ijogt ™ International Journal of GRAPH THEORY

ISSN 2320 – 6543

Volume 1 **Issue 1** **February 2013** **pp.23-31**

DISCREPANCY-FREE ORDERS AND IMPROVED ALGORITHM FOR THE MAXIMUM CLIQUE PROBLEM

Peter Zs. Schmidt [a,*]

[a] Department of Applied Mathematics, Faculty of Sciences, University of Pécs

Ifjúság útja 6., H-7627 Pécs, Hungary

∗ **Corresponding author: Peter Zs. Schmidt.** Tel.: +36-30-331-8901; fax: +36-72-574-660; e-mail: psmitt@gmail.com

[Received 28/12/2012 | Received in revised form 12/02/2013 | Accepted 13/02/2013]

ABSTRACT

An efficient new algorithm for the Maximum Clique Problem is presented within this paper. After resuming the known algorithm of Carraghan & Pardalos and the cliquer of Östergård, a special register function is introduced, which allows reordering the nodes of a graph in such a way, that Östergård's algorithm runs significantly faster than upon other usual orders of nodes. After the definition of discrepancy-free orders of the nodes, a sorting algorithm is specified on the nodes, producing a discrepancy-free order of them. Finally a combination of the sorting algorithm with Östergård's algorithm is given as a new solver for the Maximum Clique Problem. Benchmark results on DIMACS graphs are presented in table format.

Keywords: *Benchmark; DIMACS; Discrete Optimization; Graph Theory; Maximum Clique Problem; NP-Hard.*
2010 Mathematics Subject Classification: 05C69, 05C85.

1. INTRODUCTION

A *simple graph* is a topological structure$G = (V, E)$, where V is the set of *nodes* or *vertices*, and E is the set of *edges* of the graph containing the pairs of *adjacent* nodes. A simple graph is *complete* if any two of the nodes are adjacent, that is connected by an edge.

A *clique* is the set of nodes of a complete subgraph of a simple graph. The *size of a clique* is the number of its elements. A clique is *maximal* if it is not a subset of any other clique. A clique is *a maximum clique* if no other clique has more nodes than it has. Consequently, a maximum clique is the largest maximal clique of a simple graph.

Maximum clique problem (MCP) is to find a maximum clique in a given simple graph. The computational complexity of the problem is NP-hard which means that no polynomial time

algorithms are known for solving the problem. An aim of the recent research efforts on the efficient solution of MCP is to construct fast algorithms of an estimable runtime.

Detailed explanation of MCP and a survey of exact algorithms and heuristics are given by Pardalos et al. in [1] and [2]. Individual algorithms are provided by Carraghan & Pardalos [3], Östergård [4] or Kumlander [5].

Without modifying the original problem and its complexity, this paper offers a simpler approach to MCP seeking the *size* of maximum cliques instead of a maximum clique itself. Accordingly, and for the sake of preciseness, the abbreviation MCSP is used in the sequel, standing for *maximum clique size problem*.

2. RESUMPTION OF KNOWN ALGORITHMS

The first well-known general algorithm for MCP was provided by Carraghan & Pardalos [3]. Their concept is based on the enumeration of the maximal cliques. They construct a top-down search tree by starting out of the entire set of nodes. They take a pivot node and try to find a maximum clique within its neighborhood, recursively. Exhausting the graph the current pivot node is deleted from the set of vertices, and the same kind of search is conducted within the remaining nodes. Retaining the largest clique found, the search tree can be pruned, if the size of the neighbor set of selected nodes or the number of remainders do not allow finding a clique larger than the one already found.

In order to formulate the basic algorithm of Carraghan and Pardalos, let $v_n, \dots, v_1$ be an ordering of vertices of G and for every subset $U \subseteq V$ let $v_{\max_U}$ represent the element of the highest index in U. For any node $v \in V$ let $N(v) = \{u \in V | \{u, v\} \in E\}$ denote the set of nodes adjacent to v, called the neighborhood of v. The function reflecting the search concept of Carraghan and Pardalos is then

$$\underline{\mathbf{CP}(U, k, d)}:$$

if $U = \emptyset$ **then return** d

if $|U| > k - d$ **then** $k := \max\{k, \mathbf{CP}(U \cap N(v_{\max_U}), k, d+1)\}$

if $|U| > k - d$ **then** $k := \max\{k, \mathbf{CP}(U \setminus \{v_{\max_U}\}, k, d)\}$

return k

The arguments to give are the subset U in which we seek the maximum clique size; the largest size k already found; and the depth d of the recursion.The solution of MCSP is $\mathbf{CP}(V, 0, 0)$.

