Fast maximum weight clique extraction algorithm: Optimal tables for branch-and-bound

Shimizu, Satoshi
Yamaguchi, Kazuaki
Saitoh, Toshiki
Masuda, Sumio

(Citation)
Discrete Applied Mathematics,223:120-134

(Issue Date)
2017-05-31

(Resource Type)
journal article

(Version)
Accepted Manuscript

(URL)
https://hdl.handle.net/20.500.14094/90004069

Fast maximum weight clique extraction algorithm: optimal tables for branch-and-bound

Satoshi Shimizu, Kazuaki Yamaguchi, Toshiki Saitoh, Sumio Masuda

Graduate School of Engineering, Kobe University

Abstract

A new branch-and-bound algorithm for the maximum weight clique problem is proposed. The proposed algorithm consists of two phases, a *precomputation phase* and a *branch-and-bound phase*. In the precomputation phase, the weights of maximum weight cliques in many small subgraphs are calculated and stored in *optimal tables*. In the branch-and-bound phase, each problem is divided into smaller subproblems, and unnecessary subproblems are pruned using the optimal tables. We performed experiments with the proposed algorithm and five existing algorithms for several types of graphs. The results indicate that only the proposed algorithm can obtain exact solutions for all graphs and that it performs much faster than other algorithms for nearly all graphs.

Keywords: maximum weight clique, exact algorithm, branch-and-bound, upper bound calculation, NP-hard

1. Introduction

A set of vertices V' in a graph $G = (V, E)$ is called a clique if any pair of vertices in V' are adjacent. The maximum clique problem (MCP) is to find the clique of maximum cardinality of a given graph. Here, let $w(v)$ denote the weight

Email addresses: ss81054@gmail.com (Satoshi Shimizu), ky@kobe-u.ac.jp (Kazuaki Yamaguchi), saitoh@eedept.kobe-u.ac.jp (Toshiki Saitoh), masuda@kobe-u.ac.jp (Sumio Masuda)

of $v \in V$. For a set of vertices $V' \subseteq V$, let $w(V') = \sum_{v \in V'} w(v)$. Given an undirected graph $G = (V, E)$ and weight of vertices $w(\cdot)$, the maximum weight clique problem (MWCP) is to find a clique C such that $w(C)$ is the maximum. Note that the MWCP is a generalization of the MCP.

The MCP and the MWCP are known to be NP-hard [1], and have many applications in coding theory [2], network design [3], computer vision [4], bioinformatics [5], economics [6], etc. The maximum independent set problem and the minimum vertex cover problem for general graphs are equivalent to the MCP and have been well studied.

The branch-and-bound technique is often used in exact algorithms. The branch procedure divides a problem into smaller subproblems and solves them in a recursive manner. During this process, the upper bound of each subproblem is calculated and pruned if it is proved that the subproblem does not contain the global optimum solution (the bounding procedure). In previous studies, several techniques have been investigated to obtain upper bounds for subproblems. For the MCP, vertex coloring is used in numerous algorithms [7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. They calculate vertex coloring in $O(|V|^2)$ or $O(|V|^3)$ time for each subproblem. For the MWCP, some algorithms calculate vertex coloring only once before starting branch-and-bound and use it to obtain upper bound in $O(|V|)$ for each subproblem [17, 18, 19, 20, 21]. Upper bound calculation of $O(1)$ time has also been proposed in [22, 23]. In these methods, $|V|$ subproblems are solved sequentially. During the execution, an upper bound of subpblem P is calculated from an exact value of subproblems which are already solved. Some algorithms uses some upper bounds shown above [17, 18, 21]. Other approaches have been proposed by previous studies [24, 25, 26, 27, 28, 29, 30, 31, 32]. The computation time of algorithms including branch-and-bound procedures strongly depends on tightness and computation time of upper bound calculation.

Controlling their balance is very important for branch-and-bound algorithms.

In this paper, we propose a new exact branch-and-bound algorithm for MWCP. Our algorithm consists of two phases, a *precomputation phase* and a *branch-and-bound phase*. In the precomputation, the weights of maximum weight cliques in many small subgraphs are calculated and stored in *optimal tables*. In the branch-and-bound phase, each problem is divided into smaller subproblems and solved in a recursive manner. The branch-and-bound phase is nearly the same as other branch-and-bound algorithms, i.e., the upper bound of each subproblem is calculated using the optimal tables, and the subproblem is pruned if it is unnecessary.

The remainder of this paper is organized as follows. An outline of the proposed algorithm, OTClique, is described in Section 2. Experimental results are shown in Section 4. We conclude the paper in Section 5.

2. Proposed algorithm *OTClique*

The proposed *OTClique* algorithm is outlined as follows.

- Precomputation Phase: determines branching order and generates the optimal tables
- Branch-and-bound Phase: solves the problem via a branch-and-bound procedure by pruning unnecessary subproblems by their upper bounds

Before explaining the proposed algorithm, we define some notations and analyze some properties of our upper bound function $UB(\cdot, \cdot)$. We then describe the phases of the proposed algorithm in detail.

2.1. Notation

For an undirected graph $G = (V, E)$ and a set of vertices $V' \subseteq V$, let $G(V')$ and $w_{opt}(V')$ denote the subgraph of G induced by V' and the weight of the

maximum weight clique in $G(V')$, respectively. For any vertex $v \in V$, $N(v)$ denotes the set of vertices adjacent to v in G. For any integer $k \geq 2$, a k-tuple $\Pi = (P_1, P_2, \ldots, P_k)$ is a partition of V if $P_1, P_2, \ldots, P_k$ are mutually disjoint and $\bigcup_{i=1}^{k} P_i = V$.

2.2. Upper bound function $UB(\cdot, \cdot)$

Here, we present an analysis of the following function for a subset of vertices $V' \subseteq V$ and a partition $\Pi = (P_1, P_2, \ldots, P_k)$ of V :

$$UB(\Pi, V') = \sum_{i=1}^{k} w_{opt}(V' \cap P_i) \ . \tag{1}$$

The following lemma shows that $UB(\Pi, V')$ is an upper bound of the weight of the maximum weight clique in $G(V')$.

Lemma 1. *Let $G = (V, E)$ be a vertex-weighted graph and $\Pi = (P_1, P_2, \ldots, P_k)$ be a partition of V. Then, the following inequality holds for any $V' \subseteq V$:*

$$w_{opt}(V') \leq UB(\Pi, V') \ . \tag{2}$$

Proof. The following inequality is immediately obtained, where C is the maximum weight clique in $G(V')$:

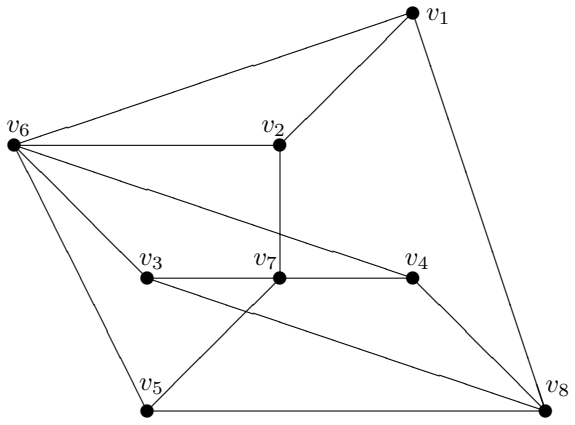
$$\begin{aligned} w_{opt}(V') &= w(C) && (3)\\ &= \sum_{i=1}^{k} w(C \cap P_i) && (4)\\ &\leq \sum_{i=1}^{k} w_{opt}(V' \cap P_i) && (5)\\ &= UB(\Pi, V') \ . && (6) \end{aligned}$$

□

2.2.1. Example

Let $G = (V, E)$ be a graph shown in Figure 1 and $\Pi = (P_1, P_2, P_3)$ be a partition of V, where P_1, P_2 and P_3 are $\{v_1, v_2\}, \{v_3, v_4, v_5\}$ and $\{v_6, v_7, v_8\}$, respectively. The weights of the vertices are shown in Figure 1. For example, the value of $UB(\Pi, V')$ for $V' = \{v_1, v_2, v_3, v_4, v_6, v_8\}$ is calculated in the following manner :

$$\begin{aligned} UB(\Pi, V') &= w_{opt}(V' \cap P_1) + w_{opt}(V' \cap P_2) + w_{opt}(V' \cap P_3) && (7) \\ &= w_{opt}(\{v_1, v_2\}) + w_{opt}(\{v_3, v_4\}) + w_{opt}(\{v_6, v_8\}) && (8) \\ &= 2 + 3 + 5 = 10 \ . && (9) \end{aligned}$$



vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
weight	1	1	2	3	3	2	4	5
partition	P_1		P_2			P_3		

