An Efficient Branch-and-Bound Algorithm for Finding a Maximum Clique*

Etsuji Tomita and Tomokazu Seki

The Graduate School of Electro-Communications
The University of Electro-Communications
Chofugaoka 1-5-1, Chofu, Tokyo 182-8585, Japan
{tomita,seki-t}@ice.uec.ac.jp

Abstract. We present an exact and efficient branch-and-bound algorithm for finding a maximum clique in an arbitrary graph. The algorithm is not specialized for any particular kind of graph. It employs approximate coloring and appropriate sorting of vertices to get an upper bound on the size of a maximum clique. We demonstrate by computational experiments on random graphs with up to 15,000 vertices and on DIMACS benchmark graphs that our algorithm remarkably outperforms other existing algorithms in general. It has been successfully applied to interesting problems in bioinformatics, image processing, the design of quantum circuits, and the design of DNA and RNA sequences for bio-molecular computation.

1 Introduction

Given an undirected graph G, a *clique* is a subgraph of G in which all the pairs of vertices are adjacent. Finding a maximum clique in a graph is one of the most important NP-hard problems in discrete mathematics and theoretical computer science and has been studied by many researchers. Pardalos and Xue [13] and Bomze *et al.* [4] give excellent surveys on this problem together with quite many references. See also *Chapter 7: Selected Applications in* [4] for applications of maximum clique algorithms.

One standard approach for finding a maximum clique is based on the branch-and-bound method. Several branch-and-bound algorithms use approximate coloring to get an upper bound on the size of a maximum clique. Elaborate coloring can greatly reduce the search space. Coloring, however, is time consuming, and it becomes important to choose the proper trade-off between the time needed for approximate coloring and the reduction of the search space thereby obtained. Many efforts have been made along this line, see Bomze *et al.* [4]. Quite recently,

* This work is partially supported by Grant-in-Aid for Scientific Research No.13680435 from MESSC of Japan and Research Fund of the University of Electro-Communications. It is also given a grant by Funai Foundation for Information Technology.

C.S. Calude et al. (Eds.): DMTCS 2003, LNCS 2731, pp. 278–289, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Östergård [12] proposed a new maximum-clique algorithm, supported by computational experiments. Sewell [15] presents a maximum-clique algorithm designed for dense graphs.

In this paper, we present a branch-and-bound algorithm, MCQ, for finding a maximum clique based on approximate coloring and appropriate sorting of the vertices. We experimentally compare MCQ with other algorithms, especially that of Östergård [12]. The experimental results show that our algorithm is very effective and fast for many random graphs and DIMACS benchmark graphs. Tested graphs include large random graphs with up to 15,000 vertices. Such results are important for practical use [2]. A preliminary version of this paper was given in [14], and the algorithm has yielded interesting results in bioinformatics [2] and other areas [8,11,10].

2 Preliminaries

[1] Throughout this paper, we are concerned with a simple undirected graph $G = (V, E)$ with a finite set V of vertices and a finite set E of unordered pairs (v, w) of distinct vertices, called edges. The set V of vertices is considered to be *ordered*, and the i-th element in V is denoted by $V[i]$. A pair of vertices v and w are said to be adjacent if $(v, w) \in E$. [2] For a vertex $v \in V$, let $\Gamma(v)$ be the set of all vertices which are adjacent to v in $G = (V, E)$, i.e.,

$$\Gamma(v) = \{w \in V | (v, w) \in E\} \ (\not\ni v).$$

We call $|\Gamma(v)|$, the number of vertices adjacent to a vertex v, the degree of v. In general, for a set S, the number of its elements is denoted by $|S|$.

The maximum degree in graph G is denoted by $\Delta(G)$. [3] For a subset $W \subseteq V$ of vertices, $G(W) = (W, E(W))$ with $E(W) = \{(v, w) \in W \times W | (v, w) \in E\}$ is called a subgraph of $G = (V, E)$ induced by W. [4] Given a subset $Q \subseteq V$ of vertices, the induced subgraph $G(Q)$ is said to be a clique if $(v, w) \in E$ for all $v, w \in Q$ with $v \neq w$. If this is the case, we may simply say that Q is a clique. In particular, a clique of the maximum size is called a *maximum clique*. The number of vertices of a maximum clique in graph $G = (V, E)$ is denoted by $\omega(G)$ or $\omega(V)$.