Numbers of improvements of the algorithm **CP** are based on more efficient pruning of the search tree. Most of them take advantage of coloring the subgraph induced by the subset U and using the color number as upper bound for the size of the largest clique in U.

Another way of speeding up **CP** is to optimize the structure of the search tree by ordering the nodes in a suitable way, putting those in front having fewer neighbors, leaving those back having more of them.

A substantially new strategy of the analysis came from Östergård [4] who builds up the search tree from the bottom, starting out from an empty set of nodes and registering the largest subclique size for each newly added node on the way. This register serves then for an efficient pruning of the tree and contains the maximum clique size in the end.

In order to conceive the procedure let $C(v_k)$ denote the size of the largest clique in the subset $\{v_k, \dots, v_1\}$. Some obvious properties of this function follow:

- $C: V \to \mathbb{N}$ is positive function;
- $C: V \to \mathbb{N}$ is monotonic function;
- $C(v_n)$is the size of maximum cliques.

Computing C is equivalent to solving MCSP. The recursive way of the computation starts with $C(v_1) = 1$. Assuming that $C(v_k)$ is already known, consider the set of nodes $U(v_{k+1}) = N(v_{k+1}) \cap \{v_k, \dots, v_1\}$. If $U(v_{k+1})$ contains a clique of size $C(v_k)$ then $C(v_{k+1}) = C(v_k) + 1$ else $C(v_{k+1}) = C(v_k)$. If C has already been defined for every elements of the subset $U \subseteq V$ Östergård's algorithm for deciding whether the set of nodes U contains a clique of size k follows:

$$\underline{\mathbf{HAS}(U,k)}:$$

if $k = 0$ **then return TRUE**

if $C(v_{\max_U}) < k$ **or** $|U| < k$ **then return FALSE**

return $\left(\mathbf{HAS}(U \cap N(v_{\max_U}), k-1)\ \mathbf{or}\ \mathbf{HAS}(U \setminus \{v_{\max_U}\}, k)\right)$

Similarly to **CP** one can try to tune Östergård's algorithm either by coloring-based pruning or by specific ordering of the nodes.

3. A NEW REGISTER FUNCTION

Aiming to obtain more efficient register function for pruning, let $c(v_k)$ denote the size of the largest clique in the subset$\{v_k, \dots, v_1\}$ containing the node v_k itself. Some obvious properties of this function follow:

- $c: V \to \mathbb{N}$ is positive function;
- $c(v_k) \leq C(v_k)$;
- If $c(v_{k+1}) \leq c(v_k)$ then$C(v_{k+1}) = C(v_k)$.

The latter statement stands on the facts that$C$ is monotonic and if $C(v_{k+1}) > C(v_k)$ then $C(v_{k+1}) = C(v_k) + 1$ which means that v_{k+1} is an element of some largest clique in$\{v_{k+1}, \dots, v_1\}$, thus $c(v_{k+1}) = C(v_{k+1}) = C(v_k) + 1 \geq c(v_k) + 1 > c(v_k)$

The computation of C is based on the recursive computation of c. The recursive procedure starts again with $C(v_1) = c(v_1) = 1$. Assuming that $C(v_k)$ and $c(v_k)$ are already known, consider again the set of nodes $U(v_{k+1}) = N(v_{k+1}) \cap \{v_k, \dots, v_1\}$. The task is now to calculate the size of the largest clique in $U(v_{k+1})$ which is based on a **CP**-variant:

$\underline{\mathbf{CP}'(U,k,d)}$:

if $U = \emptyset$ **then return** d

if $c(v_{\max_U}) > k - d$ **and** $|U| > k - d$ **then**

$k := \max\{k, \mathbf{CP}'(U \cap N(v_{\max_U}), k, d+1)\}$

if $C(v_{\max_U}) > k - d$ **and** $|U| > k - d$ **then**

$k := \max\{k, \mathbf{CP}'(U \setminus \{v_{\max_U}\}, k, d)\}$

return k

Hence $c(v_{k+1}) = 1 + \mathbf{CP}'(U(v_{k+1}), 0)$.

If $c(v_{k+1}) > C(v_k)$ then $C(v_{k+1}) = c(v_{k+1})$ else $C(v_{k+1}) = C(v_k)$.

Considering an arbitrary graph $\mathbf{CP}'$ seems still expensive, and it is indeed, without further optimization.$\mathbf{CP}'$ becomes competitive if the nodes can be ordered in such a way, that the*discrepancy* between the register functions disappears, so that they become identical. In that case one can get rid of the double registry and use only the first, efficient condition for pruning the search tree.

