Algorithms for the maximum weight clique problems

Shimizu, Satoshi

(Degree)
博士（工学）

(Date of Degree)
2018-09-25

(Date of Publication)
2019-09-01

(Resource Type)
doctoral thesis

(Report Number)
甲第7301号

(URL)
https://hdl.handle.net/20.500.14094/D1007301

Doctoral Dissertation

Algorithms for the maximum weight clique problems

(最大重みクリーク問題に対するアルゴリズムに関する研究)

July 2018

Graduate School of Engineering
Kobe University

Satoshi Shimizu

(清水 悟司)

abstract

A set of vertices V' in a graph $G = (V, E)$ is called a clique if any pair of vertices in V' are adjacent. Sometimes a subgraph induced by V' is also called a clique. The decision problem, k-clique problem, is to determine if there is a clique of size k in a graph G. It is one of the 21 NP-complete problems shown by Richard Karp. The theory of NP-completeness is important in computer science, and relates to the unresolved problem *P vs NP Problem* that is one of the 7 millennium prize problems by clay mathematics institute.

The maximum clique problem (MCP), to find a clique of maximum cardinality of a given graph, is an optimization problem derived from k-clique problem. MCP is known to be NP-hard, and no polynomial time algorithm have been found for NP-hard problems. Since it has lots of practical applications in various fields, there are a lot of studies about MCP.

In this paper, we show various kind of algorithms for kinds of maximum weight clique problems. To represent some properties in practical applications, weights are given to vertices or edges. The maximum weight clique problem (MWCP) is to find a clique of maximum weight in a vertex-weighted graph. The maximum edge-weight clique problem (MEWCP) is to find a clique of maximum weight in a edge-weighted graph. These problems are generalization of MCP and are also NP-hard. There are six chapters in this paper. In the first chapter, we show the background that includes introduction of these problems.

In the second chapter, we propose exact algorithms for the MWCP. Proposed algorithms are based on the branch-and-bound. The branching procedure divides a problem into smaller subproblems and solves them in a recursive manner. During this process, an 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 this paper, we propose a new upper bound function. It uses optimal weights of multiple subgraphs. To calculate the value of it in short time, our algorithm prepares tables before the branch-and-bound. In the branch-and-bound, upper bounds are calculated in short time using the tables. By some computer experiments, we confirm proposed algorithms are faster than previous algorithms.

In the third chapter, we show exact algorithms for the MEWCP. For MEWCP, there are only exact methods that formulate MEWCP in mathematical programming. In this paper, we propose a new formulation technique to improve the performance of mathematical programming solvers. For the mixed-integer programming formulation, it renumbers vertex indices to control the range of variables. In addition, we propose a branch-and-bound algorithm for MEWCP. The upper bound calculation of MEWCP is more complicated than MWCP because of edge weights. Proposed algorithm decomposes edge weights in each subproblem into three groups and calculates an upper bound for each of them. It assigns some edge weights to vertices as pseudo vertex weights, and applies the upper bound calculation of MWCP for them. For the rest of edge-weights, it calculates upper bounds using the optimal weight of subproblem already searched. By some computer experiments, we confirm proposed algorithm is faster than previous methods.

In the fourth chapter, we propose greedy algorithms for the minimum weight vertex cover problem. The minimum weight vertex cover problem is equivalent to MWCP in computational complexity. In this paper, two heuristic algorithms of better average performance are proposed. Proposed algorithms are based on a greedy algorithm that constructs a minimal solution in linear time. To obtain better solutions, our algorithms search a lot of solutions and return the best among them. One is based on rotating technique and the other is based on the branching technique. The time complexity of them is linear time per solution. By some computer experiments, we confirm they can find better solutions in shorter time than previous algorithms.

In the fifth chapter, we propose data structures for local search algorithms for MEWCP. Local search is often used for MCP, MWCP and MEWCP. For each solution, local search algorithms define *neighborhood*

that is a set of solutions. They start from a solution and continuously move it to a neighbor to find better solutions. Various techniques are proposed to avoid staying in local optimums. Since neighborhoods are scanned on each movement, the time to calculate neighborhood is important. In this paper, we propose two new data structures to manage neighborhoods. One can be used for graphs represented by adjacency lists. For each vertex, it calculates the number of adjacent vertices in the clique to manage neighborhoods. The other is for graphs represented by adjacency matrix. For each vertex, it calculates the number of non-adjacent vertices in the clique to manage neighborhoods. These data structures combined with some local search algorithms are compared using some benchmarks. We confirm proposed methods are better than previous one.

In the sixth chapter, we conclude our research. In this paper, we propose variety of algorithms for the maximum weight clique problems. Each of them can be used in kinds of practical applications.

Contents

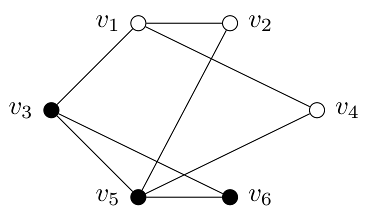
Chapter 1

Introduction

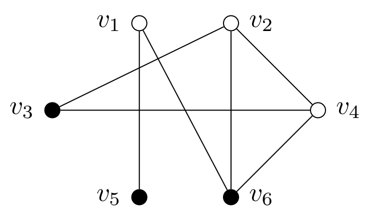
Graphs are widely used to represent structures or relationships such as networks, molecular structures, electric circuits, word cooccurrence relations and so on. In some cases, each vertex or each edge has a non-negative weight that represents importance, frequency, cost, etc. A set of vertices V' in an undirected graph $G = (V, E)$ is called a clique if any pair of vertices in V' are adjacent. Sometimes a subgraph induced by V' is also called a clique. A clique corresponds to a group of strong ties such as communities on social networks.

The decision problem, k-clique problem, is to determine if there is a clique of size k in a graph G. k-clique problem is one of the 21 NP-complete problems shown by Richard Karp [30]. Problems classified as NP-complete are widely believed to be intractable. No polynomial time algorithm for them has not been found. If a polynomial time algorithm for them is given, every problem classified as NP can be solved in polynomial time. This relates to the unresolved problem called *P vs NP Problem*, that is one of the 7 millennium prize problems by Clay Mathematics Institute.

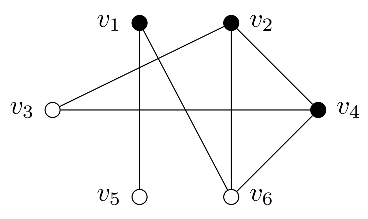
The maximum clique problem (MCP), to find the clique of maximum cardinality of a given graph, is an optimization problem derived from k-clique problem. MCP is known to be NP-hard [23]. There are some *weighted* generalization of MCP. The maximum weight clique problem (MWCP) is to find a clique C such that sum of vertex weights is the maximum in a given vertex-weighted graph. The maximum edge-weight clique problem (MEWCP) is to find a clique C such that sum of edge weights is the maximum in a given edge-weighted graph. In addition, some NP-hard problems are equivalent to MCP. For an undirected graph $G = (V, E)$, a subset V' of V is a vertex cover of G if at least one endpoint of any edge is in V'. A vertex set I in an undirected graph $G = (V, E)$ is an independent set if any pare of vertices in I are not adjacent. For an independent set I, the complement set $V \setminus I$ is a vertex cover. A clique C in $G = (V, E)$ is an independent set in the complement graph of G. Therefore the maximum independent set problem (MISP) and the minimum vertex cover problem (MVCP) for general graphs are equivalent to the MCP. For example, Figure 1.1 shows examples of MCP, MISP, and MVCP. In Figure 1.1a, the maximum clique is $\{v_3, v_5, v_6\}$. The graphs of Figure 1.1b and 1.1c are the complement graph of the graph of Figure 1.1a. In Figure 1.1b, the maximum independent is $\{v_3, v_5, v_6\}$. In Figure 1.1c, the minimum vertex cover is $\{v_1, v_2, v_4\}$. Of course, the maximum weight independent set problem (MWISP) and the minimum weight vertex cover problem (MWVCP) for general graphs are equivalent to the MWCP. These problems have many applications in coding theory [11], network design [57], computer vision [26], bioinformatics [31], auctions [14], protein side-chain packing [3, 13], market basket analysis [16], communication analysis [20, 19], etc.



(a) MCP



(b) MISP



(c) MVCP

Figure 1.1: Problem examples

The purpose of this paper focuses on the algorithms for these maximum weight clique problems. Since these problems are NP-hard, various kinds of algorithms are studied [65].

Exact algorithms : Algorithms that calculate exact solutions in exponential time.

Approximate algorithms : Algorithms that calculate approximate solutions in polynomial time. Approximation ratio is guaranteed.

Heuristics : Algorithms that calculates good solutions. No approximation ratio is guaranteed. Some algorithms such as *greedy algorithms* give solutions in polynomial time. Other algorithms such as *local search* continuously search for better solutions until they are terminated.

In this paper, we propose exact algorithms for MWCP, exact algorithms for MEWCP, greedy algorithms for MWVCP and data structures for local search algorithms for MEWCP.

The branch-and-bound technique is often used in exact algorithms. The branching procedure divides a problem into smaller subproblems and solves them in a recursive manner. During this process, the bounding procedure calculates an upper bound of each subproblem and prunes the subproblem if it is proved that the subproblem does not contain the global optimum solution. In previous studies, several techniques have been investigated to obtain upper bounds for subproblems. For the MCP, vertex coloring is used in numerous algorithms [2, 5, 9, 53, 54, 55, 61, 60, 59, 64]. 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 [33, 34, 35, 36]. A bounding procedure in [44, 43] runs in $O(1)$ time using optimum values of subproblems already searched. In these methods, $|V|$ subproblems are solved sequentially. During the execution, an upper bound of subproblem P is calculated from an exact value of subproblems which are already solved. Some algorithms uses some upper bounds shown above [33, 34]. In [68], longest paths in a directed acyclic graph are calculated in a bounding procedure. An upper bound calculation based on MaxSAT reasoning is proposed in [22]. Other approaches have been proposed by previous studies [6, 4, 15, 45, 46, 47, 51, 52]. 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 two branch-and-bound algorithm for MWCP. One is called *VCTable* that uses vertex coloring to calculate upper bounds. VCTable calculates vertex coloring only once at the beginning. It calculates upper bounds for all of subgraphs and store them in upper bound tables. Using upper bound tables, it can calculate upper bounds of vertex coloring in short time. The other is *OTClique*. Before branch-and-bound, it calculates the weights of optimal solutions for a lot of small subgraphs and stores the values to *optimal tables*. Optimal tables are used to calculate upper bounds in branch-and-bound. We compare algorithms using some kinds of benchmarks. VCTable is faster than previous ones in many cases, and only OTClique can obtain exact solutions for all graphs and it performs much faster than others for nearly all graphs.

For MEWCP, there are only exact methods [24] that formulate MEWCP in mathematical programming. In this paper, we introduce some formulations by modifying formulations of the maximum diversity problem (MDP). Given an edge-weighted complete graph and a natural number b, MDP [39], also known as b-clique problem [56], is to find a clique of size b that has maximum sum of edge weights. For MEWCP, we propose a new formulation technique to improve performance of mathematical programming solvers.

For the mixed-integer programming formulation, it renumbers vertex indices to control the range of variables. In addition, we propose a branch-and-bound algorithm for MEWCP. The upper bound calculation for MEWCP is more complicated than MWCP. In the branch-and-bound, each subproblem of MWCP and MEWCP consists of a candidate vertex set and a clique under construction. In MWCP, an upper bound can be calculated using only the subgraph induced by the candidate vertex set. On the other hand, in MEWCP, both of the candidate vertex set and the clique must be considered in upper bound calculation because of edge weights between them. Proposed algorithm in this paper, called *EWCLIQUE*, decomposes subproblems into three components and calculate an upper bounds for each component. It assigns some edge weights to vertices as pseudo vertex weights, and applies the upper bound calculation of MWCP for them. For the rest of edge-weights, it calculates upper bounds using the optimal weight of subproblem already searched. By some benchmarks, we confirm that EWCLIQUE is faster than methods based on mathematical programming.

Since these problems are known to be NP-hard, non-exact algorithms such as approximation algorithms or heuristics are studied. It is easily shown from [25] that MCP, MWCP, MEWCP and MWISP are hard to approximate within a constant factor. MWVCP is also NP-hard, but some approximation algorithms have been proposed [18, 7, 48, 41, 8, 29]. These approximation algorithms guarantee approximation ratio less than or equal to 2, however, any of them do not guarantee minimality of solutions. To analyse average performance of these approximation algorithms, computational experiments are done in [58]. In addition, a post-processing called DW was proposed in [58]. DW minimalizes vertex covers by removing vertices in nonincreasing order of weight. In this paper, we propose two fast greedy algorithms for MWVCP of better average performance than those approximation algorithms. Proposed algorithms are based on a greedy algorithm which removes vertices from a vertex cover initialized by V. The base greedy algorithm guarantees the minimality of solutions, and moreover, computation time is linear to the size of the given graph. Since the base greedy algorithms is very simple, it is faster than known algorithms, but it finds worse solutions. To obtain better solutions, the strategy of proposed algorithms is to construct a large number of feasible solutions by a greedy algorithm without increasing the computational complexity per solution. One is based on rotating technique and the other is based on the branching technique. We confirm that proposed algorithms find better solutions and the computation time is shorter than the approximation algorithms for MWVCP.

Another approach for these problems is heuristics that continuously search for better solutions until they are terminated. Some of them based on metaheuristics [12, 28]. For MCP, MWCP and MEWCP, local search is often used [49, 66, 63, 21, 50]. For each solution, local search algorithms define *neighborhood* that is a set of solutions. They start from a solution and continuously move the solution to a neighbor to find better solutions. If they always move solutions to the best (maximum weight) neighbors, they can easily trapped into local optimums. To avoid staying in local optimums, each algorithm adopts various techniques. Phased local search (PLS) [49, 50] switches three phases that have different policies to chose neighbors. Multi neighborhood tabu-search (MN/TS) maintains a tabu-list that inhibits MN/TS from moving to recently visited solutions. For local search algorithms, the time to calculate neighborhoods is important because they scan neighborhoods on every movement. For adjacency list representation of a graph G, a data structure is proposed to manage neighborhoods [21]. In this paper, we propose two new data structures to manage neighborhoods. One can be used for graphs represented by adjacency lists. For each vertex, it calculates the number of adjacent vertices in the clique to manage neighborhoods. The other is for graphs represented by adjacency matrix. For each vertex, it calculates the number of non-adjacent vertices in the clique to manage neighborhoods. The time complexity to update neighborhoods of proposed methods is smaller than previous one. In addition, for MEWCP, calculating the weight of a clique takes more time than MWCP because of edge weights. Proposed method reduces the time complexity for clique weight calculation. By some experiments, we confirm that proposed data structures perform better than previous one.

The remainder of this paper is organized as follows. Exact algorithms for MWCP are described in Chapter 2. Exact algorithms for MEWCP are shown in Chapter 3. Greedy algorithms for MWVCP are described in Chapter 4. Data structures for local search algorithms for MEWCP are shown in Chapter 5. We conclude the paper in Chapter 6.

Chapter 2

Exact Algorithms for MWCP

2.1 Introduction

In this chapter, we propose two new branch-and-bound based exact algorithms for MWCP. Branch-and-bound algorithms recursively divide problems into smaller subproblems to find the optimal solution, and unnecessary subproblems are pruned by calculating upper bounds.

First, we propose *VCTable*. It calculates upper bounds using vertex coloring. Before branch-and-bound, it calculates the upper bounds of all subgraphs and stores the values in vertex coloring upper bound tables. In branch-and-bound, using the tables, it calculates upper bounds in short time. By computational experiments, we show that it is faster than other algorithms in many cases.

Second, we propose a new branch-and-bound algorithm *OTClique*. OTClique 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, optimal tables are used to calculate upper bounds. We performed experiments with OTClique and existing algorithms for several types of graphs. The results indicate that only the OTClique can obtain exact solutions for all graphs and that it performs much faster than other algorithms for nearly all graphs.

In the section 2.2, we describe VCTable. Proposed algorithm OTClique is described in the section 2.3. Computer experiments are shown in the section 2.4.

2.2 Proposed algorithm VCTable

Proposed algorithm VCTable is shown in this section.

2.2.1 Notation

For an undirected graph $G = (V, E)$, $w(v)$ denotes the weight of $v \in V$. For a set of vertices $V' \subseteq V$, $w(V')$ denotes $\sum_{v \in V'} w(v)$. 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.

2.2.2 Upper bounds of vertex coloring

VCTable uses vertex coloring to calculate upper bounds. The vertex coloring is a procedure to divide V into mutually disjoint independent sets $\{I_1, I_2, \ldots, I_k\}$, where k is the number of independent sets. Some algorithms for MCP [5, 60] and MWCP [36] use vertex coloring to calculate upper bounds. The following inequality holds because at most one vertex can be included in a clique from each I_i :

$$w_{opt}(V) \leq \sum_{i=1}^{k} \max\{w(v) \mid v \in I_i\}. \tag{2.1}$$

For any $S \in V$, the following inequality is obtained :

$$w_{opt}(S) \leq \sum_{i=1}^{k} \max\{w(v) \mid v \in I_i \cap S\}. \tag{2.2}$$

VCTable calculates upper bounds using this inequality (called coloring upper bound). Vertex coloring is done only once in the beginning of VCTable. In the branch-and-bound, VCTable reuses it for every subproblem. In addition, VCTable limits the maximum size of each I_i to one word length of CPU. This limitation is required for the upper bound tables described in the next.

2.2.3 Upper bound tables

To calculate coloring upper bounds, finding the vertex of maximum weight in each I_i is needed. We propose upper bound tables to reduce the computation time to calculate upper bounds. After the vertex coloring, VCTable represents vertex sets by bit vectors. Since each independent set size is less than or equal to one word length, each I_i can be represented by one word bit vector. For each bit vectors of I_i, VCTable calculates coloring upper bounds for all subgraphs and stores them to a table (an array in C). The index to refer the tables is a bit vector of one word.