A subset $W \subseteq V$ of vertices is said to be independent if $(v, w) \notin E$ for all $v, w \in W$.

3 Maximum Clique Algorithm MCQ

Our maximum clique algorithm MCQ is shown in Figures 1, 2 and 3. In the following subsections we explain the basic approach and the motivations for our heuristics.

```
procedure MCQ (G = (V, E))
begin
    global Q := ∅;
    global Q_max := ∅;
    Sort vertices of V in a descending order
            with respect to their degrees;
    for i := 1 to Δ(G)
        do N[V[i]] := i od
    for i := Δ(G) + 1 to |V|
        do N[V[i]] := Δ(G) + 1 od
    EXPAND(V, N);
    output Q_max
end {of MCQ }
```

Fig.1. Algorithm MCQ

```
procedure EXPAND(R, N)
begin
    while R ≠ ∅ do
        p := the vertex in R such that N[p] = Max{N[q] | q ∈ R};
                {i.e., the last vertex in R}
        if |Q| + N[p] > |Q_max| then
            Q := Q ∪ {p};
            R_p := R ∩ Γ(p);
            if R_p ≠ ∅ then
                NUMBER-SORT(R_p, N');
                {the initial value of N' has no significance}
                EXPAND(R_p, N')
            else if |Q| > |Q_max| then Q_max := Q fi
            fi
            Q := Q − {p}
            else return
        fi
        R := R − {p}
    od
end {of EXPAND}
```

Fig.2. EXPAND

3.1 A Basic Algorithm

Our algorithm begins with a small clique, and continues finding larger and larger cliques until one is found that can be verified to have the maximum size. More precisely, we maintain global variables Q, Q_{max}, where Q consists of vertices of a current clique, Q_{max} consists of vertices of the largest clique found so far. Let $R \subseteq V$ consist of vertices (candidates) which may be added to Q. We begin the algorithm by letting $Q := \emptyset$, $Q_{max} := \emptyset$, and $R := V$ (the set of all vertices). We select a certain vertex p from R and add p to Q ($Q := Q \cup \{p\}$). Then we

compute $R_p := R \cap \Gamma(p)$ as the new set of candidate vertices. This procedure (EXPAND) is applied recursively, while $R_p \neq \emptyset$.

When $R_p = \emptyset$ is reached, Q constitutes a maximal clique. If Q is maximal and $|Q| > |Q_{max}|$ holds, Q_{max} is replaced by Q. We then backtrack by removing p from Q and R. We select a new vertex p from the resulting R and continue the same procedure until $R = \emptyset$. This is a well known basic algorithm for finding a maximum clique (see, for example, [7]).

3.2 Pruning

Now, in order to prune unnecessary searching, we make use of *approximate coloring* of vertices as introduced by Fujii and Tomita [7] and Tomita *et al.*[16]. We assign in advance for each $p \in R$ a positive integer $N[p]$ called the *Number* or *Color* of p with the following property:

(i) If $(p, r) \in E$ then $N[p] \neq N[r]$, and
(ii) $N[p] = 1$, or if $N[p] = k > 1$, then there exist vertices
$p_1 \in \Gamma(p), p_2 \in \Gamma(p), \ldots, p_{k-1} \in \Gamma(p)$ in R with
$N[p_1] = 1, N[p_2] = 2, \ldots, N[p_{k-1}] = k - 1$.

Consequently, we know that

$$\omega(R) \leq \mathrm{Max}\{N[p] | p \in R\},$$

and hence if $|Q| + \mathrm{Max}\{N[p] | p \in R\} \leq |Q_{max}|$ holds then we can disregard such R.