The existence of that discrepancy-free order will be proven and different ways of its construction will be presented in the next chapter.

4. DISCREPANCY-FREE ORDER OF THE NODES

Let $u > v$ denote that the node$u$ has higher index and so precedes the node$v$ in the order $\{v_n, \dots, v_1\}$. Recall that $C: V \to \mathbb{N}$ is a monotonic function, namely weakly increasing, because if $u > v$ then $C(u) \geq C(v)$.

Consider an arbitrary node u in the current order. If $c(u) = 1$ and the node u moves to the end of the queue then $C(u) = 1$ and no other value changes in the register functions.

Consider two *consecutive* nodes u and v in the order $u > v$. If $c(u) \leq c(v)$ then swapping the nodes can decrement the value of $C(u)$ and $c(u)$ without any further side effects in the register functions. The new values of $C(u)$ and $c(u)$ must be recalculated.

Iterating the transposition of such consecutive nodes leads to an order, where for every $u > v$ consecutive nodes $c(u) \geq c(v)$ stands, therefore $c: V \to \mathbb{N}$ becomes monotonic, namely weakly increasing.

Theorem 4.1. If both of the register functions are weakly increasing on a given order of nodes then they are identical.

Proof. (Mathematical induction) The statement holds for one node because $C(v_1) = c(v_1) = 1$. Let be assumed that the statement holds for the nodes $\{v_k, \dots, v_1\}$. Since the register functions are weakly increasing, $C(v_{k+1}) \geq C(v_k)$ and $c(v_{k+1}) \geq c(v_k)$. Moreover the function C is dominating the function c, therefore $C(v_{k+1}) \geq c(v_{k+1})$. It was already established that $C(v_{k+1}) = C(v_k) + 1$ means that v_{k+1} is an element of some largest clique in $\{v_{k+1}, \dots, v_1\}$, thus $c(v_{k+1}) = C(v_{k+1})$. Reversely, if $c(v_{k+1}) = c(v_k) + 1$ then, based on the dominance, $C(v_{k+1}) \geq c(v_{k+1})$, and thus the equality stands again. □

An order of nodes is called **discrepancy-free** if the register function c is monotonic on the nodes. It is possible to find many different discrepancy-free orders of the nodes of an arbitrary graph. The total summation $\sum_{v \in V} c(v)$ of the register function c is usually different too, in the case of the different discrepancy-free orders.

The simplest way of producing a discrepancy-free order of nodes is a combination of $\mathbf{CP}'$ and bubble sort. Let the nodes following u in the current order be denoted by $\underline{U}(u) = \{v \in V \mid u > v\}$. Assuming $c(v)$ is known and monotonic for every $v \in \underline{U}(u)$, $\mathbf{CP}'$ can determine $c(u)$, then bubble sort can move u backward in the list until the first node $v \in \underline{U}(u)$ such that $c(u) > c(v)$. At that point $\mathbf{CP}'$ should update $c(u)$, and if changed, bubble sort moves u on, and so forth.

Instead of swapping consecutive nodes, the final place of a node can be determined in advance by recursively calling $\mathbf{CP}'$, and the physical move can be executed when $\mathbf{CP}'$ cannot decrease the value of $c(u)$ anymore. This would result in a combination of $\mathbf{CP}'$ and insertion sort, slightly more effective than the previous one.

The function returning the node of lowest index in the subset U before which a new node u can be inserted without changing the monotonicity of c is as follows:

$$\underline{\mathbf{DOWNTO}(U, u)}:$$

$c(u) = 1 + \mathbf{CP}'(U \cap N(u), 0)$

if $c(u) > c\left(v_{\max_U}\right)$ **then return** $v_{\max_U}$

$i := \max\left\{ j \in \mathbb{N} \mid v_j \in U \wedge c(u) > c(v_j) \right\}$

return $\mathbf{DOWNTO}\left(\{v_i\} \cup \underline{U}(v_i), u\right)$

Inserting new nodes before the result of **DOWNTO** will produce a discrepancy-free order and compute c at the same time. Consequently that is another way of solving MCSP, although not an efficient one.

5. IMPROVED ALGORITHM FOR MCSP AND MCP

Experiments with Östergård's algorithm on various node orders resulted in a weak ranking of polynomial ordering methods. The four best of them, which ensure the shortest runtimes of the cliquer on random graphs, are introduced in the following paragraphs.