Figure 1: Weighted graph

2.2.2. Optimal tables

The calculation of $UB(\Pi, V')$ takes long time if $w_{opt}(V' \cap P_i)$ is calculated in each bounding procedure. To avoid this, all the values of subproblems of each P_i are stored in the optimal tables before starting branch-and-bound processes. Optimal tables of the graph in Figure 1 are shown in Figure 2. Vertex sets are represented by bit vectors. Any $S \subseteq P_i$ is represented by a bit vector whose length is $|P_i|$. By this representation, the value $w_{opt}(S)$ for any $S \subseteq P_i$ can be obtained from the corresponding optimal table in $O(1)$ time. Therefore, for any $V' \subseteq V$, the value of $UB(\Pi, V')$ can be calculated in $O(k)$ time, where k is the number of sets in Π. For example, the upper bound calculation shown in 2.2.1 can be done as following :

$$UB(\Pi, V') = table[1][11] + table[2][011] + table[3][101] \quad (10)$$

$$= 2 + 3 + 5 = 10 \ . \quad (11)$$

2.2.3. Tightness of upper bound

The tightness of the upper bound $UB(\cdot, \cdot)$ strongly depends on Π. If each P_i in Π is an independent set, the upper bound by $UB(\Pi, V')$ will be equivalent to the upper bound used in previous studies[20], [21]. Here we show an idea to obtain tighter upper bounds in the following.

Lemma 2. *Let $G = (V, E)$ be a vertex-weighted graph and $\Pi = (P_1, P_2, \ldots, P_k)$ be a partition of V. The following inequality holds for any $V' \subset V$:*

$$UB(\Pi, V') \leq k \cdot w_{opt}(V') \ . \quad (12)$$

P_1

$S \subseteq P_1$	Bits	$w_{opt}(S)$
$\emptyset$	00	0
$\{v_1\}$	01	1
$\{v_2\}$	10	1
$\{v_1, v_2\}$	11	2

P_2

$S \subseteq P_2$	Bits	$w_{opt}(S)$
$\emptyset$	000	0
$\{v_3\}$	001	2
$\{v_4\}$	010	3
$\{v_3, v_4\}$	011	3
$\{v_5\}$	100	3
$\{v_3, v_5\}$	101	3
$\{v_4, v_5\}$	110	3
$\{v_3, v_4, v_5\}$	111	3

P_3

$S \subseteq P_3$	Bits	$w_{opt}(S)$
$\emptyset$	000	0
$\{v_6\}$	001	2
$\{v_7\}$	010	4
$\{v_6, v_7\}$	011	4
$\{v_8\}$	100	5
$\{v_6, v_8\}$	101	5
$\{v_7, v_8\}$	110	5
$\{v_6, v_7, v_8\}$	111	5

Figure 2: Optimal tables

PROOF. The inequality (12) is immediately obtained in the following way :

$$
\begin{aligned}
UB(\Pi, V') &= \sum_{i=1}^{k} w_{opt}(V' \cap P_i) && (13)\\
&\leq \sum_{i=1}^{k} w_{opt}(V') && (14)\\
&= k \cdot w_{opt}(V') \ . && (15)
\end{aligned}
$$

□

Lemma 2 shows that the tightness of $UB(\cdot, \cdot)$ depends on k, i.e., the number of subsets contained in Π. Therefore, to obtain tight upper bounds, k should be as small as possible. Algorithm 3 (shown later) makes k smaller by merging small subsets in Π to obtain tighter upper bounds.

Let us define the following notation :

$$\Pi(i) = (P_1, \ldots, P_{i-1}, P_i \cup P_{i+1}, P_{i+2}, \ldots, P_k) \quad (16)$$

$$\triangle(V', \Pi, i) = UB(\Pi, V') - UB(\Pi(i), V') \ . \quad (17)$$

The function $\triangle(V', \Pi, i)$ denotes the difference in the upper bounds between the partitions Π and $\Pi(i)$. In the following, we describe an important property of this function.

Lemma 3. *For any vertex-weighted graph $G = (V, E)$, any partition $\Pi = (P_1, P_2, \ldots, P_k)$ of V and any subset V' of V, $\triangle(V', \Pi, i)$ satisfies the following inequality :*

$$\triangle(V', \Pi, i) \leq \min\{w_{opt}(V' \cap P_i), w_{opt}(V' \cap P_{i+1})\} \ . \quad (18)$$

PROOF. From the definition of $\triangle$, the following inequality is easily obtained :

$$\begin{aligned} \triangle(V', \Pi, i) &= UB(\Pi, V') - UB(\Pi(i), V') \\ &= w_{opt}(V' \cap P_i) + w_{opt}(V' \cap P_{i+1}) - w_{opt}(V' \cap (P_i \cup P_{i+1})) \\ &\leq w_{opt}(V' \cap P_i) + w_{opt}(V' \cap P_{i+1}) \\ &\quad - \max\{w_{opt}(V' \cap P_i), w_{opt}(V' \cap P_{i+1})\} \\ &= \min\{w_{opt}(V' \cap P_i), w_{opt}(V' \cap P_{i+1})\} \ . \end{aligned} \quad (19)$$

□

2.2.4. Size of optimal tables

As subsets are merged, the value of $UB(\cdot, \cdot)$ gets tighter, and simultaneously, optimal tables get larger. In the following, we analyze the size of the area used by optimal tables. For each P_i, the values $w_{opt}(V')$ for all subsets $V' \subseteq P_i$ are stored in the optimal table for P_i. Therefore, the number of stored values is $2^{|P_i|}$for P_i and $\sum_{P_i \in \Pi} 2^{|P_i|}$ for all the optimal tables. By merging P_i and P_{i+1},

the difference of the total number of the stored values is following:

$$\begin{aligned} & 2^{|P_i|+|P_{i+1}|} - (2^{|P_i|} + 2^{|P_{i+1}|}) \\ =& 2^{|P_i|+|P_{i+1}|}(1 - (2^{-|P_{i+1}|} + 2^{-|P_i|})) \\ \geq& 2^{|P_i|+|P_{i+1}|}(1 - (2^{-1} + 2^{-1})) \\ =& 0 \ . \end{aligned} \tag{20}$$

If there is a large subset in Π, the algorithm cannot run due to a lack of memory. To avoid this problem, the upper bound l for the size of P_i should be given as an input parameter according to the amount of available memory and the number of vertices in V. Here, we show an example for calculating upper bound of l. Suppose each element of the optimal tables requires 4 bytes. If the available memory in the computer is 10^9 bytes, l must satisfy the following inequality :

$$4 \cdot \left\lceil \frac{|V|}{l} \right\rceil \cdot 2^l \leq 10^9 \ . \tag{21}$$

For example, $l \leq 22$ in case $|V| = 1000$.

2.3. Precomputation phase

The precomputation phase consists of several procedures. First, the algorithm divides vertices into independent sets and assigns numbers to these vertices (Algorithm 2). Vertices numbering determines which vertex will be chosen as a branch variable in the branch-and-bound phase. Next, a partition of V is constructed by merging some independent sets (Algorithm 3), where the parameter l is given as an input that satisfies (21). Finally, the algorithm generates the optimal tables (Algorithm 4). The entire precomputation phase is shown in Algorithm 1.

Algorithm 2 attempts to generate independent sets as large as possible; how-

Algorithm 1 Precomputation phase

INPUT: An undirected graph $G = (V, E)$, vertex weight $w(\cdot)$ and size paramter l

OUTPUT: A sequence of vertices $[v_n, v_{n-1}, \ldots, v_1]$, a partition of V : $\Pi = (P_1, P_2, \ldots, P_k)$ and optimal tables for each P_i

```
1: GENERATING_INDEPENDENT_SETS(G,w)
2: GENERATING_PARTITION(I_1,I_2,...,I_j)
3: for i from 1 to k do
4:     GENERATING_OPTIMAL_TABLE(P_i)
5: end for
```

ever, note that the cardinality of each independent set is limited to l. When the current independent set becomes maximal or the cardinality becomes l, a new independent set is created. Vertices are chosen in a weight-descending order, so that vertices of large weights are chosen at early stage. If some vertices are of maximum weight, one of the smallest degree is chosen (according to results of preliminary experiments). During this process, vertices are named $v_n, v_{n-1}, \cdots, v_1$ in sequence.