For the graph G_{ex} shown in Figure 2.1, let $I_1 = \{v_8, v_7, v_6\}$, $I_2 = \{v_5, v_4, v_3\}$, $I_3 = \{v_2, v_1\}$. Figure 2.2 shows upper bound tables for G_{ex} and these independent sets I_1, I_2, I_3. Using these tables, calculation of the maximum weight of each independent set can be replaced with only one operation to refer the tables. For example shown in Figure 2.2, a vertex subset $\{v_8, v_6, v_5, v_4, v_2\}$ is represented by a set of bit vectors $\{101, 110, 10\}$. The coloring upper bound is calculated as follows :

$$table[1][101] + table[2][110] + table[3][10] = 8 + 8 + 4 = 20$$

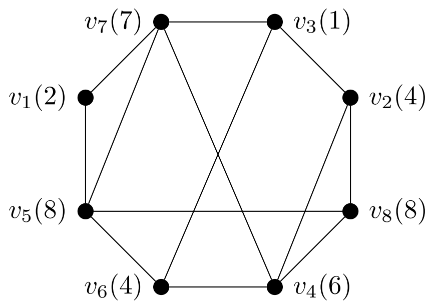


Figure 2.1: a graph example G_{ex}

In addition, when each independent set size is smaller than one word length, more than one independent set can be packed in one bit vector. For example, Figure 2.3 shows the table when I_1 and I_2 are packed into one bit vector. For a vertex subset $\{v_8, v_6, v_5, v_4, v_2\}$, the upper bound can be calculated as follows:

$$table[1,2][101110] + table[3][10] = 16 + 4 = 20$$

This packing technique is efficient in dense graphs, because sizes of independent sets are small.

I_1

$S \subseteq I_1$	Bits	$\max\{w(v) \mid v \in S\}$
$\emptyset$	000	0
$\{v_6\}$	001	4
$\{v_7\}$	010	7
$\{v_7, v_6\}$	011	7
$\{v_8\}$	100	8
$\{v_8, v_6\}$	101	8
$\{v_8, v_7\}$	110	8
$\{v_8, v_7, v_6\}$	111	8

I_2

$S \subseteq I_2$	Bits	$\max\{w(v) \mid v \in S\}$
$\emptyset$	000	0
$\{v_3\}$	001	1
$\{v_4\}$	010	6
$\{v_4, v_3\}$	011	6
$\{v_5\}$	100	8
$\{v_5, v_3\}$	101	8
$\{v_5, v_4\}$	110	8
$\{v_5, v_4, v_3\}$	111	8

I_3

$S \subseteq I_3$	Bits	$\max\{w(v) \mid v \in S\}$
$\emptyset$	00	0
$\{v_1\}$	01	2
$\{v_2\}$	10	4
$\{v_2, v_1\}$	11	4

Figure 2.2: vertex coloring upper bound tables

$I_1 \cup I_2$

$S \subseteq I_1 \cup I_2$	Bits	$\max\{w(v) \mid v \in S \cap I_1\}$ $+ \max\{w(v) \mid v \in S \cap I_2\}$
$\emptyset$	000000	0
$\{v_3\}$	000001	1
$\{v_4\}$	000010	6
$\{v_4, v_3\}$	000011	6
$\{v_5\}$	000100	8
$\{v_5, v_3\}$	000101	8
$\vdots$	$\vdots$	$\vdots$
$\{v_7, v_5, v_4, v_3\}$	010111	15
$\{v_7, v_6\}$	011000	7
$\{v_7, v_6, v_3\}$	011001	8
$\{v_7, v_6, v_4\}$	011010	13
$\{v_7, v_6, v_4, v_3\}$	011011	13
$\{v_7, v_6, v_5\}$	011100	15
$\vdots$	$\vdots$	$\vdots$
$\{v_8, v_7, v_6, v_4\}$	111010	14
$\{v_8, v_7, v_6, v_4, v_3\}$	111011	14
$\{v_8, v_7, v_6, v_5\}$	111100	16
$\{v_8, v_7, v_6, v_5, v_3\}$	111101	16
$\{v_8, v_7, v_6, v_5, v_4\}$	111110	16
$\{v_8, v_7, v_6, v_5, v_4, v_3\}$	111111	16

Figure 2.3: Packed vertex coloring upper bound tables

2.2.4 Upper bounds of $c[\cdot]$

Let $[v_n, v_{n-1}, \ldots, v_1]$ be a vertex sequence, where n is the number of vertices in V. VCTable defines V_i as $\{v_1, v_2, \ldots, v_i\}$ (clearly, $V_n = V$). For $i = 1, 2, \ldots, n$, VCTable calculates the maximum weight clique of $G(V_i)$ by branch-and-bound, and stores $w_{opt}(V_i)$ in $c[i]$ to use it as upper bounds later. For any subset $S \subseteq V$, the following inequality holds because $S \subseteq V_i$:

$$w_{opt}(S) \leq c[\max\{i \mid v_i \in S\}] \tag{2.3}$$

The time complexity of this upper bound calculation is $O(1)$. This is originally proposed by Östergård[43]. VCTable uses both upper bounds of $c[\cdot]$ and vertex coloring upper bounds to prune unnecessary subproblems.

2.2.5 Initial ordering

The independent set assignment $I_1, I_2, \ldots, I_k$ and vertex sequence $[v_n, v_{n-1}, \ldots, v_1]$ affect overall performance of VCTable. The initial ordering of VCTable is based on the following policies :

Policy 1 Assign large index to vertices of large weights.

Policy 2 Put vertices of large weights into one independent set.

Since $c[i] \leq c[i+1]$ holds for any i, Policy 1 is to keep upper bounds of $c[\cdot]$ small. Since the coloring upper bounds is total weight of maximum weighted vertices in each independent set, Policy 2 makes coloring upper bounds small.

VCTable first constructs the independent set I_1 by picking vertices in weight nonincreasing order. If vertices u, v have same weight and $w(N(u)) \leq w(N(v))$, VCTable picks u first. In this process, indices starting from n to 1 are assigned to picked vertices. If I_1 becomes maximal, then VCTable constructs the next independent I_2 in the same way. Until all vertices are picked and assigned indices, VCTable continues this procedure. For G_{ex} shown in Figure 2.1, Figure 2.4 shows the vertex sequence obtained by VCTable.

independent sets	I_1			I_2			I_3	
vertex	f	b	h	e	g	c	d	a
vertex weight	8	7	4	8	6	1	4	2
$w(N(\cdot))$	18	17	15	21	23	15	15	15
assigned vertex number	v_8	v_7	v_6	v_5	v_4	v_3	v_2	v_1
$c[\cdot]$ calculated in branch-and-bound	18	17	12	10	10	5	4	2

Figure 2.4: Initial ordering of VCTable for G_{ex}

2.2.6 Branch-and-bound

The overall algorithm of VCTable is shown in Algorithm 1. VCTable first constructs independent sets and a vertex sequence (line 2). For all subgraphs of each independent set, VCTable calculates coloring upper bounds and stores them in the vertex coloring upper bound tables (line 3). For $i = 1, 2, \ldots, n$, VCTable calculates the maximum weight clique of $G(V_i)$ by branch-and-bound (line 6), and stores $w_{opt}(V_i)$ in $c[i]$ to use it as upper bounds (line 7). The recursive procedure EXPAND$(\cdot, \cdot)$ creates subproblems to search the maximum weight clique, and calculates upper bounds to prune unnecessary subproblems. If S is empty, it is the base case that updates C_{max} (lines 12-17). Otherwise, after the bounding step, it calls itself recursively (lines 25-26). In the bounding step, the value $c[\cdot]$ is used as an upper bound (line 22). In addition, VCTable calculates coloring upper bounds using vertex coloring upper bound tables (line 22).

Algorithm 1 Proposed algorithm VCTable

INPUT: an undirected graph G and vertex weight $w[\cdot]$
OUTPUT: the maximum weight clique
GLOBAL VARIABLES: C_{max}, $c[\cdot]$

```
1: procedure MAIN
2:     Construct independent sets $I_1, I_2, \dots, I_k$ and a vertex sequence $[v_n, v_{n-1}, \dots, v_1]$.
3:     Construct vertex coloring upper bound tables.
4:     $C_{max} \leftarrow \emptyset$
5:     for $i$ from 1 to $n$ do
6:         EXPAND($V_i, \emptyset$)                ▷ Calculate the maximum weight clique of $G(V_i)$
7:         $c[i] \leftarrow w(C_{max})$    ▷ After EXPAND($V_i, \emptyset$), $C_{max}$ is the maximum weight clique of $G(V_i)$
8:     end for
9:     return $C_{max}$
10: end procedure

11: procedure EXPAND($S, C$)
12:     if $|S| = 0$ then
13:         if $w(C) > w(C_{max})$ then
14:             $C_{max} \leftarrow C$
15:         end if
16:         return
17:     end if
18:     $i \leftarrow \max\{j \mid v_j \in S\}$                ▷ $v_j$ is also in $I_h$.
19:     if $w(C) + c[i] \leq w(C_{max})$ then
20:         return
21:     end if
22:     if $w(C) + \sum_{j=1}^{k} \max\{w(v) \mid v \in I_j \cap S\} \leq w(C_{max})$ then    ▷ Using upper bound tables.
23:         return
24:     end if
25:     EXPAND($S \cap N(v_i), C \cup \{v_i\}$)        ▷ Solve subproblems where $C$ includes $v_i$
26:     EXPAND($S \setminus \{v_i\}, C$)        ▷ Solve subproblems where $C$ does not include $v_i$
27: end procedure
```

2.3 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 the upper bound function $UB(\cdot,\cdot)$. We then describe the phases of the proposed algorithm in detail.

2.3.1 Notation

For an undirected graph $G = (V, E)$, $w(v)$ denotes the weight of $v \in V$. For a set of vertices $V' \subseteq V$, $w(V')$ denotes $\sum_{v \in V'} w(v)$. 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 $\bigcup_{i=1}^{k} P_i = V$.

2.3.2 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 :

$$UB(\Pi, V') = \sum_{i=1}^{k} w_{opt}(V' \cap P_i)\ . \tag{2.4}$$

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 $V' \subseteq V$:*

$$w_{opt}(V') \leq UB(\Pi, V')\ . \tag{2.5}$$

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

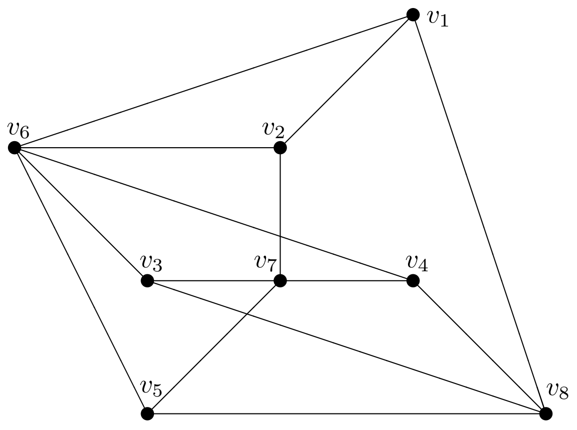
$$\begin{aligned}
w_{opt}(V') &= w(C) && (2.6)\\
&= \sum_{i=1}^{k} w(C \cap P_i) && (2.7)\\
&\leq \sum_{i=1}^{k} w_{opt}(V' \cap P_i) && (2.8)\\
&= UB(\Pi, V')\ . && (2.9)
\end{aligned}$$

□

Example

Let $G = (V, E)$ be a graph shown in Figure 2.5 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 2.5. 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 :

$$\begin{aligned} UB(\Pi, V') &= w_{opt}(V' \cap P_1) + w_{opt}(V' \cap P_2) + w_{opt}(V' \cap P_3) && (2.10)\\ &= w_{opt}(\{v_1, v_2\}) + w_{opt}(\{v_3, v_4\}) + w_{opt}(\{v_6, v_8\}) && (2.11)\\ &= 2 + 3 + 5 = 10\ . && (2.12)\end{aligned}$$



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 2.5: Weighted graph

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 2.5 are shown in Figure 2.6. 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 2.3.2 can be done as following :

$$\begin{aligned} UB(\Pi, V') &= table[1][11] + table[2][011] + table[3][101] && (2.13)\\ &= 2 + 3 + 5 = 10\ . && (2.14)\end{aligned}$$

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 2.6: Optimal tables

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 the VCTable in the section 2.2. 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 $\Pi = (P_1, P_2, \dots, P_k)$ be a partition of V. The following inequality holds for any $V' \subset V$:*

$$UB(\Pi, V') \leq k \cdot w_{opt}(V') \ . \tag{2.15}$$

Proof. The inequality (2.15) is immediately obtained in the following way :

$$\begin{aligned} UB(\Pi, V') &= \sum_{i=1}^{k} w_{opt}(V' \cap P_i) && (2.16) \\ &\leq \sum_{i=1}^{k} w_{opt}(V') && (2.17) \\ &= k \cdot w_{opt}(V') \ . && (2.18) \end{aligned}$$

□

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 4 (shown later) makes k smaller by merging small subsets in Π to obtain tighter upper bounds.

Let us define the following notation :

$$\begin{aligned} \Pi(i) &= (P_1, \dots, P_{i-1}, P_i \cup P_{i+1}, P_{i+2}, \dots, P_k) && (2.19) \\ \triangle(V', \Pi, i) &= UB(\Pi, V') - UB(\Pi(i), V') \ . && (2.20) \end{aligned}$$

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 $\Pi = (P_1, P_2, \dots, P_k)$ of V and any subset V' of V, $\triangle(V', \Pi, i)$ satisfies the following inequality :*

$$\triangle(V', \Pi, i) \leq \min\{w_{opt}(V' \cap P_i), w_{opt}(V' \cap P_{i+1})\} \ . \tag{2.21}$$

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

$$\begin{aligned} \triangle(V', \Pi, i) &= UB(\Pi, V') - UB(\Pi(i), V') \\ &= w_{opt}(V' \cap P_i) + w_{opt}(V' \cap P_{i+1}) - w_{opt}(V' \cap (P_i \cup P_{i+1})) \\ &\leq w_{opt}(V' \cap P_i) + w_{opt}(V' \cap P_{i+1}) \\ &\quad - \max\{w_{opt}(V' \cap P_i), w_{opt}(V' \cap P_{i+1})\} \\ &= \min\{w_{opt}(V' \cap P_i), w_{opt}(V' \cap P_{i+1})\} \ . \end{aligned} \tag{2.22}$$

□

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:

$$\begin{aligned} &2^{|P_i|+|P_{i+1}|} - (2^{|P_i|} + 2^{|P_{i+1}|}) \\ =&2^{|P_i|+|P_{i+1}|}(1 - (2^{-|P_{i+1}|} + 2^{-|P_i|})) \\ \geq&2^{|P_i|+|P_{i+1}|}(1 - (2^{-1} + 2^{-1})) \\ =&0 \ . \end{aligned} \tag{2.23}$$

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 :

$$4 \cdot \left\lceil \frac{|V|}{l} \right\rceil \cdot 2^l \leq 10^9 \ . \tag{2.24}$$

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

2.3.3 Precomputation phase

The precomputation phase consists of several procedures. First, the algorithm divides vertices into independent sets and assigns numbers to these vertices (Algorithm 3). 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 4), where the parameter l is given as an input that satisfies (2.24). Finally, the algorithm generates the optimal tables (Algorithm 5). The entire precomputation phase is shown in Algorithm 2.

Algorithm 2 Precomputation phase

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

Algorithm 3 attempts to generate independent sets as large as possible; however, 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 $v_n, v_{n-1}, \ldots, v_1$ in sequence.

Algorithm 3 Generating independent sets

```
INPUT: An undirected graph $G = (V, E)$, vertex weight $w(\cdot)$ and size parameter $l$
OUTPUT: A vertex sequence $[v_n, v_{n-1}, \ldots, v_1]$ and Independent sets $I_1, I_2, \ldots$
 1: procedure GENERATING_INDEPENDENT_SETS
 2:     $X \leftarrow V$
 3:     $j \leftarrow 0$
 4:     while $X$ is not empty do
 5:         $j \leftarrow j + 1$
 6:         $I_j \leftarrow \emptyset$
 7:         $X' \leftarrow X$
 8:         while $X' \neq \emptyset$ and $|I_j| < l$ do
 9:             $i \leftarrow |X|$
10:             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)
11:             $I_j \leftarrow I_j \cup \{v_i\}$
12:             $X' \leftarrow X' \backslash (\{v_i\} \cup N(v_i))$
13:             $X \leftarrow X \backslash \{v_i\}$
14:         end while
15:     end while
16:     return $[v_n, v_{n-1}, \ldots, v_1]$ and $I_1, I_2, \ldots, I_j$
17: end procedure
```

Algorithm 4 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 Generating a partition

```
INPUT: Independent sets $I_1, I_2, \ldots, I_j$, size parameter $l$
OUTPUT: A partition of $V : \Pi = (P_1, P_2, \ldots, P_k)$
1: procedure GENERATING_PARTITION
2:     $k \leftarrow 1$
3:     $P_1 \leftarrow \emptyset$
4:     for $i$ from $j$ downto 1 do
5:         if $|P_k| + |I_i| > l$ then
6:             $k \leftarrow k + 1$
7:             $P_k \leftarrow I_i$
8:         else
9:             $P_k \leftarrow P_k \cup I_i$
10:        end if
11:    end for
12:    return $(P_1, P_2, \ldots, P_k)$
13: end procedure
```

Algorithm 5 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 following equation :

$$w_{opt}(Y) = \max\{w(v) + w_{opt}(Y \cap N(v)) \ , w_{opt}(Y \setminus \{v\})\} \ . \tag{2.25}$$

  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 5 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:     $opt[\emptyset] \leftarrow 0$
3:     $\mathcal{C} \leftarrow \{\emptyset\}$
4:     $V'' \leftarrow \emptyset$
5:     while $V''$ is not $V'$ do          ▷ At the beginning of each loop, any subset of $V''$ is in $C$.
6:         $u \leftarrow$ an arbitrary vertex in $V' \setminus V''$
7:         $\mathcal{C}' \leftarrow \emptyset$
8:         for $X \in \mathcal{C}$ do          ▷ For any $X \in C$, $opt[X]$ is already calculated.
9:             $Y \leftarrow X \cup \{u\}$
10:            $opt[Y] \leftarrow \max\{w(u) + opt[X \cap N(v_u)] \ , opt[X]\}$
11:            $\mathcal{C}' \leftarrow \mathcal{C}' \cap \{Y\}$
12:        end for
13:        $\mathcal{C} \leftarrow \mathcal{C} \cup \mathcal{C}'$
14:        $V'' \leftarrow V'' \cup \{u\}$
15:    end while
16:    return $opt[\cdot]$ for all subsets of $V'$
17: end procedure
```

2.3.4 Branch-and-bound phase

Hereafter, for the vertex sequence $[v_n, v_{n-1}, \dots, v_1]$ obtained in the precomputation phase(Algorithm 3), V_i denotes $\{v_1, v_2, \dots, v_i\}$ 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 6 presents an outline of the branch-and-bound phase. The variables C_{max} and $c[\cdot]$ are global and can be accessed in the EXPAND procedure. First, EXPAND$(V_1, \emptyset)$ is called and the value $w_{opt}(V_1)$ is stored in $c[1]$. Next, EXPAND$(V_2, \emptyset)$ is called and the value $w_{opt}(V_2)$ is stored in $c[2]$. Similarly, the values are stored in $c[3], c[4], \dots$ at each iteration. When EXPAND$(V_i, \emptyset)$ is called, the values $w_{opt}(V_1)$, $w_{opt}(V_2)$, $\dots$, $w_{opt}(V_{i-1})$ are stored in $c[1]$, $c[2]$, $\dots$, $c[i-1]$, respectively. It is obvious that $w_{opt}(V') \le c[M(V')]$ 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 6 Branch-and-bound phase