The value $N[p]$ for every $p \in R$ can be easily assigned step by step by a so-called *greedy coloring* algorithm as follows: Assume the vertices in $R = \{p_1, p_2, \ldots, p_m\}$ are arranged in this order. First let $N[p_1] = 1$. Subsequently, let $N[p_2] = 2$ if $p_2 \in \Gamma(p_1)$ else $N[p_1] = 1, \ldots$, and so on. After $Numbers$ are assigned to all vertices in R, we sort these vertices in ascending order with respect to their $Numbers$. We call this numbering and sorting procedure NUMBER-SORT. See Figure 3 for details of Procedure NUMBER-SORT. This procedure runs in $O(|R|^2)$ time. Note that the quality of such *sequential* coloring depends heavily on how the vertices are ordered. Therefore, the SORT portion of the NUMBER-SORT procedure is important. Then we consider a simple and effective sorting procedure as follows.

Let $\mathrm{Max}\{N[r] | r \in R\} = maxno$,

$$C_i = \{r \in R | N[r] = i\}, \quad i = 1, 2, \ldots, maxno,$$

and

$$R = C_1 \cup C_2 \cup \ldots \cup C_{maxno},$$

where vertices in R are *ordered* in such a way that the vertices in C_1 appear

```
procedure NUMBER-SORT(R, N)
begin
{NUMBER}
    maxno := 1;
    C1 := ∅;   C2 := ∅;
    while R ≠ ∅ do
        p := the first vertex in R;
        k := 1;
        while Ck ∩ Γ(p) ≠ ∅
            do  k := k + 1  od
        if k > maxno then
            maxno := k;
            Cmaxno+1 := ∅
        fi
        N[p] := k;
        Ck := Ck ∪ {p};
        R := R − {p}
    od
{SORT}
    i := 1;
    for k := 1 to maxno do
        for j := 1 to |Ck| do
            R[i] := Ck[j];
            i := i + 1
        od
    od
end {of NUMBER-SORT}
```

Fig.3. NUMBER-SORT

first in the same order as in C_1, and then vertices in C_2 follow in the same way, and so on.

Then let

$$C'_i = C_i \cap \Gamma(p),\ i = 1, 2, \ldots, maxno,$$

for some $p \in R$, and we have that

$$R_p = R \cap \Gamma(p) = C'_1 \cup C'_2 \cup \ldots \cup C'_{maxno},$$

where vertices in R_p are ordered as described above. Both of C_i and C'_i are independent sets, and $C'_i \subseteq C_i$, for $i = 1, 2, \ldots, maxno$. Then it is clear that the maximum $Number$ ($Color$) needed for coloring C'_i is less than or equal to that needed for C_i. This means that the above coloring is steadily improved step by step owing to the procedure NUMBER-SORT. In addition, it should be strongly noted that the latter part {SORT} in Fig.3 takes *only* $O(|R|)$ time.

A more elaborate coloring can be more effective in reducing the total search space, but our preliminary computational experiments indicate that elaborate

coloring schemes take so much more time to compute that they have an overall negative effect on performance.

In procedure EXPAND(R, N), after applying NUMBER-SORT more than once, a maximum clique contains a vertex p in R such that $N[p] \geq \omega(R)$. It is generally expected that a vertex p in R such that $N[p] = \mathrm{Max}\{N[q]|q \in R\}$ has high probability of belonging to a maximum clique. Accordingly, we select a vertex p in R such that $N[p] = \mathrm{Max}\{N[q]|q \in R\}$ as described at the beginning of **while** loop in EXPAND(R, N). Here, a vertex p such that $N[p] = \mathrm{Max}\{N[q]|q \in R\}$ is the *last* element in the ordered set R of vertices after the application of NUMBER-SORT. Therefore, we simply select the rightmost vertex p in R while $R \neq \emptyset$. Consequently, vertices in R are searched from the last (right) to the first (left).