Algorithm 3 is to obtain tighter upper bounds by merging some subsets. Some consecutive independent sets are chosen to be merged unless the size of the new subset exceeds l. This process is performed until no subsets can be merged. The sets $P_1, P_2, \ldots, P_k$ are returned as the partition Π.

Algorithm 4 generates an optimal table for $V' \subseteq V$. The weights of the optimal solution for all possible subsets for each P_i are calculated, and saved in the optimal table corresponding to P_i. For example, the table for $P_1 = \{v_1, v_2, v_3\}$ has values of $w_{opt}(\emptyset)$, $w_{opt}(\{v_1\})$, $w_{opt}(\{v_2\})$, $w_{opt}(\{v_1, v_2\})$, $w_{opt}(\{v_3\})$, $w_{opt}(\{v_1, v_3\})$, $w_{opt}(\{v_2, v_3\})$, and $w_{opt}(\{v_1, v_2, v_3\})$. Note that the optimal tables are efficiently constructed with dynamic programming.

- It is obvious that $w_{opt}(\emptyset) = 0$.
- If all the values of $w_{opt}(S)$ for $S \subseteq V' \setminus \{v\}$ are known for a subset V' of V, $w_{opt}(Y)$ for Y such that $v \in Y \subseteq V'$ can be calculated from the

Algorithm 2 Generating independent sets

INPUT: An undirected graph $G = (V, E)$, vertex weight $w(\cdot)$ and size paramter l

OUTPUT: A vertex sequence $[v_n, v_{n-1}, \cdots, v_1]$ and Independent sets $I_1, I_2, \ldots$

procedure GENERATING_INDEPENDENT_SETS
  $X \leftarrow V$
  $j \leftarrow 0$
  while X is not empty **do**
    $j \leftarrow j + 1$
    $I_j \leftarrow \emptyset$
    $X' \leftarrow X$
    while $X' \neq \emptyset$ and $|I_j| < l$ **do**
      $i \leftarrow |X|$
      Let v_i be the vertex of maximum weight in X' (if there are some vertices of maximum weight, one of the smallest degree is chosen)
      $I_j \leftarrow I_j \cup \{v_i\}$
      $X' \leftarrow X' \backslash (\{v_i\} \cup N(v_i))$
      $X \leftarrow X \backslash \{v_i\}$
    end while
  end while
  return $[v_n, v_{n-1}, \ldots, v_1]$ and $I_1, I_2, \ldots, I_j$
end procedure

Algorithm 3 Generating a partition

INPUT: Independent sets $I_1, I_2, \ldots, I_j$, size parameter l

OUTPUT: A partition of V : $\Pi = (P_1, P_2, \ldots, P_k)$

procedure GENERATING_PARTITION
  $k \leftarrow 1$
  $P_1 \leftarrow \emptyset$
  for i from j downto 1 **do**
    if $|P_k| + |I_i| > l$ **then**
      $k \leftarrow k + 1$
      $P_k \leftarrow I_i$
    else
      $P_k \leftarrow P_k \cup I_i$
    end if
  end for
  return $(P_1, P_2, \ldots, P_k)$
end procedure

following equation :

$$w_{opt}(Y) = \max\{w(v) + w_{opt}(Y \cap N(v)) \ , w_{opt}(Y \setminus \{v\})\} \ . \tag{22}$$

The first argument of max operator is the value of optimum solution in case Y includes v, and the other is the one in case v is not included.

Algorithm 4 Generating an optimal table

INPUT: $G = (V, E)$, $w(\cdot)$ and $V' \subseteq V$
OUTPUT: $opt[\cdot]$ for all subsets of V'

```
1: procedure GENERATING_OPTIMAL_TABLE
2:     opt[∅] ← 0
3:     C ← {∅}
4:     V'' ← ∅
5:     while V'' is not V' do   ▷ At the beginning of each loop, any subset of V'' is in C.
6:         u ← an arbitrary vertex in V' \ V''
7:         C' ← ∅
8:         for X ∈ C do          ▷ For any X ∈ C, opt[X] is already calculated.
9:             Y ← X ∪ {u}
10:            opt[Y] ← max{w(u) + opt[X ∩ N(v_u)] , opt[X]}
11:            C' ← C' ∩ {Y}
12:        end for
13:        C ← C ∪ C'
14:        V'' ← V'' ∪ {u}
15:    end while
16:    return opt[·] for all subsets of V'
17: end procedure
```

2.4. Branch-and-bound phase

Hereafter, for the vertex sequence $[v_n, v_{n-1}, \cdots, v_1]$ obtained in the precomputation phase(Algorithm 2), V_i denotes $\{v_1, v_2, \ldots, v_i\}$ for simplicity. For a set of vertices V', $M(V')$ is the maximum index of vertices in V'. For example, $M(\{v_1, v_3, v_4, v_7\}) = 7$.

Algorithm 5 presents an outline of the branch-and-bound phase. The variables C, C_{max} and $c[\cdot]$ are global and can be accessed in the EXPAND proce-

dure. First, EXPAND(V_1) is called and the value $w_{opt}(V_1)$ is stored in $c[1]$. Next, EXPAND(V_2) is called and the value $w_{opt}(V_2)$ is stored in $c[2]$. Similarly, the values are stored in $c[3], c[4], \ldots$ at each iteration. When EXPAND(V_i) is called, the values $w_{opt}(V_1)$, $w_{opt}(V_2)$, ..., $w_{opt}(V_{i-1})$ are stored in $c[1]$, $c[2]$, ..., $c[i-1]$, respectively. It is obvious that $w_{opt}(V') \leq c[M(V')]$ for a subset $V' \subset V$ because $V' \subseteq V_{M(V')}$. Therefore, $c[M(V')]$ can be used as upper bounds for the subproblem $G(V')$. A subproblem is pruned by the bounding procedure if the upper bound is sufficiently small.

Algorithm 5 Branch-and-bound phase

INPUT: $G = (V, E)$, $w(\cdot)$, $\Pi = (P_1, P_2, \ldots, P_k)$ $opt[\cdot]$ and a sequence of vertices $[v_n, v_{n-1}, \cdots, v_1]$
OUTPUT: the maximum weight clique C_{max}
GLOBAL VARIABLES: C_{max}, C, $c[\cdot]$

```
1: determine the parameter α
2: C_max ← ∅
3: for i from 1 to ⌊αn⌋ do
4:     C ← ∅                          ▷ Initialize C to use in EXPAND( )
5:     EXPAND(V_i)
6:     c[i] ← w(C_max)      ▷ After EXPAND(V_i), C_max is the maximum weight clique of G(V_i).
7: end for
8: EXPAND(V)
```

Note that the upper bound of $c[M(V')]$ has been shown in a previous study [22]. We introduce a new parameter α due to the following observation. By some preliminary experiments, we confirmed that the value $c[i]$ is frequently used and causes pruning for small i; however it is rarely (or never) prunes subproblems for large i. Moreover, calculation of $c[i]$ for large i needs to solve a lot of subproblems. The results of preliminary experiments are shown in the Tables 1, 2 and 3. In the tables, the columns *used* means the number of times that $c[\cdot]$ is used as an upper bound. The columns *bounded* is the number of times that subproblems are pruned by $c[\cdot]$. The columns *subproblems* is the number of solved subproblems to calculate $c[\cdot]$. All values are the average of

10 randomgraphs. For $|V| = 200$, edge density= 0.9, 81.89% of subproblems are solved to calculate $c[181] - c[200]$ and they pruned only 83.4 subproblems. Therefore calculating such $c[\cdot]$ is not efficient strategy. Instead of calculating such $c[\cdot]$, we propose calculating $w_{opt}(V)$ directly after calculating $c[\lfloor \alpha n \rfloor]$. We have examined several different graphs and different values of α, and have determined that the proposed algorithm performs well on average when $\alpha = 0.8$.