```
INPUT: $G = (V, E)$, $w(\cdot)$, $\Pi = (P_1, P_2, \dots, P_k)$ $opt[\cdot]$ and a sequence of vertices $[v_n, v_{n-1}, \dots, v_1]$
OUTPUT: the maximum weight clique $C_{max}$
GLOBAL VARIABLES: $C_{max}$, $c[\cdot]$
1: determine the parameter $\alpha$
2: $C_{max} \leftarrow \emptyset$
3: for $i$ from 1 to $\lfloor \alpha n \rfloor$ do
4:     EXPAND$(V_i, \emptyset)$
5:     $c[i] \leftarrow w(C_{max})$        ▷ After EXPAND$(V_i, \emptyset)$, $C_{max}$ is the maximum weight clique of $G(V_i)$.
6: end for
7: EXPAND$(V, \emptyset)$
```

Note that the upper bound of $c[M(V')]$ has been shown in a previous study [43]. 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 2.1, 2.2 and 2.3. 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 random graphs. 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 2.1: effectiveness of upper bounds $c[\cdot]$ for $|V| = 200$, edge density= 0.9 ($\alpha = 1$)

	used	bounded	subproblems	
$c[1] - c[20]$	826.9	0.0	104.6	(<0.01%)
$c[21] - c[40]$	3042.3	24.3	257.0	(<0.01%)
$c[41] - c[60]$	33572.0	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	(0.10%)
$c[121] - c[140]$	253930088.0	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 2.2: 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%)

Table 2.3: 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	484471.0	(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	1406.0	10555860.2	(28.60%)
$c[901] - c[1000]$	102980.4	27.6	16785686.5	(45.49%)

The recursive procedure EXPAND$(\cdot, \cdot)$ is shown in Algorithm 7. 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. 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 12)
- search the optimum solution in the subgraph $G(V' \setminus \{u\})$. (line 13)

For example, if $G(V)$ in Figure 2.5 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 7 Solving a subproblem

```
INPUT: $V' \in V$, $C$                                  ▷ For any $v \in V'$, $C \subseteq N(v)$
OUTPUT: Update $C_{max}$ if better cliques are found.
GLOBAL VARIABLES: $C_{max}$, $c[\cdot]$
 1: procedure EXPAND($V', C$)
 2:     if $V' = \emptyset$ then                            ▷ Recursive calls finished.
 3:         if $w(C) > w(C_{max})$ then
 4:             $C_{max} \leftarrow C$
 5:         end if
 6:         return
 7:     end if
 8:     if $UB(\Pi, V') + w(C) \leq w(C_{max})$ or $c[M(V')] + w(C) \leq w(C_{max})$ then   ▷ Bounding procedure
    by two upper bounds.
 9:         return
10:     end if
11:     $u \leftarrow v_{M(V')}$
12:     EXPAND($V' \cap N(u), C \cup \{u\}$)              ▷ Solve subproblems where $C$ includes $u$.
13:     EXPAND($V' \setminus \{u\}, C$)                   ▷ Solve subproblems where $C$ does not include $u$.
14: end procedure
```

2.3.5 A case study of OTClique

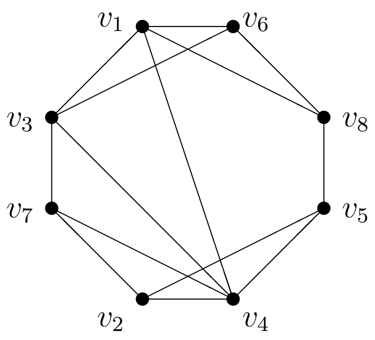
In this section, we show an example for OTClique.

Precomputation phase example

Given an undirected graph shown in of Figure 2.7a, OTClique constructs a vertex sequence and independent sets shown in Figure 2.7b by Algorithm 3. When the input parameter l is given as 3, Algorithm 4 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 2.7b, bit vector representations for some vertex sets are shown in Figure 2.7c.

Optimal Tables is implemented with two-dimensional arrays shown in Figure 2.7d. 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.



(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 2.7: A precomputation example

Upper bound calculation example

For the graph shown in Figure 2.7a, 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 :

$$\begin{aligned} UB(\Pi, S) &= table[3][01] + table[2][110] + table[1][110] \\ &= 5 + 6 + 3 \\ &= 14\ . \end{aligned} \tag{2.26}$$

2.4 Computer experiments

We implement VCTable and OTClique in C. For OTClique, we determined $l = 25$ for graphs with $n \leq 1500$, otherwise $l = 20$. We compared OTClique with Östergård's algorithm [43], Kumlander's algorithm [36] (denoted DK), Yamaguchi/Masuda's algorithm [68] (denoted YM) and IBM's mixed integer programming solver CPLEX. For CPLEX, we formulated MWCP with integer programming as follows :

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

We used the C program *Cliquer* [42] for Östergård's algorithm. For YM, we used a C++ implementation used in [68]. Although Kumlander presents a Visual Basic 6.0 implementation [32], we independently 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 the 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.

2.4.1 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 2.4 and 2.5, respectively. In Table 2.5, 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 2.6. 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 2.7 and 2.8, 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 experiments with other data, we do not show the summary of number of search tree nodes hereafter.

Table 2.4: 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 2.5: 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

Table 2.6: 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 2.7: Number of search tree nodes for random graphs

						CPLEX	
n	d	OTClique	VCTable	YM	DK	iterations	nodes
8000	0.1	1569577.2	1969224.0	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	1962662.0	1093185.9	2281087.4		
2500	0.3	12634956.5	16158044.8	9341501.7	20388682.5		
2000	0.3	4499188.0	5931297.5	3174379.3	7096941.7		
1500	0.4	15814530.7	22526798.8	9558008.3	29162795.8		
1000	0.4	1850807.0	2818775.0	1494500.5	3295741.8		
1000	0.5	28515581.8	42783759.7	15942518.3	57438717.0		
900	0.5	15621489.3	25828420.6	8696737.4	31984389.3		
700	0.6	64268108.0	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	2166003.0	5486785.2	1398292.1	6078605.7	13761832.0	199887.2
300	0.8	115162693.3	369686388.9	39410102.6	470340625.1		
200	0.8	1325098.1	5282665.4	1098077.0	6256181.4	1185288.8	21259.0
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	10439082379.0			116176.7	2642.0
150	0.95	23690554.4	80717090.8	5716143.1	94504883.0	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 2.8: 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

2.4.2 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 [43].

The computation time, its summary and number of search tree nodes are shown in Tables 2.9, 2.10 and 2.11, respectively. OTClique, VCTable, and Cliquer can 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 2.9: 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 2.10: 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

Table 2.11: 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		

2.4.3 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.

The WDP can be formulated by integer programming as follows :

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

CATS, the benchmark set of the WDP, is available online [14]. 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 2.12, 2.13 and 2.14, 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 2.12: 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 2.13: 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 2.14: 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

2.4.4 DIMACS benchmark graphs

The DIMACS benchmarks for the MCP can be obtained online [62]. We used the DIMACS benchmarks to compare weighted algorithms. Note that there are some faster algorithms for the MCP (e.g., [60]) than algorithms for the MWCP.

The computation time, its summary and the number of search tree nodes are shown in Tables 2.15, 2.16 and 2.17, respectively. In the tables, "easy instance" means the instance which at least one of algorithms can solve less than 0.1 second (26 instances) 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 2.16, 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 2.15: 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 2.16: 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 2.17: 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		

2.5 Conclusions

We have proposed two new maximum clique extraction algorithms VCTable and OTClique. VCTable calculates upper bounds of vertex coloring for all subgraphs and stores the values to tables. The tables are used to calculate upper bounds in branch-and-bound. 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.

Chapter 3

Exact Algorithms for MEWCP

3.1 Introduction

In this chapter, we propose exact algorithms for MEWCP. First, we show some mathematical programming formulations for MEWCP, and compare their computational time with a mathematical programming solver. We propose the vertex renumbering technique to reduce computational time. Computational experiments show the efficiency of proposed algorithm.

Second, we propose an exact algorithm based on branch-and-bound. By some computational experiments, we confirm proposal algorithm is faster than the methods based on mathematical programming.

In the section 3.2, we describe mathematical programming formulations for MEWCP and propose a new technique to improve the performance of them. Previous branch-and-bound algorithms for MWCP that are modified to be used in our algorithm is shown in the section 3.3. Some techniques of them are modified and used in the proposed branch-and-bound algorithm for MEWCP. The proposed branch-and-bound algorithm EWCLIQUE is described in the section 3.4. Computer experiments are shown in the section 3.5.

3.2 Mathematical programming formulations for MEWCP

Formulations of MEWCP can be obtained from formulations of the maximum diversity problem. Given an edge-weighted complete graph and a natural number b, the maximum diversity problem (MDP) [39], also known as b-clique problem [56], is to find a clique of size b that has maximum sum of edge weights. Note that MDP is different from MEWCP of the definition in this paper although MDP may sometimes be called MEWCP [1].

The formulations of MDP is shown in [39]. Formulations of MDP has constraints that represents the size of clique is b. Formulations of MEWCP is obtained by replacing them to constraints that represents at most one of two non-adjacent vertices can be included in a clique. This section describes such formulations. Note that some constraints are simplified because the edge-weights are non-negative in MEWCP and are any real number in MDP.

Hereafter, Let $V = \{v_1, v_2, \ldots, v_n\}$ and $\bar{E} = \{(v_i, v_j) \mid i \neq j, (v_i, v_j) \notin E\}$. By the condition $i \neq j$, $\bar{E}$ does not include self loops.

3.2.1 Quadratic programming

MEWCP can be formulated in QP as follows :

$$\begin{aligned} \text{maximize} \quad &: \sum_{(v_i,v_j)\in E} w(v_i, v_j)x_i x_j && (3.1)\\ \text{s.t.} \quad &: x_i + x_j \leq 1, \forall (v_i, v_j) \in \bar{E} && (3.2)\\ & \quad x_i \in \{0, 1\}, \forall v_i \in V. && (3.3) \end{aligned}$$

A binary variable x_i is set to 1 if and only if v_i is in a solution. By constraint (3.2) at most one of any nonadjacent pair of vertices can be in a clique.

3.2.2 Integer programming

MEWCP can be formulated in IP as follows :

$$\begin{aligned}
\text{maximize} \quad &: \sum_{(v_i,v_j)\in E} w(v_i,v_j)y_{ij} && (3.4)\\
\text{s.t.} \quad &: y_{ij} \le x_i, \forall (v_i,v_j) \in E && (3.5)\\
& y_{ij} \le x_j, \forall (v_i,v_j) \in E && (3.6)\\
& x_i + x_j \le 1, \forall (v_i,v_j) \in \bar{E} && (3.7)\\
& x_i \in \{0,1\}, \forall v_i \in V && (3.8)\\
& y_{ij} \in \{0,1\}, \forall (v_i,v_j) \in E. && (3.9)
\end{aligned}$$

A binary variable x_i is set to 1 if and only if v_i is in a solution. By constraints (3.5) and (3.6), for any edge (v_i, v_j), a binary variable y_{ij} is set to 1 only when both v_i and v_j are in the solution.

3.2.3 Mixed integer programming

$N(v)$ denotes the set of all vertices adjacent to v. Let $N^+(v_i) = N(v_i) \cap \{v_j \mid j > i\}$, and $U_i = \sum_{v_j \in N^+(v_i)} w(v_i, v_j)$. MEWCP can be formulated in MIP as follows :

$$\begin{aligned}
\text{maximize} \quad &: \sum_{v_i \in V\setminus\{v_n\}} z_i && (3.10)\\
\text{s.t.} \quad &: x_i + x_j \le 1, \forall (v_i,v_j) \in \bar{E} && (3.11)\\
& z_i \le U_i x_i, \forall v_i \in V \setminus \{v_n\} && (3.12)\\
& z_i \le \sum_{v_j \in N^+(v_i)} w(v_i,v_j)x_j, \forall v_i \in V \setminus \{v_n\} && (3.13)\\
& x_i \in \{0,1\}, \forall v_i \in V. && (3.14)
\end{aligned}$$

A binary variable x_i is set to 1 if and only if v_i is in a solution. If $x_i = 0$, the equation $z_i = 0$ holds by the constraint (3.12). If $x_i = 1$, the value of z_i is determined by the constraints (3.13) because the constraint (3.12) is looser than the constraint (3.13) in this case. By the constraint (3.13), z_i is bounded by the total weight of edges in the clique whose endpoints are v_i and $v_j \in N^+(v_i)$.

3.2.4 Proposed initial renumbering for formulations in MIP

The formulation in MIP has following properties.

- For the optimal solution, the equation $z_i \ge 0$ holds.
- When U_i is small, the range of z_i is small. Especially when $U_i = 0$, the equation $z_i = 0$ holds and the variable z_i is unnecessary.
- For large i, the number of elements in $N^+(v_i)$ is small and U_i becomes small.
- The distribution of U_i depends on the numbering of vertex indices.

From these properties, it is conceivable that computation time of mathematical programming solvers can be shortened by renumbering the vertex indices of the given graph. The proposed method to renumber vertex indices is based on the following policies P1 and P2.

P1 Vertices of large U_i form independent sets.

P2 Make each U_i small.

The proposal method is follow :

1. Construct a maximal independent set I_1. During the construction, the proposal method selects vertices to add to I_1 in nonincreasing order of $\sum_{u \in N(v)} w(v, u)$ and assigns indices v_1,v_2,... in the selection order.

2. Construct a maximal independent set I_2 from $V \setminus I_1$. Same as I_1, vertices are selected to add to I_2 in nonincreasing order of $\sum_{u \in N(v)} w(v,u)$. Assign unused indices to selected vertices in the selection order.

3. Construct maximal independent sets $I_3, I_4, \ldots$ in the same way until I_k where $\cup_{i=1}^{k} I_i = V$.

In constructing the maximal independent I_1, the proposed method selects vertices in nonincreasing order of sum of connected edge weights (Policy P1). Therefore the values of U_i become larger for I_1. Since $\sum_{i=1}^{n} U_i$ equals to the sum of all edge weights, the values of U_i for large indexed vertices become small by giving small indices to vertices of large U_i (Policy P2). For vertices in I_k, $U_i = 0$ holds and the corresponding variables z_i become unnecessary.

3.2.5 Computer experiments

We compare the computation time of formulations by uniform random graphs. The edge weights are uniform random integer variables from 1 to 10. The mathematical programming solver is IBM CPLEX 12.5. The OS is Linux 4.4.0. The CPU is Intel®Core$^{\mathrm{TM}}$i7-6700 CPU 3.40 GHz. RAM is 16GB.

The results are shown in the Table 3.1. In the table, d denotes the edge density. In each condition, we generate 10 instances and calculate the average computation time. Though CPU time for the proposal vertex renumbering is not included in the values in the table, it is shorter than 1msec for every case. From the results, MIP is faster than QP and IP. In addition, we confirmed proposal method makes MIP 14% faster on average.

Table 3.1: CPU time for random graphs [sec]

		MIP	MIP	QP	IP
$\lvert V\rvert$	d	(proposal)			
350	0.1	190.96	214.95	706.68	258.84
300	0.1	91.54	97.52	313.46	124.94
280	0.2	119.47	151.94	417.06	664.09
250	0.2	64.26	82.68	250.55	315.26
250	0.3	97.16	122.93	531.50	over 1000
200	0.3	39.16	46.83	154.74	470.07
200	0.4	57.54	67.78	428.47	over 1000
160	0.4	21.98	26.58	97.61	528.41
170	0.5	52.72	63.37	498.58	over 1000
140	0.5	21.28	22.09	111.73	556.29
130	0.6	28.57	31.70	256.59	634.81
120	0.6	18.10	19.12	129.01	405.38
110	0.7	31.23	36.38	356.28	557.70
100	0.7	15.12	16.33	125.42	262.75
90	0.8	21.51	22.64	266.13	386.01
80	0.8	7.28	8.70	71.41	135.77
80	0.9	14.84	18.10	118.65	181.36
70	0.9	3.80	5.09	16.69	50.80

3.3 Previous branch-and-bound algorithms for MWCP

Before describing the proposed branch-and-bound algorithm for MEWCP, two previous branch-and-bound algorithm for MWCP is shown in this section. The proposed algorithm modifies some techniques of them to use in MEWCP.

For $V' \subseteq V$, let $w_e(V') = \sum_{v \in V'} w(v)$. $P_{MWC}(C, S)$ denotes a subproblem of MWCP, where C is a set of vertices already chosen and S is a set of candidate vertices to be added to C. Note that $C \cap S$ must be an empty set, and for any element $v \in S$, $C \subseteq N(v)$ must be satisfied. The problem corresponding to the given graph $G = (V, E)$ is $P_{MWC}(\emptyset, V)$.

In a branching procedure, a subproblem $P_{MWC}(C, S)$ is divided into $|S|$ subproblems and examined in depth-first recursive manner. For each subproblem, a bounding procedure calculates an upper bound for the weight of feasible solutions in $P_{MWC}(C, S)$. By using the upper bounds, a bounding procedure prunes unnecessary subproblems to reduce the search tree size. The tightness and calculation time for each upper bound are important for the entire computation time of branch-and-bound.

3.3.1 Östergård's algorithm

A branch-and-bound algorithm for MWCP is proposed by Östergård[43]. Let $V_i = \{v_i, v_{i-1}, \ldots, v_1\}$. Östergård's algorithm calculates exact solutions of $P_{MWC}(\emptyset, V_1), P_{MWC}(\emptyset, V_2), \ldots, P_{MWC}(\emptyset, V_n)$ in this order. An exact solution of given instance is finally obtained because $P_{MWC}(\emptyset, V_n)$ is equivalent to the given problem. Each time exact solution of $P_{MWC}(\emptyset, V_i)$ is obtained, its weight is stored in $c[i]$. The array $c[\cdot]$ is used to calculate upper bounds in the following way. Hereafter $M(V')$ denotes the maximum index of vertices in V'. Let F be a feasible solution of $P_{MWC}(C, S)$. Since $S \subseteq V_{M(S)}$, the following inequality holds :

$$\begin{aligned} w(F) &= w(C) + w(S \cap F) && (3.15) \\ &\leq w(C) + w(V_{M(S)} \cap F) && (3.16) \\ &\leq w(C) + c[M(S)]. && (3.17) \end{aligned}$$

3.3.2 Longest path method

An upper bound calculation for MWCP is proposed in [68]. Let $G(V')$ be the subgraph of G induced by $V' \subseteq V$. For a vertex induced subgraph $G(S)$, let $\vec{D}$ be a directed acyclic graph such that the graph obtained by replacing each directed edge in $\vec{D}$ with an undirected edge is isomorphic to $G(S)$. The *length of a path* is defined as the sum of vertex weights in the path. Let $L(\vec{D})$ be the length of a longest path in $\vec{D}$. Let F be a feasible solution of $P_{MWC}(C, S)$. Since $F \cap S$ is a clique in $G(S)$, at least one path in $\vec{D}$ includes all vertices of $F \cap S$ and the following inequality holds :

$$\begin{aligned} w(F) &= w(C) + w(S \cap F) && (3.18) \\ &\leq w(C) + L(\vec{D}). && (3.19) \end{aligned}$$

Hence an upper bound for $w(F)$ can be obtained from a longest path in a directed acyclic graph $\vec{D}$. It is called *longest path method*.

3.4 Proposed branch-and-bound algorithm *EWCLIQUE*

In this section, we propose a new branch-and-bound algorithm EWCLIQUE.

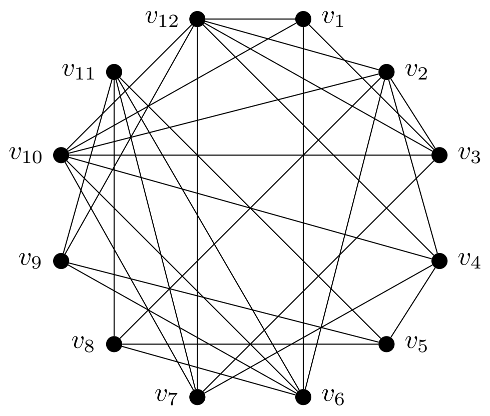
3.4.1 Branch-and-bound for MEWCP

Proposed algorithm EWCLIQUE is the first algorithm for MEWCP based on branch-and-bound. Before introducing EWCLIQUE, we show the representation of subproblems of MEWCP. Hereafter, for a clique $C \subseteq V$, let $w_e(C) = \sum_{u,v \in C} w(u,v)$.

Subproblems of MEWCP

$P_{MEWC}(C,S)$ denotes a subproblem of the MEWCP, where C is a set of vertices already chosen and S is a set of candidate vertices to be added to C. Note that $C \cap S$ must be an empty set, and for any element $v \in S$, $C \subseteq N(v)$ must be satisfied. The problem corresponding to the given graph $G = (V,E)$ is $P_{MEWC}(\emptyset, V)$.

Figure 3.1a shows an example of edge-weighted graph G_{ex}, and Figure 3.1b shows the edge-weight matrix of G_{ex}. In the edge-weight matrix, the blank spaces denote that vertices are not adjacent. Figure 3.2 shows a subproblem $P_{MEWC}(C_{ex}, S_{ex})$ of the graph G_{ex}.



(a) G_{ex}

	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}
v_1	-		3			2				1		4
v_2		-	2	1		1		2		2		2
v_3	3	2	-				4			5		3
v_4		1		-	4		6			3		7
v_5				4	-			4	6		1	
v_6	2	1				-		4	3	5	6	
v_7			4	6			-			3	4	6
v_8		2			4	4		-			5	
v_9					6	3			-		10	5
v_{10}	1	2	5	3		5	3			-		6
v_{11}					1	6	4	5	10		-	
v_{12}	4	2	3	7			6		5	6		-

(b) Edge-weight matrix of G_{ex}

Figure 3.1: a graph example G_{ex}

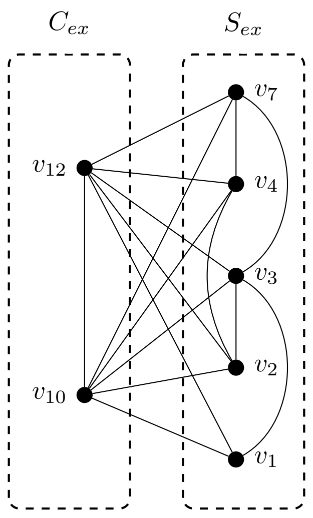


Figure 3.2: A subproblem $P_{MEWC}(C_{ex}, S_{ex})$ of G_{ex}

Three components of a feasible solution

Let F be a feasible solution of $P_{MEWC}(C, S)$. Figure 3.3 shows the relationship between $P_{MEWC}(C, S)$ and F. Since $F = C \cup (S \cap F)$, edges in $G(F)$ can be decomposed into the following three groups :

- Edges between two vertices in C.
- Edges between a vertex in C and a vertex in $S \cap F$.
- Edges between two vertices in $S \cap F$.

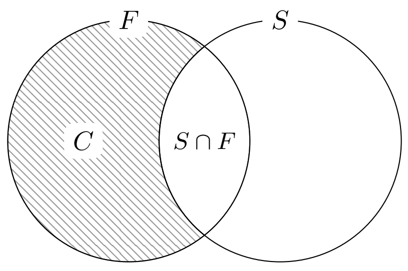


Figure 3.3: F in $P_{MEWC}(C, S)$

According to the decomposition, the following equation is obtained :

$$w_e(F) = w_e(C) + \sum_{u \in C} \sum_{v \in S \cap F} w(u, v) + w_e(S \cap F). \tag{3.20}$$

Figure 3.4 shows the three components of $w_e(F)$. For the second term $\sum_{u \in C} \sum_{v \in S \cap F} w(u, v)$ of the equation (3.20), no corresponding terms are in the equations for MWCP (3.15)-(3.19). To adapt branch-and-bound to MEWCP, efficient upper bound calculation methods for these terms are required.

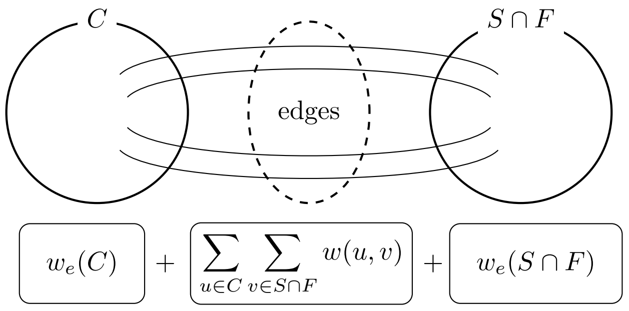

C
$S \cap F$
edges
$w_e(C)$ + $\sum_{u\in C}\sum_{v\in S\cap F} w(u,v)$ + $w_e(S\cap F)$

Figure 3.4: Components of $w_e(F)$ calculation

3.4.2 Outline of EWCLIQUE

The outline of proposed algorithm EWCLIQUE is as follows :

Branching Procedure

Similar to Östergård's algorithm for MWCP, proposed algorithm EWCLIQUE finds optimum solutions of $P_{MEWC}(\emptyset, V_1)$, $P_{MEWC}(\emptyset, V_2)$, …, $P_{MEWC}(\emptyset, V_n)$ in this order, where V_i denotes a vertex set $\{v_i, v_{i-1}, \ldots, v_1\}$. It solves each $P_{MEWC}(\emptyset, V_i)$ by branch-and-bound and stores each optimum value of $P_{MEWC}(\emptyset, V_i)$ in $c[i]$ to use in the bounding procedure.

Bounding Procedure

EWCLIQUE decomposes a subproblem into three components described in 3.4.1, and calculates an upper bound of each component. For the third term $w_e(S \cap F)$ of the equation (3.20), the array $c[\cdot]$ calculated in the branching procedure is used. To handle the second term $\sum_{u\in C}\sum_{v\in S\cap F} w(u,v)$ of the equation (3.20), EWCLIQUE introduces a pseudo vertex weights described in the next.

Pseudo vertex weights

For each vertex $v \in S$ of a subproblem $P_{MEWC}(C, S)$, EWCLIQUE introduces a pseudo vertex weight $w_\rho(C, v) = \sum_{u\in C} w(u,v)$. For example, Figure 3.5 shows the pseudo vertex weight of each $v \in S_{ex}$. Hereafter, $w_\rho(v)$ denotes $w_\rho(C, v)$ when $P_{MEWC}(C, S)$ can be obviously identified.

The pseudo vertex weights satisfy the following equation :

$$\sum_{u\in C}\sum_{v\in S\cap F} w(u,v) = \sum_{v\in S\cap F} w_\rho(v). \tag{3.21}$$

Hence, after assigning $w_\rho(v)$ to each $v \in S$, EWCLIQUE can calculate an upper bound of the second term $\sum_{u\in C}\sum_{v\in S\cap F} w(u,v)$ of the equation (3.20) only with $G(S\cap F)$, without C. Further details of the upper bound calculation is shown in 3.4.5.

$v_i \in S_{ex}$	$w_\rho(v_i)$		
v_1	$w(v_1, v_{10}) + w(v_1, v_{12})$	=	5
v_2	$w(v_2, v_{10}) + w(v_2, v_{12})$	=	4
v_3	$w(v_3, v_{10}) + w(v_3, v_{12})$	=	8
v_4	$w(v_4, v_{10}) + w(v_4, v_{12})$	=	10
v_7	$w(v_7, v_{10}) + w(v_7, v_{12})$	=	9

Figure 3.5: Pseudo vertex weights for $P_{MEWC}(C_{ex}, S_{ex})$

3.4.3 Main routine

We show in Algorithm 8 the main routine of proposed algorithm EWCLIQUE. Before branch-and-bound, EWCLIQUE renumbers vertex indexes for efficiency (described in 3.4.6). The global variable C_{max} is the current-best solution, initialized by an empty set. In the **for** loop, EWCLIQUE finds optimum solutions of $P_{MEWC}(\emptyset, V_1)$, $P_{MEWC}(\emptyset, V_2)$, …, $P_{MEWC}(\emptyset, V_n)$ by the subroutine EXPAND described in 3.4.4. In each loop, the subroutine EXPAND updates C_{max} if better solution is found, and the value of $w_e(C_{max})$ is stored in $c[i]$ for calculation of upper bounds of subproblems.

Algorithm 8 EWCLIQUE