Let $d(v,U) = |N(v) \cap U|$ denote the degree of the node$v$ in the graph induced by the subset U. Furthermore, let V_i denote the subset $\{v_n, \dots, v_i\}$ and V^i denote the subset $\{v_i, \dots, v_1\}$ of the ordered nodes.

Definition 5.1. The nodes $\{v_n, \dots, v_1\}$ are in **Min** order if

$$\forall i \left(d(v_i, V^i) = \min_{j \le i} \{ d(v_j, V^i) \} \right)$$

Definition 5.2. The nodes $\{v_n, \dots, v_1\}$ are in **Max** order if

$$\forall i \left(d(v_i, V_i) = \max_{j \ge i} \{ d(v_j, V_i) \} \right)$$

Let $\mathcal{C}: V \to \mathbb{N}$ denote a coloring function of the graph, so that

$$\{u, v\} \in E \Longrightarrow \mathcal{C}(u) \neq \mathcal{C}(v)$$

Let $N^{\mathcal{C}}(v_i)$ denote the union of the neighborhood of nodes preceding v_i and having the same color as v_i has, thus

$$N^{\mathcal{C}}(v_i) = \bigcup_{k>i} \{ N(v_k) \mid \mathcal{C}(v_k) = \mathcal{C}(v_i) \}$$

Definition 5.3. The nodes $\{v_n, \dots, v_1\}$ are in **MinColor** order if

- $\mathcal{C}: V \to \mathbb{N}$ is weakly decreasing monotonic function on the nodes, thus $i < j \Longrightarrow \mathcal{C}(v_i) \geq \mathcal{C}(v_j)$;
- $\forall i \left(\mathcal{C}(v_j) < C(v_i) \Longrightarrow \cup \{ N(v_k) | \mathcal{C}(v_k) = \mathcal{C}(v_j) \} = V \right)$;
- and

$$\forall i \left(d\left(v_i, V \setminus N^{\mathcal{C}}(v_i)\right) = \min_{j \le i} \left\{ d\left(v_j, V \setminus N^{\mathcal{C}}(v_i)\right) \right\} \right)$$

Let $N_{\mathcal{C}}(v_i)$ denote the union of the neighborhood of nodes following v_i and having the same color as v_i has, thus

$$N_{\mathcal{C}}(v_i) = \bigcup_{k<i} \{ N(v_k) \mid \mathcal{C}(v_k) = \mathcal{C}(v_i) \}$$

Definition 5.4. The nodes $\{v_n, \dots, v_1\}$ are in **MaxColor** order if

- $\mathcal{C}: V \to \mathbb{N}$ is weakly increasing monotonic function on the nodes, thus $i < j \Longrightarrow \mathcal{C}(v_i) \leq \mathcal{C}(v_j)$;
- $\forall i\left(\mathcal{C}(v_j) < C(v_i) \Longrightarrow \cup\{N(v_k)|\mathcal{C}(v_k) = \mathcal{C}(v_j)\} = V\right)$;
- and

$$\forall i \left(d\left(v_i, V \setminus N_{\mathcal{C}}(v_i)\right) = \max_{j \geq i} \left\{d\left(v_j, V \setminus N_{\mathcal{C}}(v_i)\right)\right\}\right)$$

The well-known coloring method for **MinColor** and **MaxColor** orders is based on the recursive generation of maximal independent subsets of the nodes.

Experiments with Östergård's algorithm on discrepancy-free orders of nodes constructed from the above defined best orders clarified that its runtime on a discrepancy-free order is often shorter than upon the initial order. Random graph tests suggest using the discrepancy-free order constructed from the **Max** order of nodes.

The results of benchmarking on DIMACS graphs are presented in *Table 1*. Algorithms have been implemented using bitwise representation of the graph. Tests were run by an Intel®Core™ i5-2520M CPU @ 2.50 GHz.