3.3 Initial Sorting and Simple Numbering

Fujii and Tomita [7] have shown that both search space and overall running time are reduced when one sorts the vertices in an ascending order with respect to their degrees prior to the application of a branch-and-bound algorithm for finding a maximum clique. Carraghan and Pardalos [6] also employ a similar technique successfully. Therefore, at the beginning of our algorithm, we sort vertices in V in a descending order with respect to their degrees. This means that a vertex with the minimum degree is selected at the beginning of the **while** loop in EXPAND(V, N) since the selection there is from right to left.

Furthermore, we initially assign $Numbers$ to the vertices in R simply so that $N[V[i]] = i$ for $i \leq \Delta(G)$, and $N[V[i]] = \Delta(G) + 1$ for $\Delta(G) + 1 \leq i \leq |V|$. This initial $Number$ has the desired property that in EXPAND(V, N), $N[p] \geq \omega(V)$ for any p in V **while** $V \neq \emptyset$. Thus, this simple initial $Number$ suffices.

This completes our explanation of the MCQ algorithm shown in Figures 1, 2 and 3. Note that the rather time-consuming calculation of the degree of vertices is carried out only at the beginning of MCQ and nowhere in NUMBER-SORT. Therefore, the total time needed to reduce the search space can be *very small*. It should be also noted that the initial order of vertices in our algorithm is effective for the reduction of the search space as described at the beginning of this subsection. And it is also true in the following subproblems. This is because the initial order of vertices in the same $Number$ is *inherited* in the following subproblems owing to the way of NUMBER-SORT.

4 Computational Experiments

We have implemented the algorithm MCQ in the language C and carried out computational experiments to evaluate it. The computer used has a Pentium4 2.20GHz CPU and a Linux operating system. See Appendix for details. Here, benchmark program dfmax given by Applegate and Johnson (see Johnson and Trick [9]) is used to obtain user time[sec] to solve the given five benchmark instances.

Table 1. CPU time [sec] for random graphs

Graph			dfmax	MCQ	New	COCR
n	p	ω	[9]		[12]	[15]
100	0.5	9-10	0.0019	**0.0009**		0.13
	0.6	11-13	0.0061	**0.0026**	0.0035	0.14
	0.7	14-16	0.0286	**0.0070**	0.011	0.18
	0.8	19-21	0.22	**0.026**	0.10	0.24
	0.9	29-32	5.97	**0.066**	1.04	0.31
	0.95	39-46	40.94	**0.0023**	0.31	
150	0.7	16-18	0.57	**0.12**		0.53
	0.8	23	11.23	**0.93**		1.18
	0.9	36-39	1,743.7	9.57		**1.83**
	0.95	50-57	61,118.8	**4.03**		
200	0.4	9	0.012	**0.0061**	0.011	
	0.5	11-12	0.058	**0.025**	0.03	0.40
	0.6	14	0.46	**0.14**	0.27	0.82
	0.7	18-19	6.18	**1.13**	4.75	2.59
	0.8	24-27	314.92	20.19	231.54	**13.66**
300	0.4	9-10	0.078	**0.038**	0.074	
	0.5	12-13	0.59	**0.24**	0.32	1.78
	0.6	15-16	7.83	**2.28**	5.50	7.83
	0.7	19-21	233.69	**37.12**	179.71	
500	0.2	7	0.018	**0.012**	0.03	
	0.3	8-9	0.13	**0.067**	0.13	
	0.4	11	1.02	**0.48**	0.94	
	0.5	13-14	14.45	**5.40**	11.40	27.41
	0.6	17	399.22	**96.34**	288.10	
1,000	0.2	7-8	0.24	**0.18**	0.33	
	0.3	9-10	3.09	**1.85**	2.58	
	0.4	12	51.92	**22.98**	36.46	
	0.5	15	1,766.85	**576.36**		
5,000	0.1	7	13.51	**11.60**		
	0.2	9	531.65	**426.38**		
	0.3	12	28,315.34	**18,726.21**		
10,000	0.1	7	256.43	**185.99**		
	0.2	10	23,044.55	**16,967.60**		
15,000	0.1	8	1,354.64	**876.65**		

4.1 Results for Random Graphs

Prior to the present work, it was confirmed that an earlier version of MCQ was faster than Balas and Yu [1]'s algorithm by computational experiments for random graphs [16].