```
INPUT: $G = (V, E)$, $w(\cdot,\cdot)$
OUTPUT: a maximum edge-weight clique $C_{max}$
GLOBAL VARIABLES: $C_{max}$, $c[\cdot]$
1: Renumber vertex indexes. (described in 3.4.6)
2: $C_{max} \leftarrow \emptyset$
3: for $i$ from 1 to $|V|$ do
4:     EXPAND$(\emptyset, V_i)$          ▷ $C_{max}$ is updated if better solution is found(described in 3.4.4)
5:     $c[i] \leftarrow w_e(C_{max})$
6: end for
7: return $C_{max}$
```

Algorithm 9 Solving a subproblem

```
INPUT: a subproblem $P_{MEWC}(C, S)$
OUTPUT: Update $C_{max}$ to a better clique if it exists.
GLOBAL VARIABLES: $C_{max}$, $c[\cdot]$
1: procedure EXPAND$(C, S)$
2:     if $S = \emptyset$ then
3:         if $w_e(C) > w_e(C_{max})$ then
4:             $C_{max} \leftarrow C$
5:         end if
6:         return
7:     end if
8:     $lp[\cdot] \leftarrow$ LONGESTPATH$(C, S)$          ▷ described in 3.4.5
9:     while $S \neq \emptyset$ do
10:        if $w_e(C) + c[M(S)] + lp[M(S)] > w_e(C_{max})$ then
11:            $C' \leftarrow C \cup \{v_{M(S)}\}$
12:            $S' \leftarrow S \cap N(v_{M(S)})$
13:            EXPAND$(C', S')$          ▷ $P_{MEWC}(C', S')$
14:        end if
15:        $S \leftarrow S \setminus \{v_{M(S)}\}$
16:    end while
17: end procedure
```

3.4.4 Subroutine EXPAND

Here we describe the detail of the subroutine EXPAND (in Algorithm 9), which calls itself recursively to find better solution than current C_{max}. In the case of $S = \emptyset$, EXPAND updates C_{max} if $w_e(C) > w_e(C_{max})$ (lines 3-5), and it is the base case of recursive calls (line 6). If $S \neq \emptyset$, in the while loop from line 9, EXPAND divides $P_{MEWC}(C, S)$ into $|S|$ subproblems and examines them recursively.

For each loop from line 9, a subproblem $P_{MEWC}(C', S')$ is created. C' is a clique obtained by adding $v_{M(S)}$ to C (line 11). S' is a candidate vertex set obtained by removing non-adjacent vertices of $v_{M(S)}$ from S (line 12). At the end of the while loop, $v_{M(S)}$ is removed from S and continue the loop unless $S = \emptyset$.

During this process, EWCLIQUE calculates upper bounds of feasible solutions and prunes unnecessary subproblems in line 10. The subroutine "LONGESTPATH" in line 8 constructs a directed acyclic graph from $P_{MEWC}(C, S)$, calculates longest paths of the DAG using pseudo vertex weights, and store their length in the array $lp[\cdot]$ of size $M(S)$. The array $lp[\cdot]$ is used for upper bound calculation.

3.4.5 Upper bound calculation

Let F' be a feasible solution of $P_{MEWC}(C', S')$ in line 13 of Algorithm 9. Here we show that the value of $w_e(C) + c[M(S)] + lp[M(S)]$ in line 10 is an upper bound of $w_e(F')$. From the lines 11 and 12 of Algorithm 9, $C' = C \cup \{v_{M(S)}\}$ and $S' = S \cap N(v_{M(S)})$. Figure 3.6 schematically illustrates inclusion relation of $P_{MEWC}(C, S)$, $P_{MEWC}(C', S')$ and F' by intervals.

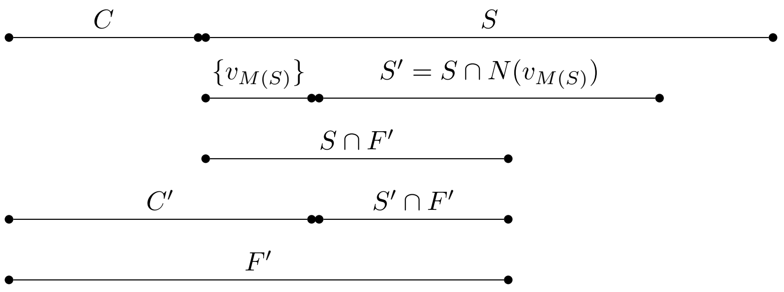


Figure 3.6: Inclusion relation of $P_{MEWC}(C, S)$, $P_{MEWC}(C', S')$ and F'

Since $F' = C \cup (S \cap F')$, the following equation holds similarly to (3.20) :

$$w_e(F') = w_e(C) + \sum_{u \in C} \sum_{v \in S \cap F'} w(u, v) + w_e(S \cap F'). \tag{3.22}$$

EWCLIQUE calculates an upper bound of the equation (3.22). The accurate value of $w_e(C)$ is already calculated during the branching procedure. Therefore proposed algorithm calculates the following upper bounds :

$UB1$**:** an upper bound for $w_e(S \cap F')$

$UB2$**:** an upper bound for $\sum_{u \in C} \sum_{v \in S \cap F'} w(u, v)$

$w_e(C) + UB1 + UB2$ is an upper bound for $w_e(F')$ and used to prune unnecessary subproblems. The calculation of $UB1$ and $UB2$ is described in the next.

UB1: upper bound of $w_e(S \cap F')$

Since $S \subseteq V_{M(S)}$, following inequality holds :

$$w_e(S \cap F') \leq w_e(V_{M(S)} \cap F') \leq c[M(S)]. \tag{3.23}$$

Therefore $c[M(S)]$ can be used as $UB1$. For the tightness of upper bounds, in the subroutine EXPAND, the largest indexed vertex in C is chosen as a branching variable, so that vertices of large indexes disappear from S in early stages (line 13, Algorithm 7).

UB2: upper bound of $\sum_{u \in C} \sum_{v \in S \cap F'} w(u, v)$

For a subproblem $P_{MEWC}(C, S)$, EWCLIQUE constructs a vertex-weighted directed acyclic graph, and calculates the length of longest paths that can be used as $UB2$. $(\overrightarrow{v, u})$ denotes a directed edge from v to u when it is necessary to distinguish from an undirected edge, and the vertex-weighted directed acyclic graph consists of follows :

- A vertex set S.
- A directed edge set $\{(\overrightarrow{v_i, v_j}) \mid (v_i, v_j) \in E(S), i < j\}$, where $E(S)$ is the edge set of $G(S)$.

- A pseudo vertex weight $w_\rho(\cdot)$ calculated from $P_{MEWC}(C,S)$ is assigned as a vertex weight to each vertex in S.

$\vec{D}(C,S)$ denotes this vertex-weighted directed acyclic graph for $P_{MEWC}(C,S)$. Hereafter $\vec{D}$ denotes $\vec{D}(C,S)$, when $P_{MEWC}(C,S)$ can be obviously identified. Figure 3.7 shows $\vec{D}_{ex}$ of $P_{MEWC}(C_{ex},S_{ex})$. Numbers in parentheses represent vertex weights.

Let $[v_1,\dots,v_k]$ a path from v_1 to v_k. We define path length in $\vec{D}$ as the sum of vertex weights in the path. Let $\Pi(\vec{D},v)$ be a set of paths in $\vec{D}$ whose endpoint is v. Let $L(\vec{D},v)$ be length of longest paths in $\Pi(\vec{D},v)$. For example, $\Pi(\vec{D}_{ex},v_3)=\{[v_1,v_3],[v_2,v_3]\}$. The longest path in $\Pi(\vec{D}_{ex},v_3)$ is $[v_1,v_3]$ and its length is $L(\vec{D}_{ex},v_3)=13$, respectively. Note that any path in $\Pi(\vec{D},v_i)$ does not include vertices in $V\setminus V_i$ because of the edge direction of $\vec{D}$. Since F' is a feasible solution of $P_{MEWC}(C',S')$ where $C'=C\cup\{v_{M(S)}\}$ and $S'=S\cap N(v_{M(S)})$, F' is a clique including $v_{M(S)}$. Hence there exists at least one path in $\Pi(\vec{D},v_{M(S)})$ that includes all vertices of $S\cap F'$. Therefore following inequality holds :

$$\sum_{u\in C}\sum_{v\in S\cap F'} w(u,v)=\sum_{v\in S\cap F'} w_\rho(v)\le L(\vec{D},v_{M(S)}). \tag{3.24}$$

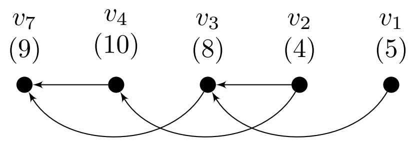


Figure 3.7: $\vec{D}_{ex}$

Algorithm 10 Calculate length of longest paths

INPUT: a subproblem $P_{MEWC}(C,S)$
OUTPUT: an array $lp[\cdot]$ that contains length of longest paths