Table 1: effectiveness of upper bounds $c[\cdot]$ for $|V| = 200$, edge density= 0.9 ($\alpha = 1$)

	used	bounded	subproblems	
$c[1] - c[20]$	826.9	0	104.6	(<0.01%)
$c[21] - c[40]$	3042.3	24.3	257.0	(<0.01%)
$c[41] - c[60]$	33572	454.2	971.7	(<0.01%)
$c[61] - c[80]$	362613.7	6463.6	5206.2	(<0.01%)
$c[81] - c[100]$	4312548.7	128316.1	33524.3	(0.02%)
$c[101] - c[120]$	42676994.1	1024640.3	166424.1	(0.1%)
$c[121] - c[140]$	253930088	8351456.5	1033750.7	(0.65%)
$c[141] - c[160]$	73743921.1	1695526.3	4973929.2	(3.11%)
$c[161] - c[180]$	1311731.8	19625.4	22738721.7	(14.23%)
$c[181] - c[200]$	2314.4	83.4	130887702.5	(81.89%)

Table 2: effectiveness of upper bounds $c[\cdot]$ for $|V| = 8000$, edge density= 0.1 ($\alpha = 1$)

	used	bounded	subproblems	
$c[1] - c[800]$	52558.4	15127.9	7419.0	(0.37%)
$c[801] - c[1600]$	246376.6	37310.2	25023.0	(1.26%)
$c[1601] - c[2400]$	429166.7	58495.4	35537.4	(1.79%)
$c[2401] - c[3200]$	769166.1	88722.5	50853.9	(2.56%)
$c[3201] - c[4000]$	1372352.4	128977.3	79919.4	(4.03%)
$c[4001] - c[4800]$	2060687.6	118745.2	129617.7	(6.54%)
$c[4801] - c[5600]$	2511924.6	96418.2	210840.0	(10.63%)
$c[5601] - c[6400]$	1947118.8	3455.8	330369.8	(16.66%)
$c[6401] - c[7200]$	855274.5	530.2	455283.4	(22.96%)
$c[7201] - c[8000]$	128104.9	0.0	657799.3	(33.18%)

The recursive procedure EXPAND($\cdot$) is shown in Algorithm 6. The steps from line 2 to line 7 correspond to process for leaf nodes in a search tree of branch-and-bound procedure. If a better solution is found, C_{max} is updated.

Table 3: effectiveness of upper bounds $c[\cdot]$ for $|V| = 1000$, edge density= 0.5 ($\alpha = 1$)

	used	bounded	subproblems	
$c[1] - c[100]$	5528.2	397.1	427.9	(<0.01%)
$c[101] - c[200]$	119690.4	10320.7	5817.6	(0.02%)
$c[201] - c[300]$	928299.7	96030.8	38283.9	(0.10%)
$c[301] - c[400]$	4329700.3	481179.8	132128.6	(0.36%)
$c[401] - c[500]$	14513157.3	2000919.5	484471	(1.31%)
$c[501] - c[600]$	40029819.7	3549796.3	1027140.8	(2.78%)
$c[601] - c[700]$	57663230.7	2105972.5	2529338.4	(6.85%)
$c[701] - c[800]$	28002863.9	130138.7	5343812.8	(14.48%)
$c[801] - c[900]$	4155355.9	1406	10555860.2	(28.60%)
$c[901] - c[1000]$	102980.4	27.6	16785686.5	(45.49%)

The bounding procedure is the steps from line 8 to line 10. In line 8, the upper bounds $UB(\cdot, \cdot)$ and $c[\cdot]$ are calculated, and the subproblem is pruned if one of the upper bounds is sufficiently small. In line 11, the vertex of the maximum index is chosen as a branching variable u, so that $M(V')$ gets smaller. In the rest of the algorithm, subproblems of $G(V')$ are examined in the following order.

- search the optimum solution in the subgraph $G(V' \cap N(u))$. (line 13)
- search the optimum solution in the subgraph $G(V' \setminus \{u\})$. (line 15)

For example, if $G(V)$ in Figure 1 is given, the algorithm searches the optimum solution in $G(V \cap N(v_8))$, i.e., $G(\{v_1, v_3, v_4, v_5\})$. Next, the algorithm searches the optimum solution in $G(V_7)$.

Algorithm 6 Solving a subproblem

INPUT: $V' \in V$ ▷ For any $v \in V'$, $C \subseteq N(v)$
OUTPUT: Update C_{max} if better cliques are found.
GLOBAL VARIABLES: C_{max}, C, $c[\cdot]$

```
1: procedure EXPAND(V')
2:     if V' = ∅ then                                   ▷ Recursive calls finished.
3:         if w(C) > w(C_max) then
4:             C_max ← C
5:         end if
6:         return
7:     end if
8:     if UB(Π, V') + w(C) ≤ w(C_max) or c[M(V')] + w(C) ≤ w(C_max) then
       ▷ Bounding procedure by two upper bounds.
9:         return
10:    end if
11:    u ← v_{M(V')}
12:    C ← C ∪ {u}
13:    EXPAND(V' ∩ N(u))              ▷ Solve subproblems where C includes u.
14:    C ← C \ {u}
15:    EXPAND(V' \ {u})   ▷ Solve subproblems where C does not includes u.
16: end procedure
```

3. A case study

In this section, we show an example for OTClique.

3.1. Precomputation phase example

Given an undirected graph shown in of Figure 3a, OTClique constructs a vertex sequence and independent sets shown in Figure 3b by Algorithm 2. When the input parameter l is given as 3, Algotirhm 3 merges I_2 and I_3 to P_2. Also, I_4 and I_5 are merged to P_1.

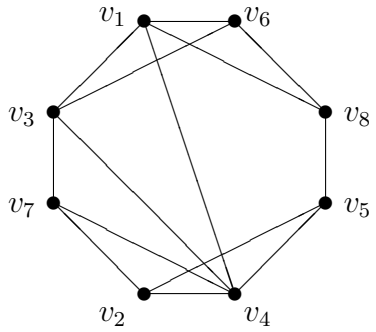
Any vertex set is represented by an array of bit vectors. Each bit vector corresponds to a vertex subset P_i and each bit is corresponds to a vertex in P_i. For the vertex partition shown in Figure 3b, bit vector representations for some vertex sets are shown in Figure 3c.

Optimal Tables is implemented with two-dimensional arrays shown in Figure 3d. Any subsets of P_i is represented by a bit vector. For example, 011 for P_2 means $\{v_5, v_4\}$. Therefore, $w_{opt}(\{v_5, v_4\}) = 8$ can be obtained from $table[2][011]$ in O(1) time.

3.2. Upper bound calculation example

For the graph shown in Figure 3a, a vertex subset $S = \{v_2, v_3, v_5, v_6, v_7\}$ is represented by an array of bit vectors $\{01, 110, 110\}$. Hence, an upper bound $UB(\Pi, S)$ can be calculated with optimal tables as follows:

$$\begin{aligned} UB(\Pi, S) &= table[3][01] + table[2][110] + table[1][110] \\ &= 5 + 6 + 3 \\ &= 14\ . \end{aligned} \tag{23}$$



(a) An undirected graph

vertex	v_8	v_7	v_6	v_5	v_4	v_3	v_2	v_1
weight	7	5	6	4	4	3	1	2
independent sets	I_1		I_2		I_3	I_4		I_5
partition	P_3		P_2			P_1		

(b) Vertex sequence, independent sets and partition

V : $\{11, 111, 111\}$
P_3 : $\{11, 000, 000\}$
P_2 : $\{00, 111, 000\}$
P_1 : $\{00, 000, 111\}$
$N(v_3)$: $\{01, 101, 001\}$
V_5 : $\{00, 011, 111\}$

(c) Bit vector representation examples

	3	2	1
000	0	0	0
001	5	4	2
010	7	4	1
011	7	8	2
100	-	6	3
101	-	6	5
110	-	6	3
111	-	8	5

(d) Optimal tables

Figure 3: A precomputation example

4. Numerical experiments

We implemented OTClique in C. We determined $l = 25$ for graphs with $n \leq 1500$, otherwise $l = 20$. We compared OTClique with Östergård's algorithm [22], Yamaguchi/Masuda's algorithm [32] (denoted YM), Kumlander's algorithm [20] (denoted DK), our previous algorithm *VCTable* [21] and IBM's mixed integer programming solver CPLEX. For CPLEX, we formulated MWCP with integer programming as follows :

$$\begin{aligned} \text{maximize} \quad &: \quad \sum_{v_i \in V} w(v_i) \cdot x_i \\ \text{s.t.} \quad &: \quad x_i + x_j \leq 1, \quad (v_i, v_j) \notin E \\ &\quad\quad x_i \in \{0, 1\}, \quad \forall v_i \in V \ . \end{aligned}$$

We used the C program *Cliquer* [33] for Östergård's algorithm. For YM, we used a C++ implementation [32]. For VCTable, we used our own C implementation [21]. Although Kumlander presents a Visual Basic 6.0 implementation [34], we independetly implemented DK in C to avoid performance variations between VB and C. We used an Intel(R) Core(TM) i7-2600 3.40 GHz, 8 GB of main memory, and GNU/Linux. The compiler was gcc 4.4.6 (optimization option -O2). In addition, version 12.5.0.0. of CPLEX was used. Note that CPLEX is a multi-thread solver, and the others are single-thread solvers. In our computer experiments, the CPU usage was approximately 800% for CPLEX, and the CPU usage for the others was approximately 100%.