Now for each pair of n (the number of vertices) up to 15,000 and p (edge probability) in Table 1, random graphs are generated so that there exists an edge for each pair of vertices with probability p. Then average CPU time [seconds] required to solve these graphs by dfmax and MCQ are listed in Table 1. The averages are taken for 10 random graphs for each pair of n and p. The exceptions are for $n \geq 5,000$, where each CPU time is for one graph with the given n and p. In addition, CPU times by New in Östergård [12] and COCR in Sewell [15] are added to Table 1, where each CPU time is adjusted according to the ratio

Table 2. Branches for random graphs

Graph			dfmax	MCQ	COCR
n	p	ω	[9]		[15]
100	0.5	9-10	3,116	418	256
	0.6	11-13	11,708	942	390
	0.7	14-16	61,547	2,413	763
	0.8	19-21	486,742	6,987	894
	0.9	29-32	13,618,587	10,854	343
	0.95	39-46	86,439,090	2,618	
150	0.8	23	21,234,191	173,919	19,754
	0.9	36-39	2,453,082,795	1,046,341	17,735
	0.95	50-57	$> 4.29 \times 10^9$	273,581	
200	0.5	11-12	85,899	7,900	4,500
	0.6	14	728,151	38,378	16,747
	0.7	18-19	10,186,589	233,495	62,708
	0.8	24-27	542,315,390	3,121,043	260,473
300	0.5	12-13	758,484	56,912	36,622
	0.6	15-16	10,978,531	473,629	197,937
500	0.3	8-9	131,196	18,829	
	0.4	11	1,148,304	124,059	
	0.5	13-14	16,946,013	1,124,109	582,631
	0.6	17	469,171,354	16,062,634	
1,000	0.2	7-8	211,337	43,380	
	0.3	9-10	2,989,296	463,536	
	0.4	12	50,154,790	4,413,740	
	0.5	15	1,712,895,181	89,634,336	
5,000	0.1	7	1,763,294	539,549	
	0.2	9	158,904,545	31,785,500	
	0.3	12	3,013,993,714	1,303,729,211	
10,000	0.1	7	35,158,358	5,718.030	
	0.2	10	481,284,721	588,220,975	
15,000	0.1	8	158,776,693	22,196,166	

given in Appendix. Bold faced entries are the ones that are fastest in the same row.

For random graphs in this Table, MCQ is fastest except for the cases where $[n = 150, p = 0.9]$ and $[n = 200, p = 0.8]$. In these cases, COCR is fastest. COCR is specially designed for solving the maximum clique problem in *dense* graphs. Now we let *Branches* mean the total number of EXPAND() calls excluding the one at the beginning. Hence, *Branches* correspond to an extent of the search space. A part of the associated Branches by dfmax and MCQ is listed in Table 2 together with the corresponding values by COCR which is cited from [15].

Together, Tables 1 and 2 show that MCQ is successful in general for obtaining a good trade-off between the increase in time and the reduction of the search space associated with approximate coloring.

Up to the present, it is widely recognized that dfmax is the fastest maximum-clique algorithm for sparse graphs [12]. Table 1, however, shows that MCQ is faster than dfmax for all graphs tested, including very sparse graphs. It is to be noted that MCQ is much faster than dfmax when the number of vertices is very large, even for sparse graphs.

Table 3. CPU time [sec] for DIMACS benchmark graphs

In each row, the *fastest* entry is **bold faced**, ***italicized***, ***underlined***, ⋆ **marked**,⋆ ⋆ **marked**, and ⋆⋆⋆ **marked** if it is more than 2, 10, 20, 100, 400, and at least 2,000 times faster than or equal to the second fastest one, respectively. The other *fastest* entries are ∘ **marked**.