```
1: procedure LONGESTPATH($C,S$)
2:     $S' \leftarrow S$
3:     while $S' \neq \emptyset$ do
4:         $i \leftarrow \min\{j \mid v_j \in S'\}$
5:         if $S\cap V_i\cap N(v_i)=\emptyset$ then
6:             $lp[i] \leftarrow w_\rho(v_i)$
7:         else
8:             $lp[i] \leftarrow w_\rho(v_i)+\max\{lp[u] \mid u\in S\cap V_i\cap N(v_i)\}$
9:         end if
10:        $S' \leftarrow S'\setminus\{v_i\}$
11:    end while
12:    return $lp[\cdot]$
13: end procedure
```

Proposed algorithm uses $L(\vec{D},v_{M(S)})$ as $UB2$. At line 8 of Algorithm 7, the subroutine LONGEST-PATH shown in Algorithm 10 is called to calculate $UB2$. For each $v_i\in S$, it calculates $L(\vec{D},v_i)$ and stores the value with $lp[i]$. The calculation of $lp[i]$ is done in nonincreasing order of i. There are two cases of calculating $lp[i]$. One is the case that no neighbor of v_i is in $S\cap V_i$ (line 6). In this case, $\Pi(\vec{D},v_i)$ is $\{[v_i]\}$ because of the edge direction of $\vec{D}$. Hence $lp[i]$ equals to $w_\rho(v_i)$. In the other case that some neighbors of v_i are in $S\cap V_i$ (line 8), $lp[i]$ equals to the sum of $w_\rho(v_i)$ and the length of longest paths connected to v_i.

A case study

Here is an example of upper bound calculation for $P_{MEWC}(C_{ex}, S_{ex})$ of Figure 3.2. Figure 3.8 shows $c[\cdot]$ that are already calculated. First proposed algorithm calculates $lp[i]$ for each $v_i \in S_{ex}$ using $w_\rho(v_i)$ and $\vec{D}_{ex}$ in Figure 3.5 and 3.7 respectively. For v_1, $lp[v_1] = w_\rho(v_1)$ because $S_{ex} \cap V_1 \cap N(v_1) = \emptyset$. Same as $lp[v_1]$, $lp[v_2] = w_\rho(v_2)$. For v_3, $S_{ex} \cap V_3 \cap N(v_3) = \{v_1, v_2\} \neq \emptyset$. Proposed algorithm selects v_1 from $\{v_1, v_2\}$ that constructs the longest path. Therefore $lp[v_3] = w_\rho(v_3) + max(lp[v_1], lp[v_2]) = w_\rho(v_3) + lp[v_1] = 13$. Similarly, for v_4 and v_7, length of paths are calculated. The result is shown in Figure 3.9.

Let F'_{ex} be a feasible solution of the subproblem $P_{MEWC}(C'_{ex}, S'_{ex})$ such that $S'_{ex} = S_{ex} \cap N(v_7) = \{v_3, v_4\}$ and $C'_{ex} = C_{ex} \cup \{v_7\} = \{v_{12}, v_{10}, v_7\}$. Since $S_{ex} \subseteq V_{M(S_{ex})} = V_7$, $UB1$ can be calculated as following :

$$\begin{aligned} w_e(S_{ex} \cap F'_{ex}) &\leq w_e(V_7 \cap F'_{ex}) & (3.25)\\ &\leq c[7] & (3.26)\\ &= 6. & (3.27)\end{aligned}$$

Since $M(S_{ex}) = 7$, $UB2$ is calculated as follows :

$$\begin{aligned} \sum_{u \in C_{ex}} \sum_{v \in S_{ex} \cap F'_{ex}} w(u, v) &\leq L_\rho(\vec{D}_{ex}, v_7) & (3.28)\\ &= lp[v_7] & (3.29)\\ &= 23. & (3.30)\end{aligned}$$

In summary, an upper bound for $w_e(F'_{ex})$ is calculated as follows :

$$\begin{aligned} w_e(F'_{ex}) &= w_e(C_{ex}) + \sum_{u \in C_{ex}} \sum_{v \in S_{ex} \cap F'_{ex}} w(u, v) + w_e(S_{ex} \cap F'_{ex}) & (3.31)\\ &\leq w_e(C_{ex}) + lp[7] + c[7] & (3.32)\\ &= 6 + 23 + 6 & (3.33)\\ &= 35. & (3.34)\end{aligned}$$

The maximum edge-weight clique of $P_{MEWC}(C'_{ex}, S'_{ex})$ is $\{v_{12}, v_{10}, v_7, v_4\}$, and its weight is 31, smaller than 35 of the inequality (3.34).

subproblem	optimal weight
$P_{MEWC}(\emptyset, V_1)$	$c[1] = 0$
$P_{MEWC}(\emptyset, V_2)$	$c[2] = 0$
$P_{MEWC}(\emptyset, V_3)$	$c[3] = w_e(\{v_1, v_3\}) = 3$
$P_{MEWC}(\emptyset, V_4)$	$c[4] = w_e(\{v_1, v_3\}) = 3$
$P_{MEWC}(\emptyset, V_5)$	$c[5] = w_e(\{v_4, v_5\}) = 4$
$P_{MEWC}(\emptyset, V_6)$	$c[6] = w_e(\{v_4, v_5\}) = 4$
$P_{MEWC}(\emptyset, V_7)$	$c[7] = w_e(\{v_4, v_7\}) = 6$
$P_{MEWC}(\emptyset, V_8)$	$c[8] = w_e(\{v_2, v_6, v_8\}) = 7$
$P_{MEWC}(\emptyset, V_9)$	$c[9] = w_e(\{v_2, v_6, v_8\}) = 7$
$P_{MEWC}(\emptyset, V_{10})$	$c[10] = w_e(\{v_4, v_7, v_{10}\}) = 12$
$P_{MEWC}(\emptyset, V_{11})$	$c[11] = w_e(\{v_6, v_9, v_{11}\}) = 19$
⋮	⋮

Figure 3.8: Optimal weights stored in $c[\cdot]$

v_i	$S_{ex} \cap V_i \cap N(v_i)$	$lp[i]$		
v_1	$\emptyset$	$w_\rho(v_1)$	$=$	5
v_2	$\emptyset$	$w_\rho(v_2)$	$=$	4
v_3	$\{v_1, v_2\}$	$w_\rho(v_3) + \max(lp[v_1], lp[v_2])$	$=$	13
v_4	$\{v_2\}$	$w_\rho(v_4) + \max(lp[v_2])$	$=$	14
v_7	$\{v_3, v_4\}$	$w_\rho(v_7) + \max(lp[v_3], lp[v_4])$	$=$	23

Figure 3.9: Longest path calculation

3.4.6 Vertex renumbering

For the tightness of upper bounds, proposed algorithm renumbers vertex indexes before branch-and-bound. Let $\sigma(v) = \sum_{u \in N(v)} w(v, u)$. Proposed algorithm adopts following two policies :

Policy 1
Vertices of large $\sigma(\cdot)$ are given large indexes.

Policy 2
Vertices in an independent set are given consecutive indexes.

Similar policies are adopted by algorithms for MWCP shown in the Chapter 2. Since $c[i] \leq c[i+1]$ for any i, policy 1 is adopted to make $c[\cdot]$ small. Policy 2 also makes $c[\cdot]$ small. After renumbering, if $\{v_i, v_{i-1}, \ldots, v_1\}$ is an independent set, $c[i] = c[i-1] = \ldots = c[1] = 0$.

According to these policies, proposed algorithm renumbers vertices by greedy vertex coloring as following.

(1) Maximalize an independent set I_1 by adding vertices in V in nonincreasing order of $\sigma(\cdot)$. Give indexes $v_n, v_{n-1} \ldots$ to vertices in order of addition.

(2) Maximalize an independent set I_2 by adding vertices in $V \setminus I_1$ in nonincreasing order of $\sigma(\cdot)$. Give unused largest indexes to vertices in order of addition.

(3) Repeatedly maximalize independent sets $I_3, I_4, \ldots, I_k$ and give unused indexes to vertices until $\cup_{i=1}^{k} I_i = V$.

3.5 Computer experiments

We implemented proposed algorithm EWCLIQUE by C++ and compare with previous methods in formulations of QP, IP and MIP solved by IBM mathematical programming solver CPLEX 12.5. For the MIP formulation, the vertex renumbering technique described in 3.2.4 is applied. The compiler is g++ 5.4.0 with optimization option -O2. The OS is Linux 4.4.0. The CPU is Intel®Core$^{\text{TM}}$i7-6700 CPU 3.40 GHz. RAM is 16GB. Note that CPLEX is a multi-thread solver based on branch-and-cut, and proposed algorithm is a single-thread solver based on branch-and-bound. In the tables, d denotes the edge density that is $\frac{2|E|}{|V|(|V|-1)}$.

3.5.1 Random graphs

We generated uniform random graphs as benchmarks. Edge-weights are integer values from 1 to 10. In each condition, we generated 10 instances and calculated the average computation time and average search tree size. The results for random graphs are shown in Table 3.2.

In all cases, proposed algorithm can solve them faster than CPLEX. For the instances that CPLEX needs several hundred seconds, proposed algorithm can solve them in less than a second. CPLEX cannot solve instances whose $|V|$ is greater than several hundred. Proposed algorithm solved $|V| = 15000$ for sparse graphs. From the results, we confirmed that proposed algorithm is better than formulations solved by CPLEX.

Table 3.2: Experimental results for random graphs

		optimal	Computation time [sec]				Number of Search tree nodes			
$\|V\|$	d	weight	EWCLIQUE	MIP	IP	QP	EWCLIQUE	MIP	IP	QP
300	0.1	60.7	less than 0.01	91.54	124.94	313.46	1835.1	164460.4	77165.7	193796.7
350	0.1	64.8	less than 0.01	190.96	258.84	706.68	2721.0	264576.7	102579.4	308754.9
15000	0.1	174.7	460.90	over 1000	over 1000	over 1000	702255007.1	-	-	-
250	0.2	97.3	less than 0.01	64.26	315.26	250.55	6412.8	365899.4	704421.5	438526.7
280	0.2	102.4	less than 0.01	119.47	664.09	417.06	9862.9	502859.6	1243089.2	636879.1
5500	0.2	254.8	440.29	over 1000	over 1000	out of memory	1537843711.0	-	-	-
200	0.3	150.0	less than 0.01	39.16	470.07	154.74	14169.8	438264.9	1375256.2	923563.2
250	0.3	155.5	0.01	97.16	over 1000	531.50	34844.2	823443.6	-	2301720.2
2500	0.3	332.8	459.05	over 1000	over 1000	out of memory	1824084938.8	-	-	-
160	0.4	185.5	0.01	21.98	528.41	97.61	31813.3	379811.3	2058272.9	1563023.3
200	0.4	224.0	0.02	57.54	over 1000	428.47	70701.9	869013.2	-	4983953.9
1400	0.4	444.3	758.22	over 1000	over 1000	over 1000	2993314273.1	-	-	-
140	0.5	272.7	0.02	21.28	556.29	111.73	88105.7	507343.1	2933500.6	3028762.4
170	0.5	300.6	0.06	52.72	over 1000	498.58	224608.1	1356597.9	-	9767768.2
750	0.5	560.3	603.91	over 1000	over 1000	over 1000	2342210511.1	-	-	-
120	0.6	399.0	0.05	18.10	405.38	129.01	208388.4	771892.2	3274026.2	4976548.2
130	0.6	424.6	0.07	28.57	634.81	256.59	288494.1	1183461.1	4496672.8	8570931.1
450	0.6	754.2	716.48	over 1000	over 1000	over 1000	2725875895.6	-	-	-
100	0.7	583.5	0.11	15.12	262.75	125.42	437892.1	831713.2	3213925.0	6703943.1
110	0.7	607.1	0.24	31.23	557.70	356.28	950029.7	1657212	5078319.2	15772011.0
270	0.7	1049.7	589.15	over 1000	over 1000	over 1000	2141882035.0	-	-	-
80	0.8	879.0	0.16	7.28	135.77	71.41	617626.3	517672.5	2107836.8	5237500.2
90	0.8	978.0	0.44	21.51	386.01	266.13	1578193.4	1294209.2	5293378.0	16189357.9
170	0.8	1580.2	485.50	over 1000	over 1000	over 1000	1510657832.0	-	-	-
70	0.9	1708.4	0.62	3.80	50.80	16.69	2355972.7	258544.6	993629.0	1335799.1
80	0.9	2059.2	2.93	14.84	181.36	118.65	9974393.5	902841.7	2941909.6	7694204.9
110	0.9	2666.4	590.83	over 1000	over 1000	over 1000	1951189872.0	-	-	-

3.5.2 Graphs from Reuters terror news networks

We compare algorithms with graphs from real-world applications. The Reuters terror news networks (RTN) is produced by Steve Corman and Kevin Dooley at Arizona State University. They are published in [19] and described as follows :

> The Reuters terror news network is based on all stories released during 66 consecutive days by the news agency Reuters concerning the September 11 attack on the U.S., beginning at 9:00 AM EST 9/11/01. The vertices of a network are words (terms); there is an edge between two words iff they appear in the same text unit (sentence). The weight of an edge is its frequency. The network has n = 13332 vertices (different words in the news) and m = 243447 edges, 50859 with value larger than 1. There are no self loops in the network.

Subgraphs of the RTN graph are used as benchmarks for MEWCP in [24], so we generate the same subgraphs as in [40, 24].

The results for RTN are shown in Table 3.3, where each value is obtained by a single measurement. Excepting d1-RTN, CPLEX cannot solve in 1000 seconds. Proposed algorithm solves all RTN graphs in less than 1 second. This means proposed algorithm can handle large instances that previous methods cannot handle.

Table 3.3: Experimental results for RTN graphs

			optimal	Computation time [sec]				Number of Search tree nodes			
graph	$\|V\|$	d	weight	EWCLIQUE	MIP	IP	QP	EWCLIQUE	MIP	IP	QP
d1-RTN	2420	0.0032	4524	0.01	over 1000	681.83	over 1000	2574	-	12634	-
d3-RTN	4755	0.0024	5859	0.04	over 1000	over 1000	out of memory	5571	-	-	-
d6-RTN	6210	0.0021	7085	0.07	over 1000	over 1000	out of memory	7562	-	-	-
d9-RTN	6741	0.0021	7424	0.09	over 1000	over 1000	out of memory	8051	-	-	-
d12-RTN	7448	0.0020	7424	0.11	over 1000	over 1000	out of memory	10022	-	-	-
d15-RTN	7967	0.0020	7424	0.12	over 1000	over 1000	out of memory	11076	-	-	-
d18-RTN	8623	0.0019	7424	0.15	over 1000	over 1000	out of memory	14196	-	-	-
d21-RTN	8945	0.0019	7424	0.16	over 1000	over 1000	out of memory	15880	-	-	-
d24-RTN	9305	0.0018	7424	0.18	over 1000	over 1000	out of memory	19369	-	-	-
d27-RTN	9737	0.0018	7665	0.20	over 1000	over 1000	out of memory	22707	-	-	-
d30-RTN	10101	0.0018	8147	0.22	over 1000	over 1000	over 1000	22742	-	-	-
d33-RTN	10535	0.0018	8485	0.25	over 1000	over 1000	over 1000	27160	-	-	-
d36-RTN	10887	0.0018	8485	0.27	over 1000	over 1000	over 1000	31256	-	-	-
d39-RTN	11232	0.0017	8937	0.31	over 1000	over 1000	over 1000	46294	-	-	-
d42-RTN	11434	0.0017	8937	0.34	over 1000	over 1000	over 1000	58742	-	-	-
d45-RTN	11714	0.0017	9068	0.36	over 1000	over 1000	over 1000	63828	-	-	-
d48-RTN	11961	0.0017	9068	0.40	over 1000	over 1000	over 1000	73645	-	-	-
d51-RTN	12244	0.0017	9068	0.44	over 1000	over 1000	over 1000	104122	-	-	-
d54-RTN	12561	0.0017	10147	0.41	over 1000	over 1000	over 1000	67218	-	-	-
d57-RTN	12799	0.0017	10147	0.43	over 1000	over 1000	over 1000	73036	-	-	-
d60-RTN	13032	0.0017	10147	0.45	over 1000	over 1000	over 1000	76155	-	-	-
d63-RTN	13197	0.0017	10913	0.45	over 1000	over 1000	over 1000	76703	-	-	-
d66-RTN	13308	0.0017	10913	0.51	over 1000	over 1000	over 1000	161644	-	-	-

3.5.3 DIMACS benchmark graphs

DIMACS is a benchmark set often used for MCP [62]. Since DIMACS graphs are not edge-weighted, we give edge-weights $w(v_i, v_j) = (i + j) \bmod 200 + 1$, like the experiments in [50, 24].

The results for DIMACS are shown in Table 3.4, where each value is obtained by a single measurement. Instances that all algorithms cannot solve in 1000 seconds are not shown in the table. Excepting three instances : c-fat200-5, c-fat-500-10 and san200_0.7_1, proposed algorithm is fastest.

Table 3.4: Experimental results for edge-weighted DIMACS

			optimal	Computation time [sec]				Number of Search tree nodes			
graph	$\lvert V\rvert$	d	weight	EWCLIQUE	MIP	IP	QP	EWCLIQUE	MIP	IP	QP
brock200_1	200	0.7454	21230	338.31	over 1000	over 1000	over 1000	1328614116	-	-	-
brock200_2	200	0.4963	6542	0.10	109.66	over 1000	over 1000	345371	3241394	-	-
brock200_3	200	0.6054	10303	1.27	743.58	over 1000	over 1000	4282305	23009456	-	-
brock200_4	200	0.6577	13967	4.84	over 1000	over 1000	over 1000	13814425	-	-	-
c-fat200-1	200	0.0771	7734	less than 0.01	4.80	4.92	62.62	632	1565	11981	2951
c-fat200-2	200	0.1626	26389	less than 0.01	4.72	10.62	68.58	6780	3819	24452	4263
c-fat200-5	200	0.4258	168200	74.31	7.06	15.34	85.97	138193445	5010	22597	2421
c-fat500-1	500	0.0357	10738	less than 0.01	171.91	51.33	749.86	1605	25847	82386	9931
c-fat500-2	500	0.0733	38350	less than 0.01	399.90	144.32	992.06	4679	95599	158428	17113
c-fat500-5	500	0.1859	205864	0.43	264.44	581.82	over 1000	1227023	17959	-	24738
c-fat500-10	500	0.3738	804000	over 1000	745.93	over 1000	over 1000	-	486168	-	-
DSJC500_5	500	0.5019	9626	44.43	over 1000	over 1000	over 1000	200152687	-	-	-
hamming6-2	64	0.9048	32736	less than 0.01	0.07	3.70	0.23	896	4545	19827	7158
hamming6-4	64	0.3492	396	less than 0.01	0.22	2.68	0.66	340	11550	18194	7331
hamming8-2	256	0.9686	800624	0.23	7.80	over 1000	over 1000	65731	104290	-	-
hamming8-4	256	0.6392	12360	1.46	276.15	over 1000	over 1000	2475100	9707184	-	-
johnson8-2-4	28	0.5556	192	less than 0.01	0.03	0.12	0.03	150	185	1922	827
johnson8-4-4	70	0.7681	6552	less than 0.01	0.40	8.74	2.93	3953	39487	318244	123422
johnson16-2-4	120	0.7647	3808	0.25	57.4	over 1000	over 1000	1905154	4543549	-	-
keller4	171	0.6491	6745	0.70	167.84	over 1000	over 1000	2158496	8809323	-	-
MANN_a9	45	0.9273	5460	0.02	1.22	21.77	22.91	116041	78011	1750447	673740
p_hat300-1	300	0.2438	3321	0.01	146.10	over 1000	over 1000	50151	1255846	-	-
p_hat300-2	300	0.4889	31564	42.90	over 1000	over 1000	over 1000	134486327	-	-	-
p_hat500-1	500	0.2531	4764	0.13	over 1000	over 1000	over 1000	468371	-	-	-
p_hat700-1	700	0.2493	5185	0.52	over 1000	over 1000	over 1000	1678557	-	-	-
p_hat1000-1	1000	0.2448	5436	2.92	over 1000	over 1000	over 1000	9890185	-	-	-
p_hat1500-1	1500	0.2534	7135	32.73	over 1000	over 1000	over 1000	106284583	-	-	-
san200_0.7_1	200	0.7000	45295	54.88	28.72	over 1000	over 1000	387149894	662303	-	-
san200_0.7_2	200	0.7000	15073	17.86	over 1000	over 1000	over 1000	48732878	-	-	-
san200_0.9_1	200	0.9000	242710	12.56	206.01	over 1000	over 1000	12731307	5158955	-	-
san200_0.9_2	200	0.9000	178468	833.49	over 1000	over 1000	over 1000	303169816	-	-	-
san400_0.5_1	400	0.5000	7442	60.36	over 1000	over 1000	over 1000	43132933	-	-	-
sanr200_0.7	200	0.6969	16398	18.67	over 1000	over 1000	over 1000	55871909	-	-	-
sanr400_0.5	400	0.5011	8298	9.04	over 1000	over 1000	over 1000	36003126	-	-	-

3.6 Conclusion

In this chapter, we propose the vertex renumbering technique for MIP formulation of MEWCP, and a new exact algorithm EWCLIQUE for MEWCP. By computer experiments, we confirmed that proposed vertex renumbering technique improves the performance of mathematical programming solver by 14% on average. EWCLIQUE is based on branch-and-bound. For each subproblem, EWCLIQUE considers three components to calculate the upper bound for the weights of feasible solutions. In the upper bound calculation, EWCLIQUE regards some edge-weights as pseudo vertex weights for vertices. EWCLIQUE uses two upper bound calculations for rest edge-weights and pseudo vertex weights. Upper bounds are obtained by merging them. With some benchmarks, we compared proposed algorithm and some formulations solved by CPLEX. We confirmed proposed algorithm EWCLIQUE is faster than previous methods.

Chapter 4

Greedy Algorithms for MWVCP

4.1 Introduction

In this chapter, we propose two fast greedy algorithms for MWVCP of better average performance than previous approximation algorithms for MWVCP. Proposed algorithms are based on a greedy algorithm which removes vertices from a vertex cover initialized by V. The base greedy algorithm guarantees the minimality of solutions, and moreover, computation time is linear to the size of the given graph. Since the base greedy algorithms is very simple, it is faster than known algorithms, but it finds worse solutions. To obtain better solutions, the strategy of proposed algorithms is to construct a large number of feasible solutions by a greedy algorithm without increasing the computational complexity per solution. We confirm that proposed algorithms find better solutions and the computation time is shorter than previous algorithms.

In the section 4.2, we describe a simple greedy algorithm for MWVCP. Based on the base algorithm, two proposed algorithms are proposed in the section 4.3. Computer experiments are shown in the section 4.4.

4.2 Greedy algorithm

In this section, we introduce a simple greedy algorithm to find a minimal vertex cover. Given a graph G and an arbitrary permutation Π of V, Algorithm 11 constructs a minimal vertex cover C, where $N(v)$ denotes the neighbors of vertex v. It initializes C by V and eliminates unnecessary vertices from C in order of Π.

Algorithm 11 Greedy Elimination