In the result tables, n denotes the number of vertices, d denotes the edge density $\frac{2|E|}{|V|(|V|-1)}$, *pre* denotes the computation time for the precomputation phase, and *total* denotes the total computation time, which includes the precomputation phase.

4.1. Random graphs

We generated uniform random graphs with various numbers of vertices and edge density. The vertex weights were integer values ranging from 1 to 10. In each case, we generated 10 instances and calculated the average computation time and number of branches.

The computation times and their summary are shown in Table 4 and 5, respectively. In Table 5, the values of minimum, geometric mean and maximum value of the ratio of each algorithm to OTClique are shown. Some unknown values (over 1000) are assumed 1000 for convenience in that calculation.

As can be seen, the proposed OTClique algorithm and VCTable can solve all instances; however, the others cannot solve some instances. For most graphs with $0.3 \leq d \leq 0.9$, OTClique is faster than the other algorithms. Although the computation time for the precomputation phase is exponential to the size of P_i, it is actually performed in less than 2 seconds. For graphs with $d \leq 0.2$, Cliquer is faster than OTClique. However, Cliquer is very slow for dense graphs.

For graphs with $d \geq 0.95$, CPLEX is faster than OTClique. We also performed some experiments for CPLEX with a fixed number of vertices and the results are shown in Table 6. CPLEX is very slow even if the graph is sparse. Note that CPLEX is a branch-and-cut based solver; thus, it behaves quite differently from other branch-and-bound-based algorithms.

The number of search tree nodes and its summary are shown in Table 7 and 8, respectively. The number of nodes of Cliquer is not shown because the program does not provide this information. In most cases, the YM algorithm demonstrates the smallest number of search tree nodes. However, OTClique is faster than the YM algorithm because OTClique calculates an upper bound in $O(|V'|)$ time for a subproblem V', whereas the YM algorithm requires $O(|V'|^2)$ time for the upper bound calculation. Since similar tendency is also seen in the

Table 4: Computation time for random graphs [sec]

		OTClique							
n	d	l	pre	total	VCTable	Cliquer	YM	DK	CPLEX
8000	0.1	20	0.97	5.09	6.54	2.69	13.24	12.01	>1000
6000	0.1	20	0.68	1.98	2.47	1.14	4.55	4.05	>1000
4000	0.2	20	0.43	6.72	9.04	4.15	25.78	20.94	>1000
3000	0.2	20	0.30	2.03	2.70	1.36	6.08	5.99	>1000
2500	0.3	20	0.07	9.38	15.95	10.03	37.84	44.53	>1000
2000	0.3	20	0.05	3.00	8.37	3.47	12.30	14.61	>1000
1500	0.4	25	1.51	9.08	14.45	15.06	42.29	58.04	>1000
1000	0.4	25	0.98	1.69	1.24	1.58	3.62	5.30	>1000
1000	0.5	25	0.88	11.67	17.85	28.67	61.50	94.84	>1000
900	0.5	25	0.74	5.74	10.29	15.29	31.46	50.73	>1000
700	0.6	25	0.49	17.02	29.99	64.38	99.07	212.70	>1000
500	0.6	25	0.38	1.72	2.48	5.62	7.11	17.12	>1000
500	0.7	25	0.44	36.57	66.35	212.79	201.98	674.06	>1000
300	0.7	25	0.31	0.72	0.96	3.13	2.01	6.49	769.63
300	0.8	25	0.33	18.85	52.88	242.38	93.70	511.97	>1000
200	0.8	25	0.23	0.45	0.71	3.29	1.11	5.45	24.97
200	0.9	25	0.30	10.63	60.44	>1000	89.85	409.55	11.24
150	0.9	25	0.21	0.46	0.96	20.32	1.58	7.20	0.89
200	0.95	25	0.30	144.11	909.88	>1000	>1000	>1000	1.96
150	0.95	25	0.21	2.06	6.49	>1000	22.57	48.09	0.27
200	0.98	25	0.34	18.75	40.44	>1000	>1000	967.34	0.02
150	0.98	25	0.26	0.43	0.47	>1000	18.70	5.09	0.01

Table 5: Summary: Computation time comparison for random graphs

	VCTable	Cliquer	YM	DK	CPLEX
min	0.73	0.53	2.14	2.05	0.014
mean	1.87	>4.10	>4.68	>8.79	>49.4
max	6.31	>2325.6	>53.3	>53.3	>2222.2

experiments with other data, we do not show the summary of number of search tree nodes hereafter.

4.2. Graphs from error-correcting codes

Error-correcting codes are important in the field of coding theory. The problem of constructing error-correcting codes of maximum size can be formulated with the MWCP [22].

The computation time, its summary and number of search tree nodes are shown in Tables 9, 10 and 11, respectively. OTClique, VCTable, and Cliquer can

Table 6: Computation time and number of search tree nodes of CPLEX

		CPLEX		
n	d	time[sec]	iterations	nodes
200	0.1	5.57	2753.7	0.0
200	0.2	7.54	5380.3	0.0
200	0.3	10.76	36948.7	350.5
200	0.4	12.44	83930.5	818.8
200	0.5	11.16	108395.2	1303.2
200	0.6	12.31	260316.9	3355.7
200	0.7	14.76	524246.0	7251.2
200	0.8	25.78	1185288.8	21259.0
200	0.9	11.46	886880.4	17404.3
200	0.95	2.00	116176.7	2642.0
200	0.98	0.02	238.1	0.0

Table 7: Number of search tree nodes for random graphs

						CPLEX	
n	d	OTClique	VCTable	YM	DK	iterations	nodes
8000	0.1	1569577.2	1969224	2984515.8	1945414.9		
6000	0.1	538047.3	662396.4	1476080.5	650523.3		
4000	0.2	4209559.4	5242273.7	2914794.5	5959659.5		
3000	0.2	1600052.8	1962662	1093185.9	2281087.4		
2500	0.3	12634956.5	16158044.8	9341501.7	20388682.5		
2000	0.3	4499188	5931297.5	3174379.3	7096941.7		
1500	0.4	15814530.7	22526798.8	9558008.3	29162795.8		
1000	0.4	1850807	2818775	1494500.5	3295741.8		
1000	0.5	28515581.8	42783759.7	15942518.3	57438717		
900	0.5	15621489.3	25828420.6	8696737.4	31984389.3		
700	0.6	64268108	110461661.2	25579875.4	142408095.5		
500	0.6	5855125.4	10794560.6	3112917.3	13138380.5		
500	0.7	174437626.4	345544601.4	57386087.3	490450681.3		
300	0.7	2166003	5486785.2	1398292.1	6078605.7	13761832	199887.2
300	0.8	115162693.3	369686388.9	39410102.6	470340625.1		
200	0.8	1325098.1	5282665.4	1098077	6256181.4	1185288.8	21259
200	0.9	92658142.3	593489366.4	24300759.4	513470878.2	886880.4	17404.3
150	0.9	2249004.1	9622659.3	1210309.3	11607417.5	39049.1	999.7
200	0.95	1648509971.5	10439082379			116176.7	2642
150	0.95	23690554.4	80717090.8	5716143.1	94504883	2190.3	34.9
200	0.98	252017914.8	590062099.6		1755388755.3	238.1	0.0
150	0.98	2914216.4	6804953.8	2538974.4	11978621.5	84.9	0.0

Table 8: Summary: Comparison of Number of search tree nodes (random graph, $0.1 \leq d \leq 0.9$)

	VCTable	YM	DK
min	1.226623271	0.262262536	1.209044818
mean	1.914480733	0.642868328	2.254038621
max	6.405150715	2.743402857	5.541562408

solve all instances; however, YM, DK and CPLEX cannot solve some instances within 1000 seconds. There is a difference from the experiments for random graphs; Cliquer is the fastest for random sparse graphs. However, in these experiments OTClique is often faster than Cliquer even though all graphs are very sparse.

