857.99</td>
</tr>
<tr>
<td>DSIC1000.5</td>
<td>17421.60</td>
<td>160000.00</td>
</tr>
<tr>
<td>C2000.9</td>
<td>5573.20</td>
<td>78125.00</td>
</tr>
<tr>
<td>C4000.9</td>
<td>1536.78</td>
<td>34965.03</td>
</tr>
<tr>
<td>MANN_p27</td>
<td>485436.89</td>
<td>909090.91</td>
</tr>
<tr>
<td>MANN_p45</td>
<td>298255.13</td>
<td>425531.91</td>
</tr>
<tr>
<td>MANN_p81</td>
<td>142657.1</td>
<td>166666.67</td>
</tr>
<tr>
<td>brock200.2</td>
<td>109769.48</td>
<td>543478.26</td>
</tr>
<tr>
<td>brock200.4</td>
<td>147492.63</td>
<td>699300.70</td>
</tr>
<tr>
<td>brock400.2</td>
<td>103412.62</td>
<td>555555.56</td>
</tr>
<tr>
<td>brock400.4</td>
<td>105374.08</td>
<td>552486.19</td>
</tr>
<tr>
<td>brock800.2</td>
<td>33715.44</td>
<td>264550.26</td>
</tr>
<tr>
<td>brock800.4</td>
<td>33311.13</td>
<td>262467.19</td>
</tr>
<tr>
<td>gen200_p0.9.24</td>
<td>221234.44</td>
<td>1080000.00</td>
</tr>
<tr>
<td>gen200_p0.9.55</td>
<td>273224.04</td>
<td>943396.23</td>
</tr>
<tr>
<td>gen400_p0.9.25</td>
<td>210884.03</td>
<td>800000.00</td>
</tr>
<tr>
<td>gen400_p0.9.65</td>
<td>204498.98</td>
<td>740740.74</td>
</tr>
<tr>
<td>gen400_p0.9.75</td>
<td>205761.32</td>
<td>724637.68</td>
</tr>
<tr>
<td>hamming8.4</td>
<td>114122.17</td>
<td>508181.82</td>
</tr>
<tr>
<td>hamming10.4</td>
<td>46339.20</td>
<td>316455.70</td>
</tr>
<tr>
<td>keller4</td>
<td>139646.93</td>
<td>940485.19</td>
</tr>
<tr>
<td>keller5</td>
<td>55035.77</td>
<td>296735.91</td>
</tr>
<tr>
<td>keller6</td>
<td>7011.15</td>
<td>101626.02</td>
</tr>
<tr>
<td>p_hat300.1</td>
<td>57870.37</td>
<td>508441.98</td>
</tr>
<tr>
<td>p_hat300.2</td>
<td>112233.45</td>
<td>558659.22</td>
</tr>
<tr>
<td>p_hat300.3</td>
<td>171821.31</td>
<td>729927.01</td>
</tr>
<tr>
<td>p_hat700.1</td>
<td>21758.05</td>
<td>168350.17</td>
</tr>
<tr>
<td>p_hat700.2</td>
<td>49358.34</td>
<td>337837.84</td>
</tr>
<tr>
<td>p_hat700.3</td>
<td>88417.33</td>
<td>478468.90</td>
</tr>
<tr>
<td>p_hat1500.1</td>
<td>7344.84</td>
<td>85470.09</td>
</tr>
<tr>
<td>p_hat1500.2</td>
<td>13504.39</td>
<td>184842.88</td>
</tr>
<tr>
<td>p_hat1500.3</td>
<td>30459.95</td>
<td>282485.88</td>
</tr>
</tbody>
</table>

Table 1: Speed improvement on random graphs and selected DIMACS benchmark instances of the new RLS implementation.