```
INPUT: a graph $G = (V, E)$, a permutation $\Pi$ of $V$
OUTPUT: a minimal vertex cover $C$ of $G$
1: $C \leftarrow V$
2: for all $v_i$ in order of $\Pi$ do
3:     if $N(v_i) \subseteq C$ then
4:         $C \leftarrow C \setminus \{v_i\}$
5:     end if
6: end for
7: return $C$
```

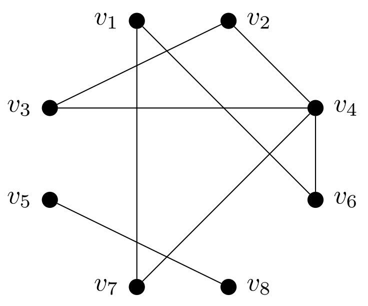
Lemma 4. *Algorithm 11 obtains a minimal vertex cover C*

Proof. In line 1, $C = V$ is obviously a vertex cover. In line 4, any neighbor of v_i is in C, and $C \setminus \{v_i\}$ is a vertex cover. Namely, C is always a vertex cover of G. Let v_j be a vertex that does not satisfy the condition in line 3. From the condition, there exists a vertex u that satisfies $u \in N(v_j)$ and $u \notin C$. Therefore $C \setminus \{v_j\}$ is not a vertex cover because it does not cover the edge (v_j, u). From the above, after the **for** loop, C is a minimal vertex cover. □

Algorithm 12 shows a liner time implementation of Greedy Elimination. S is a subset of C which always satisfies $S = \{v_i \mid N(v_i) \subseteq C\}$. In other words, line 4 of Algorithm 12 is equivalent to line 3 of Algorithm 11. S is updated when vertices are removed from C(line 6). Figure 4.2 shows the process how Algorithm 12 constructs a minimal vertex cover for the graph $G1$ in Figure 4.1 where $\Pi = [v_1, v_5, v_6, v_8, v_7, v_3, v_4, v_2]$.

Algorithm 12 A linear Time Implementation of Greedy Elimination

```
INPUT: a graph $G = (V, E)$, a permutation $\Pi$ of $V$
OUTPUT: a minimal vertex cover $C$ of $G$
1: $C \leftarrow V$
2: $S \leftarrow V$
3: for all $v_i$ in order of $\Pi$ do
4:     if $v_i \in S$ then                    ▷ $S$ always satisfies $S = \{v_i \mid N(v_i) \subseteq C\}$.
5:         $C \leftarrow C \setminus \{v_i\}$
6:         $S \leftarrow S \setminus (N(v_i) \cup \{v_i\})$
7:     end if
8: end for
9: return $C$
```



adjacency list
$N(v_1) = [v_6, v_7]$
$N(v_2) = [v_3, v_4]$
$N(v_3) = [v_2, v_4]$
$N(v_4) = [v_2, v_3, v_6, v_7]$
$N(v_5) = [v_8]$
$N(v_6) = [v_1, v_4]$
$N(v_7) = [v_1, v_4]$
$N(v_8) = [v_5]$

vertex	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
weight	9	1	1	4	7	8	8	4

Figure 4.1: Weighted graph $G1$

process	C	S	bit vector of S
initial	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	11111111
eliminate $v_1 \in S$	$\{v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_2, v_3, v_4, v_5, v_8\}$	01111001
eliminate $v_5 \in S$	$\{v_2, v_3, v_4, v_6, v_7, v_8\}$	$\{v_2, v_3, v_4\}$	01110000
skip $v_6 \notin S$	$\{v_2, v_3, v_4, v_6, v_7, v_8\}$	$\{v_2, v_3, v_4\}$	01110000
skip $v_8 \notin S$	$\{v_2, v_3, v_4, v_6, v_7, v_8\}$	$\{v_2, v_3, v_4\}$	01110000
skip $v_7 \notin S$	$\{v_2, v_3, v_4, v_6, v_7, v_8\}$	$\{v_2, v_3, v_4\}$	01110000
eliminate $v_3 \in S$	$\{v_2, v_4, v_6, v_7, v_8\}$	$\emptyset$	00000000
return C	$\{v_2, v_4, v_6, v_7, v_8\}$	-	-

$\Pi = [v_1, v_5, v_6, v_8, v_7, v_3, v_4, v_2]$

Figure 4.2: An example of constructing a minimal vertex cover by Algorithm 12

Here we describe the time complexity of Algorithm 12. Hereafter we suppose that G is represented by an adjacency-list and S is implemented by a bit vector. In the **for** loop, line 6 is processed in $O(|N(v_i)|)$ time because S is implemented by a bit vector and $N(v_i)$ is a vertex list. Other lines in the loop can be processed in $O(1)$ time. Therefore, the time complexity of Algorithm 12 is $O(n + m)$, where m denotes the number of edges in the graph. In addition, for every iteration, the size of C decreases by 1 if and only if the condition in line 4 is satisfied. The minimum cardinality of vertex covers is $n - \alpha$, where α

is cardinality of a maximum independent set of G. Therefore the condition in line 4 is satisfied at most α times. Moreover, line 6 takes $O(\Delta)$ time because $|N(v_i)| \leq \Delta$ for each i, where Δ is the maximum degree of the graph. Hence the time complexity of Algorithm 12 is evaluated as $O(n + \min\{m, \alpha\Delta\})$.

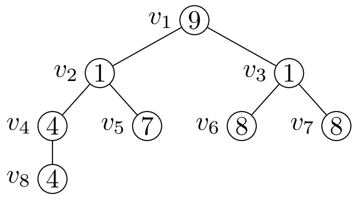
4.3 Proposed greedy algorithms

We propose two algorithms, *rotating greedy elimination* (RGE) and *branching greedy elimination* (BGE). Both of them call Algorithm 12 many times to construct a large number of feasible solutions, and finally choose the best solution C_{best}. The time complexity of proposed algorithms is $O(n + \min\{m, \alpha\Delta\})$ per solution. First, both of them construct a vertex permutation Π. After that, RGE constructs a number of feasible solutions by rotating Π, and BGE searches feasible solutions by depth-limited depth first search in order of Π. Therefore, the quality of solutions is strongly depend on the initial vertex permutation. We propose an effective method to obtain desirable Π in 4.3.1 and explain RGE and BGE in 4.3.2 and 4.3.3, respectively.

4.3.1 Vertex permutation

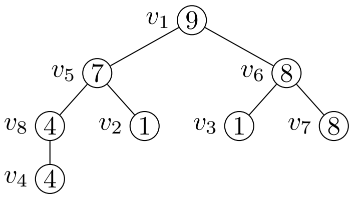
A *desirable* permutation for the greedy elimination is a permutation where *worthless* vertices appear earlier than *worthier* vertices. Worthless vertices means vertices with lower probability to be included in minimum weight vertex covers. For MWVCP, proposed vertex ordering assumes vertices of smaller weight as worthier vertices and vertices of larger weight as worthless vertices. For vertices of same weight, vertices of large degree are assumed as worthier than vertices of small degree. Since computation time of the greedy elimination is very short, proposed vertex ordering algorithm does not use sorting of $O(n \log n)$ time but 2-heap construction of $O(n)$ time.

For the graph $G1$, Figure 4.3a shows the initial binary tree according to the permutation $\Pi = [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8]$. The vertex weights are shown in the circles. First proposed permutation construction algorithm arranges the permutation as ith vertex is worthless than $(2i)$th vertex and $(2i+1)$th vertex. This can be done as same as constructing 2-heap on an array (Figure 4.3b). Then proposed permutation construction algorithm rebuilds the binary tree from the tail to the head of the permutation (shown in Figure 4.3c). By the 2-heap construction to the tree in Figure 4.3c, the permutation is arranged as ith vertex from last is worthless than $(2i)$th vertex from last and $(2i+1)$th vertex from the last. The constructed permutation is a 2-heap (Figure 4.3d). 2-heap construction is done by comparing 2 elements and swapping $O(n)$ times. Therefore the time complexity of this permutation construction is $O(n)$.



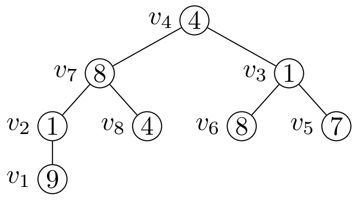
$\Pi = [v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8]$

(a) Initial binary tree



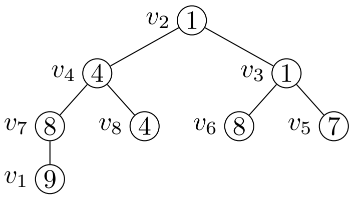
$\Pi = [v_1, v_5, v_6, v_8, v_2, v_3, v_7, v_4]$

(b) Heapify once



$\Pi = [v_1, v_5, v_6, v_8, v_2, v_3, v_7, v_4]$

(c) Rebuild binary tree



$\Pi = [v_1, v_5, v_6, v_8, v_7, v_3, v_4, v_2]$

(d) Heapify again

Figure 4.3: Permutation construction

4.3.2 Rotating greedy elimination (RGE)

For a permutation $\Pi = [v_1, v_2, \dots, v_n]$, let a rotating operation $\mathrm{rot}(\Pi) = [v_2, v_3, \dots, v_n, v_1]$. RGE, shown in Algorithm 13 requires an input parameter l_r such that $l_r \leq n$. RGE makes l_r permutations with $\mathrm{rot}(\cdot)$ and constructs l_r feasible solutions by greedy elimination for them. Finally, it chooses the best one among them.

The procedure MINIMALIZE is almost same as Algorithm 12. The difference is line 16 in Algorithm 13. It prunes greedy elimination if there is no possibility to get better solution than C_{best}. The value of $w(C) - w(S)$ is used as an lower bound for feasible solutions at line 16.

Figure 4.4 shows an example of RGE for the graph $G1$. In this case, C of 1st iteration is same as C constructed in Figure 4.2 because they use same Π. After 1st iteration, RGE uses another permutation to construct C and update C_{best}.

The time complexity of the procedure MINIMALIZE is $O(n + \min\{m, \alpha\Delta\})$, same as Algorithm 12. Hence the time complexity of Algorithm 13 is $O(l_r(n + \min\{m, \alpha\Delta\}))$.

Algorithm 13 RGE

```
INPUT: $G = (V, E)$, $w(\cdot)$, $\Pi = [v_1, v_2, \dots, v_n]$, $l_r$
OUTPUT: a vertex cover $C_{best}$
GLOBAL VARIABLES: $C_{best}$
 1: $C_{best} \leftarrow V$
 2: $P \leftarrow \Pi$
 3: for $i = 1$ to $l_r$ do
 4:     MINIMALIZE($P$)
 5:     $P \leftarrow \mathrm{rot}(P)$
 6: end for
 7: return $C_{best}$