Graph				dfmax	MCQ	New	MIPO	SQUEEZE
Name	n	Density	ω	[9]		[12]	[3]	[5]
brock200_1	200	0.745	21	23.96	**2.84**	19.05		1,471.0
brock200_2	200	0.496	12	0.05	∘ 0.016	0.018	847.3	60.4
brock200_3	200	0.605	15	0.35	∘ 0.09	0.15		194.5
brock200_4	200	0.658	17	1.47	∘ 0.32	0.34		419.1
c-fat200-1	200	0.077	12	∘ 0.00016	0.00024	0.0035	22.1	3.21
c-fat200-2	200	0.163	24	∘ 0.0003	0.0005	0.0035	11.2	2.78
c-fat200-5	200	0.426	58	444.03	⋆⋆ ***0.0021***	2.74	60.5	1.74
c-fat500-1	500	0.036	14	∘ 0.0010	0.0011	0.025		51.2
c-fat500-2	500	0.073	26	∘ 0.0011	0.0018	0.028		90.7
c-fat500-5	500	0.186	64	2.73	⋆⋆ ***0.0061***	3,664.16		49.5
c-fat500-10	500	0.374	126	>24hrs.	0.0355	∘ 0.025		36.3
hamming6-2	64	0.905	32	0.0175	***0.0003***	0.0035	0.004	
hamming6-4	64	0.349	4	0.00015	∘ 0.00010	0.0035	0.48	0.09
hamming8-2	256	0.969	128	>24hrs.	0.0205	∘ 0.014	0.05	
hamming8-4	256	0.639	16	3.07	0.34	∘ 0.30		1,289.5
hamming10-2	1,024	0.990	512	>24hrs.	1.82	∘ 0.88		0.96
johnson8-2-4	28	0.556	4	0.000050	**0.000022**	0.0035	0.0003	
johnson8-4-4	70	0.768	14	0.0071	**0.0009**	0.0035	0.004	0.35
johnson16-2-4	120	0.765	8	1.21	0.34	0.095	***0.003***	777.2
MANN_a9	45	0.927	16	0.062	***0.0002***	0.0035		
MANN_a27	378	0.990	126	>24hrs.	⋆ ***8.49***	>3,500		1,524.2
keller4	171	0.649	11	0.62	**0.0450**	0.175	184.6	236.8
p_hat300-1	300	0.244	8	0.008	∘ 0.0053	0.014	1,482.8	169.1
p_hat300-2	300	0.489	25	1.04	**0.08**	0.35		360.9
p_hat300-3	300	0.744	36	1,285.91	***26.49***			8,201.1
p_hat500-1	500	0.253	9	0.09	**0.04**	0.102		1,728.9
p_hat500-2	500	0.505	36	219.37	***6.22***	150.5		5,330.7
p_hat700-1	700	0.249	11	0.32	∘ 0.16	0.24		>2hrs.
p_hat1000-1	1,000	0.245	10	1.66	∘ 0.86	2.05		
p_hat1500-1	1,500	0.253	12	15.57	**7.40**			
san200_0.7_1	200	0.700	30	4,181.46	***0.0169***	0.20	0.33	310.9
san200_0.7_2	200	0.700	18	26,878.56	∘ 0.0106	0.014	8.47	>2hrs.
san200_0.9_1	200	0.900	70	>24hrs.	2.30	0.095	∘ 0.08	0.43
san200_0.9_2	200	0.900	60	>24hrs.	3.49	1.50	**0.24**	12.6
san200_0.9_3	200	0.900	44	>24hrs.	∘ 16.41		23.60	430.7
san400_0.5_1	400	0.500	13	719.16	0.041	**0.011**	133.15	
san400_0.7_1	400	0.700	40	>24hrs.	⋆⋆⋆ ***1.72***	>3,500		
san400_0.7_2	400	0.700	30	>24hrs.	⋆ ***1.58***	177.6	786.8	
san400_0.9_1	400	0.900	100	>24hrs.	***72.69***			3,239.8
san1000	1,000	0.502	15	>24hrs.	10.06	***0.18***		
sanr200_0.7	200	0.702	18	5.02	**0.92**	4.95		318.0
sanr400_0.5	400	0.501	13	3.50	∘ 1.47	2.32		