Table 9: Computation time for graphs from error-correcting codes[sec]

			OTClique							
instance	n	d	l	pre	total	VCTable	Cliquer	YM	DK	CPLEX
11-4-4	150	0.089	25	0.17	3.50	13.96	18.45	11.52	151.45	2.49
12-4-6	230	0.038	25	0.20	18.62	92.56	18.22	54.67	>1000	8.14
14-4-7	223	0.040	25	0.17	24.84	71.98	248.41	58.29	455.17	177.13
14-6-6	807	0.0031	25	0.58	11.62	18.18	12.56	33.05	183.83	>1000
16-4-5	156	0.083	25	0.15	0.23	0.22	0.11	0.34	1.72	0.51
16-8-8	2246	0.00040	20	0.21	0.30	0.37	0.13	0.24	0.40	>1000
17-4-4	132	0.12	25	0.09	0.09	0.03	0.03	0.02	0.12	0.21
17-6-6	558	0.0064	25	0.34	8.43	50.52	45.00	11.65	57.66	563.01
19-4-6	263	0.029	25	0.25	1.55	1.75	0.43	>1000	>1000	12.37
19-8-8	2124	0.00044	20	0.20	1.44	2.03	1.11	4.58	5.03	>1000
20-6-5	1302	0.0012	25	2.22	15.55	16.07	8.48	13.76	71.58	>1000
20-6-6	1490	0.00090	25	1.28	35.98	36.15	39.42	27.53	122.47	>1000
20-8-10	2510	0.00032	20	0.26	0.67	0.68	0.47	0.44	0.93	>1000
21-10-9	5098	0.000077	20	0.57	30.80	36.48	22.30	45.58	81.61	>1000
22-10-10	8914	0.000025	20	1.07	2.18	2.45	2.72	4.99	3.98	>1000

Table 10: Summary: Computation time comparison for error-correcting codes

	VCTable	Cliquer	YM	DK	CPLEX
min	0.33	0.28	0.22	1.33	0.44
mean	1.51	1.03	>2.11	>7.54	>32.5
max	5.99	10.0	>645.1	>645	>3333

4.3. Combinatorial auction test suite (CATS)

The winner determination problem (WDP) is a problem to find the winner in a combinatorial auction, which allows a bidder to bid on some combinations of items. In the WDP, a set of items S and a set of bids B are given. Each bid is given as a subset A_i of items and a price $p[i]$. Any two bids containing the same item cannot simultaneously be winners. Winners are determined to maximize the sum of the profit.

Table 11: Number of search tree nodes for graphs from error-correcting codes

							CPLEX	
instance	n	d	OTClique	VCTable	YM	DK	iterations	nodes
11-4-4	150	0.089	30163536	180331784	25070647	275400718	397710	8549
12-4-6	230	0.038	121920915	668899369	17792987		751526	10948
14-4-7	223	0.04	143755269	458811161	41662592	515680460	19249470	563040
14-6-6	807	0.0031	30487685	46843021	4601492	79528959		
16-4-5	156	0.083	398777	1540169	338154	2316548	31621	542
16-8-8	2246	0.0004	124722	149419	179727	169038		
17-4-4	132	0.12	24780	178479	56609	189553	32334	1925
17-6-6	558	0.0064	49123059	364028794	5854424	46348199	8681379	75387
19-4-6	263	0.029	14506697	18569063			1213989	22654
19-8-8	2124	0.00044	3197921	4590127	2613216	1824505		
20-6-5	1302	0.0012	36969905	41428506	15508514	47977328		
20-6-6	1490	0.0009	72883176	78077559	21910613	75449886		
20-8-10	2510	0.00032	567832	611175	402282	510480		
21-10-9	5098	0.000077	48434957	59905383	19987039	45192855		
22-10-10	8914	0.000025	244678	243124	2134959	198318		

The WDP can be formulated by integer programming as follows:

$$\begin{aligned} \text{maximize} \quad &: \quad \sum_{b_i \in B} p[i] x_i \\ \text{s.t.} \quad &: \quad \sum_{A_i \ni s_j} x_i \leq 1, \text{for } \forall s_j \in S \\ &\quad\quad x_i \in \{0,1\}, \ \forall b_i \in B \ . \end{aligned}$$

CATS, the benchmark set of the WDP, is available online [6]. CATS can create instances of the CPLEX integer programming format. We obtained MWCP with graph $G = (V, E)$ and weights for each vertex $w(\cdot)$ from the WDP by transforming in the following manner. Vertices corresponds to bids, and for any two bids $b_i, b_j \in B$, there exists an edge (v_i, v_j) iff $A_i \cap A_j = \emptyset$. Each vertex weight is the price of each corresponding bid.

In the experiments, 10 instances were generated for each condition, and the average computation time, its summary and number of search tree nodes are shown in Tables 12, 13 and 14, respectively. In the tables, *arbitrary-400-250* denotes the instance of the *arbitrary* distribution with 400 items and 250 bids.

CATS does not produce instances of an exact number of bids; thus, the numbers in column "n" differ slightly from the expected numbers.

In these experiments CPLEX was the fastest for almost all instances, probably because of small n and large d. In addition, the outputs of CATS might be in a more desirable formulation for CPLEX. Among the branch-and-bound algorithms, OTClique is significantly faster than other algorithms.

Table 12: Computation time for CATS [sec]

			OTClique							
instance	n	d	l	pre	total	VCTable	Cliquer	YM	DK	CPLEX
arbitrary-400-250	251.5	0.71	25	0.25	0.38	0.71	16.95	0.44	2.36	5.27
arbitrary-700-200	202.1	0.81	25	0.18	0.21	0.57	224.92	0.08	0.27	1.09
matching-400-300	304.7	0.96	25	0.11	0.13	>1000	>1000	115.52	6.74	0.01
matching-700-250	251.9	0.96	25	0.05	0.05	>1000	>1000	0.13	1.14	0.01
paths-100-200	201.4	0.85	25	0.19	2.41	71.62	>1000	38.30	44.84	0.01
paths-150-200	202.1	0.86	25	0.16	5.10	209.27	>1000	17.98	70.55	0.01
regions-500-300	302.1	0.86	25	0.22	6.48	>1000	>1000	21.62	200.18	0.28
regions-700-250	252.3	0.90	25	0.20	0.72	725.61	>1000	3.21	27.86	0.09
scheduling-30-600	614.0	0.60	25	0.83	1.86	4.35	5.95	19.67	3.91	0.02
scheduling-50-500	516.9	0.71	25	0.65	2.84	3.33	7.72	32.14	2.47	0.08

Table 13: Summary: Computation time comparison for CATS

	VCTable	Cliquer	YM	DK	CPLEX
min	1.17	2.72	0.38	0.87	0.002
mean	>57.7	>254	6.32	9.39	0.083
max	>20000	>20000	889	51.8	13.87

Table 14: Number of search tree nodes for CATS

							CPLEX	
instance	n	d	OTClique	VCTable	YM	DK	iterations	nodes
arbitrary-400-250	251.5	0.71	888981.3	5656266.7	188516.6	3516208.6	640828.9	12708.4
arbitrary-700-200	202.1	0.81	163751.7	5009649.8	22621.2	473788.1	52123.4	829.5
matching-400-300	304.7	0.96	266306.1		5085000.7	4456469.6	10.2	0.0
matching-700-250	251.9	0.96	105082.2		5988.5	1644554.7	2.6	0.0
paths-100-200	201.4	0.85	19408098.7	643188530.2	18514793.1	85357910.2	82.2	0.0
paths-150-200	202.1	0.86	45093901.5	2071898703	9021836.7	129116819.3	85.3	0.0
regions-500-300	302.1	0.86	50251614.2		1787653.7	206599782.9	3793.3	47.7
regions-700-250	252.3	0.90	5090924.0	6315728338.5	363420.6	35247529.6	543.4	0.0
scheduling-30-600	614.0	0.60	5114551.6	33353543.8	1939830.5	2924728.7	99.8	0.0
scheduling-50-500	516.9	0.71	16712513.2	40218416.4	2811068.1	2435736.2	907.1	61.3

4.4. DIMACS benchmark graphs

The DIMACS benchmarks for the MCP can be obtained online [35]. We used the DIMACS benchmarks to compare weighted algorithms. Note that there are some faster algorithms for the MCP (e.g., [14]) than algorithms for the MWCP.