 8: procedure MINIMALIZE($P$)
 9:     $C \leftarrow V$
10:     $S \leftarrow V$
11:     for all $v_i$ in order of $P$ do
12:         if $v_i \in S$ then
13:             $C \leftarrow C \setminus \{v_i\}$
14:             $S \leftarrow S \setminus (N(v_i) \cup \{v_i\})$
15:         end if
16:         if $w(C) - w(S) \geq w(C_{best})$ then
17:             return
18:         end if
19:     end for
20:     $C_{best} \leftarrow C$                    ▷ $w(C) < w(C_{best})$ holds because of line 16.
21:     return
22: end procedure
```

process	P	C after MINIMALIZE	C_{best}	$w(C_{best})$
initial	$[v_1, v_5, v_6, v_8, v_7, v_3, v_4, v_2]$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	42
1st iteration	$[v_1, v_5, v_6, v_8, v_7, v_3, v_4, v_2]$	$\{v_2, v_4, v_6, v_7, v_8\}$	$\{v_2, v_4, v_6, v_7, v_8\}$	25
2nd iteration	$[v_5, v_6, v_8, v_7, v_3, v_4, v_2, v_1]$	$\{v_1, v_2, v_4, v_8\}$	$\{v_1, v_2, v_4, v_8\}$	18
3rd iteration	$[v_6, v_8, v_7, v_3, v_4, v_2, v_1, v_5]$	pruned	$\{v_1, v_2, v_4, v_8\}$	18
4th iteration	$[v_8, v_7, v_3, v_4, v_2, v_1, v_5, v_6]$	pruned	$\{v_1, v_2, v_4, v_8\}$	18
return C_{best}	-	-	$\{v_1, v_2, v_4, v_8\}$	18

Figure 4.4: An example of RGE ($l_r = 4$)

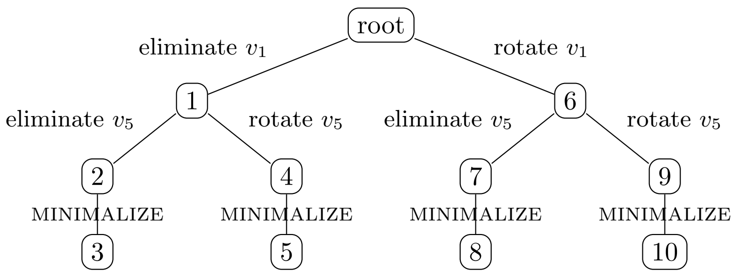
4.3.3 Branching greedy elimination (BGE)

BGE is shown in Algorithm 14. Given an input parameter l_b, BGE constructs at most 2^{l_b} feasible solutions by greedy elimination and selects the best solution among them. It searches solutions by depth first search whose depth is limited to l_b. After the depth becomes to l_b, it calls the greedy elimination to obtain minimalized vertex cover C.

In procedures EXPAND and MINIMALIZE, S is a subset of C such that any vertex in S of which might be eliminated from C later. The procedure MINIMALIZE is the greedy elimination with pruning. The procedure EXPAND calls MINIMALIZE 2^{l_b} times at most. When $\mathit{depth} < l_b$, it finds the first $v_i \in S$ in order of P (loop at line 16), and then explores the following two cases. In the first case, BGE eliminates v_i from C (line 20). In the other case, BGE rotates P by rot() operation (line 21). This is because v_i may be eliminated to minimalize C later. When $\mathit{depth} = l_b$, the procedure EXPAND calls MINIMALIZE to obtain minimalized C. BGE prunes subproblems of no possibility to get better solution than C_{best}(line 11 and 33). The value of $w(C) - w(S)$ is used as a lower bound for feasible solutions.

Figure 4.5 shows an example of search tree by BGE for the graph $G1$. In this case, C of search tree node 3 is same as C constructed in Figure 4.2. After that, BGE updates C_{best} by branching procedure.

EXPAND and MINIMALIZE are called at most 2^{l_b} times. Each line of the procedure EXPAND can be done in $\mathrm{O}(n)$ excluding recursive calls. Same as Algorithm 13, the time complexity of the procedure MINIMALIZE is $\mathrm{O}(n + \min\{m, \alpha\Delta\})$. In summary, the time complexity of Algorithm 14 is $\mathrm{O}(2^{l_b}(n + \min\{m, \alpha\Delta\}))$.



process	P	S	C	C_{best}	$w(C_{best})$
root	$[v_1, v_5, v_6, v_8, v_7, v_3, v_4, v_2]$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	42
1	$[v_1, v_5, v_6, v_8, v_7, v_3, v_4, v_2]$	$\{v_2, v_3, v_4, v_5, v_8\}$	$\{v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	42
2	$[v_5, v_6, v_8, v_7, v_3, v_4, v_2]$	$\{v_2, v_3, v_4\}$	$\{v_2, v_3, v_4, v_6, v_7, v_8\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	42
3	-	-	$\{v_2, v_4, v_6, v_7, v_8\}$	$\{v_2, v_4, v_6, v_7, v_8\}$	25
4	$[v_6, v_8, v_7, v_3, v_4, v_2, v_5]$	$\{v_2, v_3, v_4, v_5, v_8\}$	$\{v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_2, v_4, v_6, v_7, v_8\}$	25
5	-	-	$\{v_2, v_4, v_5, v_6, v_7\}$	$\{v_2, v_4, v_6, v_7, v_8\}$	25
6	$[v_5, v_6, v_8, v_7, v_3, v_4, v_2, v_1]$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_2, v_4, v_6, v_7, v_8\}$	25
7	$[v_5, v_6, v_8, v_7, v_3, v_4, v_2, v_1]$	$\{v_1, v_2, v_3, v_4, v_6, v_7\}$	$\{v_1, v_2, v_3, v_4, v_6, v_7, v_8\}$	$\{v_2, v_4, v_6, v_7, v_8\}$	25
8	-	-	$\{v_1, v_2, v_4, v_8\}$	$\{v_1, v_2, v_4, v_8\}$	18
9	$[v_6, v_8, v_7, v_3, v_4, v_2, v_1, v_5]$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$	$\{v_1, v_2, v_4, v_8\}$	18
10	-	-	pruned	$\{v_1, v_2, v_4, v_8\}$	18
return C_{best}	-	-	-	$\{v_1, v_2, v_4, v_8\}$	18

Figure 4.5: An example of search tree by BGE ($l_b = 2$)

Algorithm 14 BGE

```
INPUT: $G = (V, E)$, $w(\cdot)$, $\Pi = [v_1, v_2, \ldots, v_n]$, $l_b$
OUTPUT: a vertex cover $C_{best}$
GLOBAL VARIABLES: $C_{best}$
 1: $C_{best} \leftarrow V$
 2: EXPAND$(V, \Pi, V, 0)$
 3: return $C_{best}$

 4: procedure EXPAND$(C, P, S, depth)$
 5:     if $S = \emptyset$ then
 6:         if $w(C_{best}) > w(C)$ then
 7:             $C_{best} \leftarrow C$
 8:         end if
 9:         return
10:     end if
11:     if $w(C) - w(S) \geq w(C_{best})$ then
12:         return
13:     end if
14:     if $depth < l_b$ then
15:         $v_i \leftarrow$ the first vertex of $P$
16:         while $v_i \notin S$ do
17:             remove $v_i$ from $P$
18:             $v_i \leftarrow$ the first vertex of $P$
19:         end while
20:         EXPAND$(C \setminus \{v_i\}$, $P$, $S \setminus (N(v_i) \cup \{v_i\})$, $depth + 1)$
21:         EXPAND$(C$, rot$(P)$,$S$, $depth + 1)$
22:     else
23:         MINIMALIZE$(C$, $P$, $S)$
24:     end if
25:     return
26: end procedure