4.2 Results for DIMACS Benchmark Graphs

Table 3 lists the CPU times required by dfmax and MCQ to solve DIMACS benchmark graphs given in Johnson and Trick [9]. In addition, CPU times by New [12], MIPO in Balas *et al.* [3], and SQUEEZE in Bourjolly *et al.* [5] are added to Table 3, where each CPU time is adjusted according to the ratio given in Appendix.

The results in Table 3 show that MCQ is faster than dfmax except for only very "small" graphs which MCQ needs less than 0.0025 seconds to solve.

MCQ is also faster than New [12] except for several graphs. For a more detailed comparison of MCQ vs. New [12], we note that MCQ is more than 10 times faster than New [12] for 14 graphs, while New [12] is more than 10 times faster than MCQ for only 2 graphs in Table 3. In addition, MCQ is more than 100 times faster than New [12] for 5 graphs, while New [12] is more than 100 times faster than MCQ for no graph in Table 3. In particular, *MCQ is more than 1,000 times faster than New [12] for 4 of them.* As for COCR [15], the comparison in Table 3 of Östergård [12] shows that New [12] is faster than COCR [15] except for MANN_a27. The adjusted CPU time of COCR for MANN_a27 is 4.33 seconds which is shorter than any entry in our Table 2. Note here that the edge density of MANN_a27 is 0.99 and is very high. (For Wood [17], see Table 2 of Östergård [12] for reference.)

MCQ is faster than MIPO [3] except for instances of johnson16-2-4, san200_0.9_1, and san200_0.9_2 in Table 3, where the reason for these exceptions are not clear. MCQ is faster than SQUEEZE [5] for all instances in Table 3.

Note here that *MCQ is more than 100 times faster than all the other algorithms to solve 5 instances* in Table 3.

Summarizing the results in Sections 4.1 and 4.2, we can regard that MCQ remarkably outperforms other algorithms cited here in general.

5 Conclusions

We have shown that our pruning technique by NUMBER-SORT based upon greedy coloring is very effective and hence algorithm MCQ outperforms other algorithms in general. We have also shown experimental results for large random graphs with up to 15,000 vertices which becomes important for practical use. If we use more elaborate coloring, we can increase the performance for dense graphs but with possible deterioration for sparse graphs as in Sewell [15]. High performance of MCQ comes from its *simplicity*, especially from the simplicity of NUMBER-SORT together with the appropriate initial sorting and simple *Numbering* of vertices.

Our algorithm MCQ has already been successfully applied to solve some interesting problems in bioinformatics by Bahadur *et al.* [2], image processing by Hotta *et al.* [8], the design of quantum circuits by Nakui *et al.* [11], the design of DNA and RNA sequences for bio-molecular computation by Kobayashi *et al.* [10].

Acknowledgement

The authors would like to express their gratitude to T. Fujii, Y. Kohata, and H. Takahashi for their contributions in an early stage of this work. Useful discussions with T.Akutsu and J.Tarui are also acknowledged. Many helpful detailed comments by E. Harley are especially appreciated.