The computation time, its summary and the number of search tree nodes are shown in Tables 15, 16 and 17, respectively. In the tables, "easy instance" means the instance which at least one of algorithms can solve less than 0.1 second (26 insntaces) and "hard instance" means all the algorithm takes at least 0.1 second (16 instances). In Table 15, we put "*" at the end of each row for "easy instance". In Table 16, the number of times that each algorithm is the fastest is shown.

For "easy instances", OTClique is not the fastest because the time required to perform the precomputation phase is relatively long for easy instances (still less than a second). However, for "hard instances", OTClique is several times faster than other algorithms in most cases. For example, previous algorithms require at least 13 hours to solve *p-hat500-3*; however, OTClique can solve the problem within 30 minutes.

Table 15: Computation time for DIMACS graphs [sec]

			OTClique								
instance	n	d	l	pre	total	VCTable	Cliquer	YM	DK	CPLEX	easy
brock200_1	200	0.75	25	0.23	0.75	2.14	3.78	3.03	12.99	155.13	
brock200_2	200	0.50	25	0.05	0.06	0.02	0.01	0.02	0.02	29.48	*
brock200_3	200	0.61	25	0.20	0.22	0.09	0.07	0.09	0.28	44.33	*
brock200_4	200	0.66	25	0.13	0.17	0.12	0.27	0.25	0.60	57.00	
brock400_1	400	0.75	25	0.27	627.16	2959.93	13192.62	1801.04	35059.79	>24h	
brock400_2	400	0.75	25	0.36	99.37	1540.65	3354.88	1927.09	13208.17	out of memory	
brock400_3	400	0.75	25	0.40	474.01	337.01	994.56	1718.99	3485.41	out of memory	
brock400_4	400	0.75	25	0.32	41.29	672.79	146.68	1855.58	3148.39	out of memory	
c-fat200-1	200	0.08	25	0.02	0.02	0.01	<0.01	<0.01	<0.01	4.25	*
c-fat200-2	200	0.16	25	0.02	0.02	0.01	<0.01	<0.01	<0.01	2.97	*
c-fat200-5	200	0.43	25	0.26	0.26	48.54	0.11	<0.01	<0.01	1.44	*
c-fat500-10	500	0.37	25	0.50	0.51	0.22	<0.01	0.01	0.01	30.60	*
c-fat500-1	500	0.04	25	0.97	0.97	0.05	<0.01	<0.01	<0.01	40.41	*
c-fat500-2	500	0.07	25	0.06	0.07	0.06	<0.01	<0.01	<0.01	52.04	*
c-fat500-5	500	0.19	25	0.50	0.50	0.01	<0.01	<0.01	<0.01	50.34	*
hamming6-2	64	0.90	25	0.06	0.07	0.04	<0.01	<0.01	<0.01	0.01	*
hamming6-4	64	0.35	25	0.06	0.07	<0.01	<0.01	<0.01	<0.01	0.09	*
hamming8-4	256	0.64	25	0.26	0.26	<0.01	<0.01	0.06	<0.01	0.31	*
johnson16-2-4	120	0.76	25	0.10	0.11	0.06	0.02	0.02	0.09	0.01	*
johnson8-2-4	28	0.56	25	0.01	0.01	<0.01	<0.01	<0.01	<0.01	<0.01	*
johnson8-4-4	70	0.77	25	0.11	0.11	<0.01	<0.01	<0.01	<0.01	0.01	*
keller4	171	0.65	25	0.13	0.13	0.06	0.06	0.04	0.31	1.86	*
MANN_a9	45	0.93	25	0.04	0.04	0.01	<0.01	<0.01	<0.01	0.01	*
p_hat1000-1	1000	0.24	25	0.19	0.58	0.49	0.68	0.78	2.05	out of memory	
p_hat1000-2	1000	0.49	25	0.58	5473.09	22235.24	>24h	>24h	>24h	out of memory	
p_hat1500-1	1500	0.25	25	0.49	3.47	4.47	5.02	7.25	13.37	2316.46	
p_hat300-1	300	0.24	25	0.07	0.07	0.03	0.01	<0.01	<0.01	197.70	*
p_hat300-2	300	0.49	25	0.16	0.17	0.08	0.16	1.55	1.57	201.27	*
p_hat300-3	300	0.74	25	0.30	2.30	32.88	290.47	1214.72	3284.28	out of memory	
p_hat500-1	500	0.25	25	0.06	0.08	0.06	0.04	0.04	0.07	5901.71	*
p_hat500-2	500	0.50	25	0.27	0.68	3.51	79.55	407.64	171.05	out of memory	
p_hat500-3	500	0.75	25	0.46	1688.62	47835.44	>24h	>24h	>24h	out of memory	
p_hat700-1	700	0.25	25	0.10	0.13	0.08	0.06	0.12	0.17	70610.60	*
p_hat700-2	700	0.50	25	0.37	25.02	103.99	9095.34	54845.59	14432.73	out of memory	
san200_0.7_1	200	0.70	25	0.13	0.14	0.21	0.48	0.01	10.79	0.07	*
san200_0.7_2	200	0.70	25	0.10	0.10	<0.01	<0.01	0.04	509.60	1.43	*
san200_0.9_1	200	0.90	25	0.28	0.28	2.01	0.06	0.21	296.46	0.02	*
san200_0.9_2	200	0.90	25	0.22	0.24	9.21	6.96	33.05	100.63	0.03	*
san200_0.9_3	200	0.90	25	0.33	39.16	3216.09	288.78	353.92	66606.68	1.16	
san400_0.5_1	400	0.50	25	0.43	0.43	86.11	<0.01	0.07	9.51	20.09	*
sanr200_0.7	200	0.70	25	0.15	0.30	0.56	1.14	0.78	3.37	81.45	
sanr400_0.5	400	0.50	25	0.31	0.54	0.61	0.61	0.75	1.78	out of memory	

Table 16: Number of times the algorithm is fastest

	OTClique	VCTable	Cliquer	YM	DK	CPLEX
easy instance	0	6	18	15	13	4
hard instance	12	3	0	0	0	1

Table 17: Number of search tree nodes for DIMACS graphs

							CPLEX	
instance	n	d	OTClique	VCTable	YM	DK	iterations	nodes
Brock200_1	200	0.75	2566471	13661530	6059104	15797409	7436306	208755
Brock200_2	200	0.50	9657	31882	15590	30566	661974	9741
Brock200_3	200	0.61	89289	315209	198252	340176	1344242	33307
brock200_4	200	0.66	172145	442065	515922	802497	2017472	51940
brock400_1	400	0.75	3488294654	16039214046	2200796435	35382562365		
brock400_2	400	0.75	553204147	8052675173	2576827212	13386023608		
brock400_3	400	0.75	2879648537	1962741399	2400677118	3904864834		
brock400_4	400	0.75	238804823	3890878794	2721130159	3302953612		
c-fat200-1	200	0.08	82	83	19	102	803	0
c-fat200-2	200	0.16	300	300	55	324	708	0
c-fat200-5	200	0.43	1712	308553765	528	1923	846	0
c-fat500-10	500	0.37	8001	8001	186	8127	3029	0
c-fat500-1	500	0.04	108	128	34	119	3838	0
c-fat500-2	500	0.07	351	351	91	377	4129	0
c-fat500-5	500	0.19	2080	2080	97	2144	3251	0
hamming6-2	64	0.90	528	548	32	584	87	0
hamming6-4	64	0.35	49	80	106	88	470	0
hamming8-4	256	0.64	972	1171	38508	1179	1351	0
johnson16-2-4	120	0.76	218423	541587	228719	547373	47	0
johnson8-2-4	28	0.56	10	45	24	51	10	0
johnson8-4-4	70	0.77	232	323	363	499	117	0
keller4	171	0.65	39213	181697	82173	437495	175708	3006
MANN_a9	45	0.93	204	704	1893	2845	97	0
p_hat1000-1	1000	0.24	1220187	1450847	986204	1589646		
p_hat1000-2	1000	0.49	20048937586	143300483577				
p_hat1500-1	1500	0.25	6923351	13656285	7116928	9335304	28604	0
p_hat300-1	300	0.24	3986	5276	5229	6282	402470	4936
p_hat300-2	300	0.49	53829	213246	1597449	1702352	737357	8884
p_hat300-3	300	0.74	11044720	219293999	708443913	4411515345		
p_hat500-1	500	0.25	48296	67740	56053	62412	4300997	70519
p_hat500-2	500	0.50	1830991	19329531	99860159	223499388		
p_hat500-3	500	0.75	9900409369	308959913207				
p_hat700-1	700	0.25	71443	158620	134150	141524	24148763	363967
p_hat700-2	700	0.50	106049123	621465294	6995805088	17558655841		
san200_0.7_1	200	0.70	78107	1204248	2393	75571516	588	0
san200_0.7_2	200	0.70	248	225	31353	1273043561	44602	668
san200_0.9_1	200	0.90	7052	22252694	11224	1601201049	319	0
san200_0.9_2	200	0.90	268649	98942797	14354090	347488437	520	0
san200_0.9_3	200	0.90	332503947	27913769597	235811036	140175563921	101776	1737
san400_0.5_1	400	0.50	1123	119048473	4892	5318885	207102	920
sanr200_0.7	200	0.70	709769	3139173	1626325	4235239	2930622	73971
sanr400_0.5	400	0.50	973190	2476362	1252324	1715154		