Table 1. Runtime of Östergård's algorithm in [sec] on different order of nodes*

Graph	Size	Density	MCS	Min	Max	MinColor	MaxColor	D-free
brock200_1	200	74.53%	21	7.8	5.0	9.2	5.3	4.6
c-fat200-5	200	42.57%	58	210.0	124.0	0.0	12.8	0.0
c-fat500-5	500	18.59%	64	0.5	0.0	0.0	190.0	0.0
c-fat500-10	500	37.37%	126	*> 10k*	0.0	0.0	*> 10k*	0.0
C125.9	125	89.77%	34	16.8	4.3	42.5	8.6	3.8
DSJC500.5	500	50.20%	13	5.6	3.6	4.6	4.1	3.2
monotone-6	216	75.53%	14	61.5	0.3	35.0	1.1	0.0
p_hat500-2	500	50.45%	36	22.5	35.0	63.0	53.0	31.0
p_hat1500-1	1500	25.34%	12	7.9	7.4	7.7	9.3	7.6
san200_0.7_1	200	69.98%	30	*> 10k*	1.2	1880.0	0.2	*> 10k*
san200_0.7_2	200	69.98%	18	*> 10k*	29.0	*> 10k*	0.0	0.0
san400_0.5_1	400	49.99%	13	1170.0	0.8	600.0	0.0	2.7
sanr200_0.7	200	69.67%	18	1.4	1.0	1.5	0.7	0.7
sanr400_0.5	400	50.10%	13	0.8	0.8	0.8	0.8	0.7

* MCS = maximum clique size

Although Östergård's algorithm runs generally faster on a discrepancy-free order of nodes than upon other orders, producing the discrepancy-free order takes more time than Östergård's algorithm itself. Fortunately there is a way of combining Östergård's algorithm with the ordering method which results in a new solver algorithm of competitive speed for MCP and MCSP.

The principle of the combination is a sort of parallel execution of the two procedures. Allow Östergård's algorithm to go forward and compute the original register function C in the usual way. Parallel to that let the insertion sort algorithm reorder the already processed nodes for that Östergård's algorithm meet gradually developing conditions when it returns to the lower part of the node sequence in a recursive call! Of course, the sorting procedure must take care of updating the register function as well. The insertion sort algorithm can thus assist Östergård's algorithm.

Beyond benefitting from the general optimizing effect of the discrepancy-free order, the new construction has the ability to save the overall performance of Östergård's cliquer. For instance, in the case of the graphsan200_0.7_1, the new algorithm runs slightly faster than Östergård's cliquer on the **Max** order of nodes, performing 1.1 second in average.

The new construction has been compared to the bitwise implementation of Östergård's cliquer, and the benchmarking results are displayed in *Table 2*. Test graphs have been preconditioned by **Max** ordering the nodes.

Table 2. Benchmark of MCP solvers on DIMACS graphs

Graph	Size	Density	MCS	Östergård	NEW
brock400_1	400	74.83%	27	01:46:40	01:36:53
brock400_2	400	74.91%	29	00:24:13	00:19:02
brock400_3	400	74.78%	31	00:06:34	00:04:33
brock400_4	400	74.89%	33	00:01:36	00:01:04
gen200_p0.9_44	200	89.98%	44	02:43:24	00:24:45
gen200_p0.9_55	200	89.98%	55	00:02:39	00:00:07
monotone-7	343	78.91%	19	00:00:24	00:00:17
monotone-8	512	81.51%	23	02:54:33	00:34:08
p_hat300-3	300	74.43%	36	00:04:00	00:00:58
p_hat700-2	700	49.75%	44	01:09:27	00:20:20
p_hat1000-2	1000	49.01%	46	84:19:05	41:02:20
san400_0.7_3	400	69.99%	22	02:32:34	00:26:26
sanr200_0.9	200	89.76%	42	03:05:10	00:45:20
sanr400_0.7	400	70.00%	21	00:24:22	00:23:39

Figures show distinct and stable improvement of different extent, from insignificant measures to spectacular ones. Tests on random graphs suggest an improvement of 50% in average.

ACKNOWLEDGEMENTS

Hereby the author expresses many thanks to S. Szabó and B. Zaválnij for the inspirations, for the open and honest discussions and for their accurate criticism during the preparation of the above

results.

REFERENCES

[1] Pardalos, P. M. – Xue, J.,*The Maximum Clique Problem*, Journal of Global Optimization 4, 301-328, 1994.

[2] Bomze, I. M. – Budinich, M. – Pardalos, P. M. – Pelillo, M., *The Maximum Clique Problem*, Handbook of Combinatorial Optimization 4, Kluwer Academic Publishers, 1-74, 1999.

[3] Carraghan, R. – Pardalos, P. M.,*An Exact Algorithm for the Maximum Clique Problem*, Operational Research Letters 9, Elsevier S. P., 375-382, 1990.

[4] Östergård, P. R. J.,*A Fast Algorithm for the Maximum Clique Problem*, Discrete Applied Mathematics 120, 197-207, 2002.

[5] Kumlander, D.,*A Simple and Efficient Algorithm for the Maximum Clique Finding Reusing a Heuristic Vertex Colouring*, IADIS International Journal on Computer Science and Information Systems, 1(2), 32-49, 2006.