References

1. E. Balas and C.S. Yu: "Finding a maximum clique in an arbitrary graph," SIAM J. Comput. 15, pp.1054-1068 (1986).
2. D. Bahadur K.C., T. Akutsu, E. Tomita, T. Seki, and A. Fujiyama: "Point matching under non-uniform distortions and protein side chain packing based on efficient maximum clique algorithms," Genome Informatics 13, pp.143-152 (2002).
3. E. Balas, S. Ceria, G. Cornuéjols, and G. Pataki: "Polyhedral methods for the maximum clique problem," pp.11-28 in [9] (1996).
4. I.M. Bomze, M. Budinich, P.M. Pardalos, and M. Pelillo: "The Maximum Clique Problem." In: D.-Z. Du and P.M. Pardalos (Eds.), Handbook of Combinatorial Optimization, Supplement vol. A, Kluwer Academic Publishers, pp.1-74 (1999).
5. J.-M. Bourjolly, P. Gill, G. Laporte, and H. Mercure: "An exact quadratic 0-1 algorithm for the stable set problem," pp.53-73 in [9] (1996).
6. R. Carraghan and P.M. Pardalos: "An exact algorithm for the maximum clique problem," Oper. Res. Lett. 9, pp.375-382 (1990).
7. T. Fujii and E. Tomita: "On efficient algorithms for finding a maximum clique ," Technical Report of IECE (in Japanese), AL81-113, pp.25-34 (1982).
8. K. Hotta, E. Tomita, T. Seki, and H. Takahashi: "Object detection method based on maximum cliques," Technical Report of IPSJ (in Japanese), 2002-MPS-42, pp.49-56 (2002).
9. D. S. Johnson and M. A. Trick, (Eds.): "Cliques, Coloring, and Satisfiability," DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol.26, American Mathematical Society (1996).
10. S. Kobayashi, T. Kondo, K. Okuda, and E. Tomita: "Extracting globally structure free sequences by local structure freeness," Technical Report CS 03-01, Dept. of Computer Science, Univ. of Electro-Communications (2003).
11. Y. Nakui, T. Nishino, E. Tomita, and T. Nakamura:" On the minimization of the quantum circuit depth based on a maximum clique with maximum vertex weight," Technical Report of Winter LA Symposium 2002, pp.9.1-9.7 (2003).
12. P.R.J. Östergård: "A fast algorithm for the maximum clique problem," Discrete Appl. Math. 120, pp.197-207 (2002).
13. P.M. Pardalos and J. Xue: "The maximum clique problem,"J. Global Optimization 4, pp. 301-328 (1994).
14. T. Seki and E. Tomita: "Efficient branch-and-bound algorithms for finding a maximum clique," Technical Report of IEICE (in Japanese), COMP 2001-50, pp.101-108 (2001).
15. E.C. Sewell: "A branch and bound algorithm for the stability number of a sparse graph," INFORMS J. Comput. 10, pp.438-447 (1998).
16. E. Tomita, Y. Kohata, and H. Takahashi: "A simple algorithm for finding a maximum clique," Techical Report UEC-TR-C5, Dept. of Communications and Systems Engineering, Univ. of Electro communications (1988).
17. D. R. Wood: "An algorithm for finding a maximum clique in a graph," Oper. Res. Lett. 21, pp.211-217 (1997).

Appendix – Clique Benchmark Results

Type of Machine: Pentium4 2.20GHz
Compiler and flags used: gcc -O2
MACHINE BENCHMARKS

***Our** user time for instances:*

Graph:	`r100.5`	`r200.5`	`r300.5`	`r400.5`	`r500.5`
T_1 :	2.13×10^{-3}	6.35×10^{-2}	0.562	3.48	13.3
Östergård[12]*'s user time for instances:*					
T_2 :	0.01	0.23	1.52	10.08	39.41
Ratio T_2/T_1 :	4.69	3.62	2.70	2.89	2.56
Sewell[15]*'s user time for instances:*					
T_3 :	0.14	3.64	31.10	191.98	734.99
Ratio T_3/T_1 :	65.73	57.32	55.34	55.06	55.11

For Östergård [12]'s user time for in instances[T_2] and Sewell [15]'s user time for instances[T_3], excluding the values of T_2/T_1 and T_3/T_1 for `r100.5` and `r200.5` since these instances are too small, the average value of $T_2/T_1 = 2.85$ and that of $T_3/T_1 = 55.2$. For Balas *et al.* [3]'s user time for instances (T_4) and Bourjolly *et al.* [5]'s user time for instances (T_5), the average value of $T_4/T_1 = 33.0$ and that of $T_5/T_1 = 11.5$, excluding the values T_4/T_1 and T_5/T_1 for `r100.5` for the same reason as above.