5. Conclusions

We have proposed a new maximum clique extraction algorithm OTClique. OTClique consists of two phases, the precomputation phase and the branch-and-bound phase. In the precomputation phase, the proposed OTClique algorithm generates a vertex partition and optimal tables. In the branch-and-bound phase, OTClique calculates the upper bound in a very short time using the optimal tables. Because the computation time for each branch is very short and the bounding procedure can prune significant search space; thus, OTClique can solve instances quickly.

From the experiments, we have confirmed that OTClique is significantly faster than other algorithms for almost all instances. For some instances, OTClique is not the fastest; however, the differences are not significant. OTClique solves such instances nearly as fast as the fastest performing algorithm in such cases. Previous algorithms cannot find the optimum solution for some instances; however, OTClique can find the optimum solution for all instances used in the experiments.

[1] M. R. Gary, D. S. Johnson, Computers and Intractability - A Guide to the Theory of NP-completeness, W H Freeman and Company, 1979.

[2] G. T. Bogdanova, A. E. Brouwer, S. N. Kapralov, P. R. Östergård, Error-correcting codes over an alphabet of four elements, Designs, Codes and Cryptography 23 (3) (2001) 333–342.

[3] S. Sorour, S. Valaee, Minimum broadcast decoding delay for generalized instantly decodable network coding, in: Global Telecommunications Conference (GLOBECOM 2010), IEEE, 2010, pp. 1–5.

[4] R. Horaud, T. Skordas, Stereo correspondence through feature grouping

and maximal cliques, IEEE Transactions on Pattern Analysis and Machine Intelligence 11 (11) (1989) 1168–1180.

[5] D. B. KC, T. Akutsu, E. Tomita, T. Seki, A. Fujiyama, Point matching under non-uniform distortions and protein side chain packing based on efficient maximum clique algorithms, Genome Informatics 13 (2002) 143–152.

[6] K. L. Brown, Combinatorial auction test suite (CATS), `http://www.cs.ubc.ca/~kevinlb/CATS/` (2000).

[7] L. Babel, A fast algorithm for the maximum weight clique problem, Computing 52 (1) (1994) 31–38.

[8] E. Balas, J. Xue, Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring, Algorithmica 15 (5) (1996) 397–412.

[9] M. Batsyn, B. Goldengorin, E. Maslov, P. M. Pardalos, Improvements to mcs algorithm for the maximum clique problem, Journal of Combinatorial Optimization 27 (2) (2014) 397–416.

[10] P. San Segundo, D. Rodríguez-Losada, A. Jiménez, An exact bit-parallel algorithm for the maximum clique problem, Computers & Operations Research 38 (2) (2011) 571–581.

[11] E. C. Sewell, A branch and bound algorithm for the stability number of a sparse graph, INFORMS Journal on Computing 10 (4) (1998) 438–447.

[12] M. Shindo, E. Tomita, A simple algorithm for finding a maximum clique and its worst-case time complexity, Systems and Computers in Japan 21 (3) (1990) 1–13.

[13] E. Tomita, T. Seki, An efficient branch-and-bound algorithm for finding a maximum clique, in: Discrete Mathematics and Theoretical Computer Science, Springer, 2003, pp. 278–289.

[14] E. Tomita, T. Kameda, An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments, Journal of Global optimization 37 (1) (2007) 95–111.

[15] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, M. Wakatsuki, A simple and faster branch-and-bound algorithm for finding a maximum clique, in: WALCOM: Algorithms and computation, Springer, 2010, pp. 191–203.

[16] D. R. Wood, An algorithm for finding a maximum clique in a graph, Operations Research Letters 21 (5) (1997) 211–217.

[17] D. Kumlander, A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search, in: Proceedings of the 5th International Conference on Modelling, Computation and Optimization in Information Systems and Management Sciences, Citeseer, 2004, pp. 202–208.

[18] D. Kumlander, Improving the maximum-weight clique algorithm for the dense graphs, in: Proceedings of the 10th WSEAS International Conference on COMPUTERS, 2006, pp. 938–943.

[19] D. Kumlander, 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) (2006) 32–49.

[20] D. Kumlander, On importance of a special sorting in the maximum weight clique algorithm based on colour classes, in: Proceedings of the second international conference on Modelling, Computation and Optimization in

Information Systems and Management Sciences Communications in Computer and Information Science, 2008, pp. 165–174.

[21] S. Shimizu, K. Yamaguchi, T. Saitoh, S. Masuda, Some improvements on Kumlander's maximum weight clique extraction algorithm, in: International Conference on Electrical, Computer, Electronics and Communication Engineering (ICECECE 2012), World Academy of Science, Engeneering and Technology, Issue 72, 2012, pp. 307–311.

[22] P. R. Östergård, A new algorithm for the maximum-weight clique problem, Nordic Journal of Computing 8 (4) (2001) 424–436.

[23] P. R. Östergård, A fast algorithm for the maximum clique problem, Discrete Applied Mathematics 120 (1) (2002) 197–207.

[24] E. Balas, C. S. Yu, Finding a maximum clique in an arbitrary graph, SIAM Journal on Computing 15 (4) (1986) 1054–1068.

[25] E. Balas, J. Xue, Minimum weighted coloring of triangulated graphs, with application to maximum weight vertex packing and clique finding in arbitrary graphs, SIAM Journal on Computing 20 (2) (1991) 209–221.

[26] R. Carraghan, P. M. Pardalos, An exact algorithm for the maximum clique problem, Operations Research Letters 9 (6) (1990) 375–382.

[27] P. M. Pardalos, N. Desai, An algorithm for finding a maximum weighted independent set in an arbitrary graph, International Journal of Computer Mathematics 38 (3-4) (1991) 163–175.

[28] P. M. Pardalos, G. P. Rodgers, A branch and bound algorithm for the maximum clique problem, Computers & operations research 19 (5) (1992) 363–375.

[29] P. M. Pardalos, J. Xue, The maximum clique problem, Journal of global Optimization 4 (3) (1994) 301–328.

[30] S. Rebennack, G. Reinelt, P. M. Pardalos, A tutorial on branch and cut algorithms for the maximum stable set problem, International Transactions in Operational Research 19 (1-2) (2012) 161–199.

[31] F. Rossi, S. Smriglio, A branch-and-cut algorithm for the maximum cardinality stable set problem, Operations Research Letters 28 (2) (2001) 63–74.

[32] K. Yamaguchi, S. Masuda, A new exact algorithm for the maximum weight clique problem, in: 23rd International Conference on Circuits/Systems, Computers and Communictions (ITC-CSCC ' 08), 2008, pp. 317–320.

[33] P. R. Östergård, Cliquer homepage, `http://users.tkk.fi/pat/cliquer.html`.

[34] D. Kumlander, Network resources for the maximum clique finding problem, `http://www.kumlander.eu/graph/`.

[35] DIMACS implementation challenges, `http://dimacs.rutgers.edu/Challenges/`.