27: procedure MINIMALIZE$(C$, $P$, $S)$
28:     for all $v_i$ in order of $P$ do
29:         if $v_i \in S$ then
30:             $C \leftarrow C \setminus \{v_i\}$
31:             $S \leftarrow S \setminus (N(v_i) \cup \{v_i\})$
32:         end if
33:         if $w(C) - w(S) \geq w(C_{best})$ then
34:             return
35:         end if
36:     end for
37:     $C_{best} \leftarrow C$
38:     return
39: end procedure
```

4.4 Computer experiments

By some numerical experiments, we compare RGE and BGE with previous algorithms for MWVCP : CLA [18], BAR [7] and PIT [48], but do not compare with NT [41] or BE [8, 29] because the experiments in [58] shows they are obviously worse than others. In addition, a post-processing DW [58] is applied to each previous algorithm. Namely, CLA with DW, BAR with DW and PIT with DW (denoted by CLA+DW, BAR+DW, PIT+DW) are also compared.

All algorithms are implemented in C++. The CPU used in experiments is Intel®Core$^{\mathrm{TM}}$i7-6700 CPU 3.40 GHz. The memory is 16GB and the OS is Linux 4.2.0. The compiler is g++ 5.2.1 with optimize option O2.

For some small instances, we measured time of repeating same calculation 100 times and calculate average time by dividing it by 100 because calculation time is shorter than 1 msec.

In each row of the tables shown in this section, the best value (minimum weight) is indicated by "$\star$". If weights obtained by proposed algorithms are better than all previous algorithms, they are indicated by bold. Calculation time of less than 0.1 msec is denoted by ϵ.

4.4.1 random graphs

We generated uniform random graphs where n is up to 50000 and edge-density (denoted by d) is from 0.1 to 0.9. The vertex weight pattern is unweighted (all vertex weight is 1), integers from 1 to 10, and integers from 1 to 1000. However, for all patterns of vertex weight, the performance is relatively similar for each algorithms. Therefore only the results of integers from 1 to 10 are shown below. For each condition, we generate 10 random graphs and calculate averages for them. The weights of vertex covers obtained from random graphs are shown in Table 4.2. Table 4.3 shows the calculation time for random graphs. The best(minimum) weight for each row is indicated by "$\star$". The summary of the results is in Table 4.1.

Table 4.1: Summary: number of $\star$ in 30 conditions of random graphs

RGE	BGE	CLA+DW	BAR+DW	PIT+DW
6	22	2	0	0

For almost all graphs, proposed algorithms can obtain better solutions than previous ones (indicated by bold). In addition, for 28 of 30 graphs, the best weight for each row is obtained by proposed algorithm(indicated by "$\star$"). The simple greedy elimination cannot obtain better solutions than previous algorithms. This means proposal methods *rotating* and *branching* works effectively. As larger l_r and l_b, weights obtained by them are better.

For graphs of small n, BGE is better than RGE. For graphs of large n, RGE is better. Branching greedy elimination searches at most 2^{l_b} solutions in total. Half of them do not include the most *worthless* vertex v_1 that appears first in the permutation Π. Rotating greedy elimination searches l_r solutions in total. Each of them does not includes each of $v_1, v_2, \ldots, v_{l_r}$. That is, BGE has investigates the case excluding v_1 intensively, whereas RGE widely searches the solution space. This difference is shown in the results. For larger graphs, more diversity is required to obtain better solutions.

For many graphs, the computation time of proposal algorithms are shorter than previous ones. For some small graphs, proposed algorithm are not faster than previous ones, but still the computation time is very short. As larger graphs, the computation time of previous algorithms increases more than proposed. The computation of proposed algorithms can be controlled with the parameters.

In Table 4.4, the values of algorithms for some instances are compared with optimal solutions obtained by an exact MWCP solver OTClique shown in the chapter 2. For sparse graphs, there are large differences between the optimal weight and the weight obtained by proposed algorithms. They become smaller as larger d. On the other hand, as n becomes larger, the differences do not become larger. The tendency can be also confirmed on previous algorithms.

Table 4.2: The weight of vertex covers obtained from random graphs

		greedy		RGE			BGE		CLA	CLA	BAR	BAR	PIT	PIT
n	d	elimination	$l_r = 64$	$l_r = 256$	$l_r = 1024$	$l_b = 6$	$l_b = 8$	$l_b = 10$		+ DW		+ DW		+ DW
500	0.1	2395.9	2389.4	2389.1	2388.6	2381.6	**2372.1**	⋆**2367.9**	2478.4	2377.7	2627.1	2438.2	2622.9	2450.4
500	0.3	2573.6	**2566.9**	**2565.7**	**2565.7**	**2560.2**	**2555.2**	⋆**2551.3**	2627.4	2572.6	2685.4	2588.5	2679.2	2594.2
500	0.5	2628.8	**2617.3**	**2617.3**	**2617.3**	**2613.7**	**2611.5**	⋆**2610.8**	2662.2	2624.5	2690.3	2633.5	2689.8	2640.7
500	0.7	**2654.5**	**2645.9**	**2645.6**	**2645.6**	**2645.9**	**2643.6**	⋆**2642.4**	2679.7	2654.8	2693.1	2662.0	2694.6	2665.1
500	0.9	2676.3	**2667.9**	**2667.1**	**2667.1**	**2666.9**	**2666.7**	⋆**2666.5**	2686.7	2674.5	2699.0	2681.7	2700.6	2681.0
1000	0.1	5130.5	5103.7	5103.7	5103.7	5103.8	**5095.3**	⋆**5084.8**	5258.0	5099.3	5428.5	5166.9	5409.0	5186.9
1000	0.3	5356.8	**5339.2**	**5338.8**	**5338.8**	**5333.4**	**5331.2**	⋆**5327.5**	5425.1	5347.3	5478.0	5373.7	5463.8	5381.8
1000	0.5	5417.7	**5401.1**	**5401.0**	**5401.0**	**5400.3**	**5398.2**	⋆**5394.0**	5461.6	5414.2	5489.3	5421.6	5486.1	5426.3
1000	0.7	5444.5	**5435.3**	**5434.9**	**5434.9**	**5435.8**	**5433.6**	⋆**5431.7**	5476.6	5447.9	5494.0	5449.8	5493.5	5455.4
1000	0.9	5469.2	**5459.4**	**5459.0**	**5459.0**	**5458.9**	**5457.8**	⋆**5457.0**	5485.6	5467.9	5496.8	5472.6	5496.3	5474.4
2500	0.1	13303.9	**13280.6**	**13279.9**	**13279.9**	**13272.9**	**13262.7**	⋆**13256.0**	13514.6	13285.5	13691.5	13354.0	13665.6	13352.3
2500	0.3	13600.1	**13579.7**	**13578.7**	**13578.0**	**13577.1**	**13572.1**	⋆**13565.7**	13688.5	13591.2	13745.4	13612.5	13740.9	13614.6
2500	0.5	13674.8	**13656.2**	**13654.5**	**13654.4**	**13655.9**	**13650.2**	⋆**13647.3**	13718.5	13669.5	13756.8	13674.6	13752.9	13681.8
2500	0.7	13705.9	**13693.7**	**13693.1**	**13693.1**	**13694.8**	**13693.6**	⋆**13690.2**	13741.5	13704.8	13761.8	13709.8	13759.9	13712.1
2500	0.9	13733.7	**13723.0**	**13721.8**	**13721.8**	**13723.7**	**13722.0**	⋆**13721.7**	13753.5	13733.2	13760.4	13737.1	13760.5	13736.1
5000	0.1	26871.9	**26848.5**	**26845.0**	**26845.0**	**26847.6**	**26838.9**	⋆**26830.5**	27151.0	26857.1	27328.9	26918.4	27317.2	26939.1
5000	0.3	27219.2	**27199.1**	**27197.8**	**27197.8**	**27194.8**	**27191.0**	⋆**27186.2**	27334.1	27216.3	27386.3	27232.1	27385.3	27230.7
5000	0.5	27300.1	**27284.4**	**27282.2**	**27281.6**	**27283.2**	**27279.7**	⋆**27277.2**	27371.1	27298.5	27394.8	27305.6	27393.6	27315.2
5000	0.7	27340.2	**27328.4**	**27326.7**	**27326.6**	**27328.2**	**27325.9**	⋆**27323.3**	27385.1	27337.0	27398.2	27348.0	27397.2	27346.9
5000	0.9	27368.9	**27360.7**	**27359.1**	⋆**27358.0**	**27360.3**	**27360.2**	**27359.5**	27394.1	27368.5	27403.7	27375.3	27404.5	27377.0
25000	0.1	136909.9	**136865.9**	**136861.2**	**136860.5**	**136866.6**	**136855.0**	⋆**136847.6**	137336.2	136893.7	137506.0	136953.0	137496.1	136962.5
25000	0.3	137354.6	**137332.1**	**137327.2**	**137325.6**	**137330.3**	**137328.2**	⋆**137321.5**	137526.2	137354.2	137564.4	137364.3	137553.6	137372.7
25000	0.5	137463.3	**137444.5**	**137440.9**	**137437.3**	**137441.1**	**137438.4**	⋆**137435.4**	137553.2	137459.0	137578.4	137467.9	137570.4	137463.7
25000	0.7	137509.7	**137499.1**	**137496.8**	⋆**137493.6**	**137499.1**	**137496.4**	**137493.7**	137572.2	137508.0	137582.4	137512.6	137582.0	137516.0
25000	0.9	137546.2	**137534.8**	**137533.5**	⋆**137532.7**	**137536.2**	**137535.4**	**137535.0**	137579.2	137544.2	137585.9	137550.4	137585.5	137550.1
50000	0.1	273380.6	273333.1	273309.3	273262.7	273337.5	273311.8	273299.6	273951.3	⋆272981.9	274529.9	273222.5	274445.8	273209.2
50000	0.3	274273.2	274236.6	274201.3	274150.2	274224.5	274209.0	274203.1	274560.5	⋆274105.5	274856.5	274206.0	274831.4	274202.8
50000	0.5	274517.5	274470.4	274450.8	⋆**274425.7**	274465.4	274459.7	274448.3	274764.5	274429.2	274927.8	274486.8	274913.4	274482.8
50000	0.7	274631.1	274604.0	274587.9	⋆**274575.2**	274598.2	274589.4	**274579.2**	274834.5	274583.3	274967.2	274621.5	274944.9	274628.0
50000	0.9	274712.5	274695.8	**274680.0**	⋆**274669.5**	**274686.5**	**274682.4**	**274674.5**	274892.5	274692.0	274981.9	274710.8	274970.9	274716.0

Table 4.3: CPU time for random graphs [msec]

		greedy	RGE			BGE			CLA	CLA	BAR	BAR	PIT	PIT
n	d	elimination	$l_r = 64$	$l_r = 256$	$l_r = 1024$	$l_b = 6$	$l_b = 8$	$l_b = 10$		+DW		+DW		+DW
500	0.1	ϵ	0.2	0.6	1.1	0.2	0.7	2.5	0.2	0.2	ϵ	ϵ	ϵ	ϵ
500	0.3	ϵ	0.2	0.5	1.0	0.1	0.5	1.7	0.4	0.5	0.1	0.1	0.1	0.1
500	0.5	ϵ	0.1	0.5	0.9	0.1	0.4	1.2	0.7	0.7	0.2	0.2	0.2	0.2
500	0.7	ϵ	0.1	0.4	0.8	0.1	0.3	0.5	1.0	1.0	0.2	0.3	0.3	0.3
500	0.9	ϵ	0.1	0.4	0.7	0.1	0.1	0.2	1.1	1.2	0.3	0.4	0.3	0.4
1000	0.1	ϵ	0.4	1.3	4.1	0.4	1.4	5.1	0.6	0.7	0.2	0.2	0.2	0.2
1000	0.3	ϵ	0.3	1.2	3.8	0.3	1.0	3.7	1.7	1.7	0.4	0.5	0.4	0.5
1000	0.5	ϵ	0.3	1.1	3.6	0.3	0.8	2.6	2.7	2.8	0.7	0.8	0.7	0.8
1000	0.7	ϵ	0.3	0.9	3.2	0.2	0.6	1.4	3.7	3.9	0.9	1.1	0.9	1.1
1000	0.9	ϵ	0.2	0.8	2.8	0.1	0.2	0.4	4.4	4.7	1.2	1.5	1.2	1.5
2500	0.1	ϵ	1.1	3.6	12.0	1.0	3.5	12.9	3.3	3.5	0.9	1.1	0.9	1.2
2500	0.3	ϵ	1.0	3.3	11.4	0.9	2.9	10.2	9.0	9.4	2.5	3.1	2.6	3.1
2500	0.5	ϵ	0.9	3.0	10.6	0.8	2.4	7.7	14.8	15.4	4.2	5.1	4.3	5.1
2500	0.7	ϵ	0.8	2.6	9.4	0.6	1.7	4.3	20.4	21.4	5.8	7.0	5.9	7.1
2500	0.9	ϵ	0.6	2.0	7.2	0.4	0.7	1.2	25.6	26.6	7.4	9.3	7.5	9.5
5000	0.1	ϵ	2.4	8.3	29.5	2.3	8.4	31.0	13.3	13.8	3.6	4.5	3.8	4.6
5000	0.3	ϵ	2.3	7.8	28.5	2.0	6.5	24.0	36.5	37.9	10.3	12.5	10.5	12.6
5000	0.5	ϵ	1.9	6.6	24.4	1.6	5.3	17.5	59.1	61.8	16.7	19.8	17.0	20.3
5000	0.7	ϵ	1.7	5.8	21.4	1.4	3.9	10.4	81.8	85.4	23.2	27.8	23.4	27.5
5000	0.9	ϵ	1.4	4.8	17.1	0.9	1.8	3.2	103.6	107.4	29.6	36.0	29.9	35.7
25000	0.1	0.6	16.1	60.2	228.0	15.9	57.3	221.0	373.3	391.2	92.3	108.8	93.5	110.0
25000	0.3	0.5	13.0	47.3	178.6	12.0	43.0	159.8	991.2	1039.5	254.9	296.1	257.6	299.2
25000	0.5	0.5	11.3	40.6	151.2	10.0	33.3	119.1	1597.7	1672.1	413.4	490.1	417.9	481.7
25000	0.7	0.5	9.9	35.0	129.1	7.6	24.0	76.0	2204.2	2307.2	572.7	662.0	578.5	684.5
25000	0.9	0.5	7.3	24.9	93.1	5.3	11.7	24.1	2819.2	2968.0	732.4	859.7	739.8	863.2
50000	0.1	1.3	56.4	218.9	821.6	57.2	213.1	850.0	1088.9	1175.8	282.2	365.3	287.7	367.7
50000	0.3	1.3	43.0	162.9	599.7	44.7	162.1	630.9	2764.3	2996.3	735.0	936.2	745.9	944.5
50000	0.5	1.3	40.5	151.6	559.8	42.5	152.2	582.9	4281.4	4668.2	1189.6	1496.4	1205.8	1507.6
50000	0.7	1.3	38.3	144.2	545.9	38.9	139.3	531.6	5745.4	6262.8	1589.3	1998.0	1611.7	2013.3
50000	0.9	1.3	36.2	137.4	516.3	36.0	130.1	490.9	7105.6	7726.4	1955.8	2455.6	1984.2	2492.7

Table 4.4: Comparison with optimal solutions

		optimal	greedy	RGE			BGE			CLA	CLA	BAR	BAR	PIT	PIT
n	d	weight	elimination	$l_r = 64$	$l_r = 256$	$l_r = 1024$	$l_b = 6$	$l_b = 8$	$l_b = 10$		+DW		+DW		+DW
200	0.1	819.3	852.7	845.6	845.6	845.6	**839.4**	**838.2**	⋆**833.5**	899.6	842.5	1006.3	885.9	1001.2	898.2
200	0.2	913.8	942.2	936.2	936.2	936.2	**926.0**	**923.2**	⋆**921.1**	974.4	934.9	1050.1	960.8	1046.6	966.0
200	0.3	957.7	981.6	**972.6**	**972.6**	**972.6**	**965.3**	**961.8**	⋆**960.0**	1012.1	973.4	1061.7	993.6	1059.0	989.5
200	0.4	984.9	**999.9**	**993.6**	**993.6**	**993.6**	**991.6**	**988.3**	⋆**986.7**	1030.7	1002.8	1065.5	1016.0	1062.3	1013.8
200	0.5	1003.9	1021.4	**1010.3**	**1010.3**	**1010.3**	**1008.5**	**1005.9**	⋆**1005.4**	1036.3	1015.0	1072.0	1030.4	1066.6	1031.0
200	0.6	1019.2	1035.9	**1025.6**	**1025.6**	**1025.6**	**1022.7**	**1022.4**	⋆**1020.5**	1048.6	1029.9	1072.0	1042.7	1071.8	1042.7
200	0.7	1031.8	**1041.8**	**1035.5**	**1035.5**	**1035.5**	**1034.6**	**1034.1**	⋆**1033.4**	1061.1	1043.6	1076.5	1049.6	1072.6	1046.3
200	0.8	1042.4	1050.6	**1044.3**	**1044.3**	**1044.3**	**1043.2**	**1043.0**	⋆**1042.8**	1063.9	1049.6	1077.2	1059.5	1075.1	1059.3
200	0.9	1051.0	**1054.9**	**1052.9**	**1052.8**	**1052.8**	**1052.3**	**1052.0**	⋆**1051.9**	1068.9	1055.1	1077.7	1062.9	1076.0	1062.1
500	0.9	2665.4	2676.3	**2667.9**	**2667.1**	**2667.1**	**2666.9**	**2666.7**	⋆**2666.5**	2686.7	2674.5	2699.0	2681.7	2700.6	2681.0
1000	0.9	5456.2	5469.2	**5459.4**	**5459.0**	**5459.0**	**5458.9**	**5457.8**	⋆**5457.0**	5485.6	5467.9	5496.8	5472.6	5496.3	5474.4
2000	0.9	10993.1	**11004.7**	**10995.1**	⋆**10994.7**	⋆**10994.7**	**10997.4**	**10995.5**	**10995.1**	11027.6	11005.7	11034.7	11011.6	11035.3	11011.3
3000	0.9	16374.5	16391.7	**16380.3**	⋆**16379.2**	⋆**16379.2**	**16381.2**	**16380.3**	**16379.9**	16411.1	16389.6	16422.4	16394.7	16418.1	16391.1
4000	0.9	22016.6	22032.9	**22023.2**	⋆**22022.3**	⋆**22022.3**	**22024.2**	**22023.2**	**22022.4**	22056.5	22029.4	22064.5	22036.7	22061.7	22035.3
5000	0.9	27353.1	27368.9	**27360.7**	**27359.1**	⋆**27358.0**	**27360.3**	**27360.2**	**27359.5**	27394.1	27368.5	27403.7	27375.3	27404.5	27377.0
6000	0.9	32877.5	32894.4	**32886.3**	**32883.7**	⋆**32882.7**	**32885.9**	**32884.6**	**32883.2**	32922.1	32893.4	32928.2	32900.6	32928.4	32902.8
7000	0.9	38517.5	**38533.1**	**38524.9**	**38523.1**	⋆**38521.9**	**38525.0**	**38524.3**	**38523.0**	38560.2	38535.3	38568.6	38537.1	38568.3	38539.2
8000	0.9	43981.2	**44000.1**	**43989.4**	**43986.4**	⋆**43985.5**	**43989.1**	**43988.8**	**43987.9**	44029.2	44000.1	44031.6	44002.0	44033.3	44005.3
9000	0.9	49461.5	**49478.8**	**49470.2**	**49468.4**	⋆**49466.9**	**49471.4**	**49470.6**	**49469.4**	49507.2	49479.6	49515.2	49482.1	49514.8	49482.8
10000	0.9	54850.7	**54869.2**	**54861.2**	**54858.9**	**54857.4**	**54859.6**	**54858.5**	⋆**54856.4**	54897.9	54869.2	54907.2	54875.4	54905.2	54872.7

4.4.2 graphs from error-correcting codes

It is important to compare algorithms with graphs of real applications. To construct error-correcting codes of maximum size can be formulated as MWCP[43]. We convert them to MWVCP and solve them by MWVCP algorithms. The summary is shown in Table 4.5. The weight of vertex covers obtained are in Table 4.6. Table 4.7 shows the calculation time for random graphs. For all graphs, proposed algorithm

Table 4.5: Summary: number of $\star$ in 18 instances of error-correcting code graphs

RGE	BGE	CLA+DW	BAR+DW	PIT+DW
6	12	0	0	0

can be obtain better weight than previous ones (indicated by bold). In addition, all of the best weight for each row is obtained by proposed algorithm (indicated by "$\star$"). The computation time behaves similarly to case of random graphs. For some small graphs, proposed algorithm are not faster than previous ones, but the computation time for them are very short. As larger graphs, the computation time of previous algorithms increases more than proposal algorithms.

In the Table 4.7, the computation time of the BGE of $l_b = 10$ is shorter than the RGE of $l_r = 1024$. This is caused by pruning searches at line 16 in Algorithm 13 and at line 11,33 in Algorithm 14. This fact shows the pruning is more effective in real applications than random graphs.

Table 4.6: The weight of vertex covers obtained from error-correcting code graphs

	optimal	greedy	RGE			BGE			CLA	CLA	BAR	BAR	PIT	PIT
graph	weight	elimination	$l_r = 64$	$l_r = 256$	$l_r = 1024$	$l_b = 6$	$l_b = 8$	$l_b = 10$		+DW		+DW		+DW
11-4-4	256	262	**258**	**258**	**258**	**258**	**258**	⋆**256**	284	258	283	265	283	265
12-4-6	790	**824**	**818**	**816**	**816**	**820**	**818**	⋆**812**	852	828	884	840	868	832
14-4-7	2828	**2900**	**2872**	**2872**	**2872**	**2886**	⋆**2858**	⋆**2858**	3012	2900	3040	2914	3012	2914
14-6-6	2367	2388	**2382**	**2381**	**2379**	**2381**	⋆**2376**	⋆**2376**	2400	2388	2403	2385	2403	2385
16-4-5	2618	**2660**	**2660**	**2660**	**2660**	**2660**	**2660**	⋆**2646**	2856	2723	2842	2695	2849	2702
16-8-8	8872	8890	⋆**8872**	⋆**8872**	⋆**8872**	⋆**8872**	⋆**8872**	⋆**8872**	8890	8880	8898	8887	8898	8889
17-4-4	1920	1940	**1932**	**1928**	**1928**	⋆**1920**	⋆**1920**	⋆**1920**	2012	1948	2028	1932	2044	1948
17-6-6	3072	**3091**	**3079**	**3079**	**3079**	**3079**	**3079**	⋆**3078**	3136	3091	3121	3092	3115	3092
19-4-6	16724	16940	**16844**	**16844**	**16844**	**16844**	**16844**	⋆**16804**	17732	16972	17752	17072	17832	16912
19-8-8	16744	16770	**16758**	**16758**	⋆**16756**	**16758**	**16758**	⋆**16756**	16788	16764	16786	16776	16786	16776
20-6-5	12920	**12944**	**12934**	**12930**	**12930**	**12934**	**12934**	⋆**12924**	12984	12954	12994	12944	12994	12954
20-6-6	29570	29640	⋆**29600**	⋆**29600**	⋆**29600**	**29620**	**29620**	⋆**29600**	29720	29640	29740	29620	29720	29640
20-8-10	44903	44950	**44912**	⋆**44903**	⋆**44903**	**44914**	**44914**	**44914**	44950	44950	44968	44914	44968	44914
21-10-9	19864	19871	**19867**	**19866**	⋆**19864**	**19867**	**19866**	**19866**	19885	19872	19886	19871	19882	19870
22-10-10	88910	88934	**88916**	**88916**	⋆**88914**	**88916**	**88916**	**88916**	88954	88934	88946	88926	88946	88935

Table 4.7: CPU time for error-correcting code graphs [msec]

	greedy	RGE			BGE			CLA	CLA	BAR	BAR	PIT	PIT
graph	elimination	$l_r = 64$	$l_r = 256$	$l_r = 1024$	$l_b = 6$	$l_b = 8$	$l_b = 10$		+DW		+DW		+DW
11-4-4	ϵ	0.1	0.2	0.2	ϵ	0.3	0.8	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
12-4-6	ϵ	0.2	0.3	0.3	0.1	0.4	1.3	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
14-4-7	ϵ	ϵ	0.3	0.3	ϵ	0.3	1.2	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
14-6-6	ϵ	0.3	0.9	3.1	0.3	0.7	2.1	1.7	1.8	0.4	0.5	0.4	0.6
16-4-5	ϵ	ϵ	0.2	0.3	ϵ	0.3	1.0	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
16-8-8	ϵ	0.7	1.9	6.5	0.4	0.8	1.3	24.4	25.5	5.8	7.8	6.0	8.7
17-4-4	ϵ	ϵ	0.2	0.1	ϵ	0.2	0.5	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ
17-6-6	ϵ	0.3	1.1	1.8	0.3	1.1	3.3	0.7	0.8	0.2	0.2	0.2	0.3
19-4-6	ϵ	0.1	0.5	0.5	0.2	0.5	1.8	ϵ	0.1	ϵ	ϵ	ϵ	ϵ
19-8-8	ϵ	1.0	3.2	11.1	0.7	1.5	2.6	18.9	19.4	4.5	6.0	4.8	6.5
20-6-5	ϵ	0.5	1.6	5.5	0.4	1.1	2.2	5.5	5.7	1.3	1.8	1.4	1.9
20-6-6	ϵ	0.5	1.6	6.2	0.6	1.7	3.3	7.2	7.6	1.8	2.3	1.9	2.5
20-8-10	ϵ	1.3	4.1	11.7	0.7	1.2	2.2	25.8	27.0	6.9	9.3	7.1	10.0
21-10-9	ϵ	3.2	12.5	48.2	2.6	5.5	11.2	113.5	116.1	27.9	35.1	28.4	38.9
22-10-10	0.2	1.7	6.2	23.5	0.8	1.1	1.5	390.9	424.6	101.4	133.3	102.5	144.3

4.5 Conclusion

In this chapter, we propose two efficient greedy algorithms for MWVCP. Proposed algorithms construct a vertex permutation that is ordered by *worthless* of vertices. Then they construct a large number of minimal vertex covers in greedy manner. By some numerical experiments, we confirmed proposed algorithms can obtain better solutions than previous 2-approximate algorithms for MWVCP. The computation time of proposed algorithms is shorter than others because each greedy procedure is done in short time.

There are some future works. Evaluation of approximation ratio is the most important future work. Besides it, improving performance in sparse graphs compared to optimal solutions is also important. Recently, algorithms based on metaheuristics were proposed for MWVCP [38, 69, 12], MWCP [65, 10] and MWIS [27]. Proposed algorithms can be used as a subroutine to construct initial solutions or improvement processes in metaheuristics.

Chapter 5

Data structures for local search algorithms on MEWCP

5.1 Introduction

In this chapter, we propose two data structures to manage neighborhoods for local search algorithms. One can be used for graphs represented by adjacency lists. The other is for graphs represented by adjacency matrix. For MEWCP, calculating the weight of a clique takes more time than MWCP because of edge weights. Proposed method also reduces the time complexity for clique weight calculation.

By some computer experiments, with local search algorithms for MEWCP, we compare all combinations of proposed data structures and a previous data structure. From the results, we confirmed proposed data structures are better than previous one. We confirmed that local search algorithms works efficiently using one of proposed data structures properly by memory capacity or graph property such as number of vertices or edge density.

In the section 5.2, some notations, definitions for local search and previous methods are described. The section 5.3 shows two proposed data structures for neighborhood management. Computer experiments are shown in the section 5.4.

5.2 Preliminary

5.2.1 Notation

For a simple undirected graph $G = (V, E)$, $w(u, v)$ denotes the weight of $(u, v) \in E$. For a clique $C \subseteq V$, let $w_e(C) = \sum_{u,v \in C} w(u, v)$. Let $d = \frac{2|E|}{|V|(|V|-1)}$. For any vertex $v \in V$, $N(v)$ denotes the set of vertices adjacent to v in G. For a clique C, let C_0 be a set of vertices that are adjacent to all vertices in C. Namely, C_0 satisfies the follow :

$$C_0 = \{v \mid C \subseteq N(v)\}.$$

Note that any $v \in C$ is not included in C_0 because it is not adjacent to itself. For a clique C, let C_1 be a set of vertices that have just one non-adjacent vertex in C and are not included in C. Namely, C_1 satisfies the follow:

$$C_1 = \{v \notin C \mid |C \setminus N(v)| = 1\}.$$

Same as [66, 50, 21], we define following three neighborhoods for a clique C :

Add-neighborhoods: A family of sets $N_{add}(C) = \{C \cup \{v\} \mid v \in C_0\}$. Namely, a set of cliques constructed by adding a vertex $v \in C_0$ to C.

Drop-neighborhoods: A family of sets $N_{drop}(C) = \{C \setminus \{v\} \mid v \in C\}$. Namely, a set of cliques constructed by removing a vertex from C.

Swap-neighborhoods: A family of sets $N_{swap}(C) = \{(C \cap N(v)) \cup \{v\} \mid v \in C_1\}$. Namely, a set of cliques constructed by removing vertices not adjacent to $v \in C_1$ from C and then adding v to C.

From the definition of C_1, $|C \setminus N(v)| = 1$ holds and the size of cliques in the swap-neighborhoods equals to $|C|$.

Local search algorithms iteratively move the clique C to one of these neighborhoods to search cliques that have larger weight.

5.2.2 Previous data structure for neighborhood management

Local search algorithms scan neighborhoods frequently. In this section, we show a previous data structure to manage and update neighborhoods for a graph represented in adjacency lists [21]. For each vertex $v \in V$ and a clique C, $\kappa(v, C)$ denotes the number of vertices that are included in C and adjacent to v. Namely, $\kappa(v, C)$ satisfies the following equation :

$$\kappa(v, C) = |C \cap N(v)|.$$

For a clique C, let $N(C)$ be a set of vertices that are not included in C and have at least one adjacent vertex in C. Namely, $N(C)$ satisfies the follow :

$$N(C) = \{v \notin C \mid \exists u \in C, v \in N(u)\}.$$

Using $\kappa(v, C)$ and $N(C)$, previous method defines vertex sets S_{add} and S_{swap} :

$$S_{add} = \{u \in N(C) \mid \kappa(u, C) = |C|\},$$

$$S_{swap} = \{u \in N(C) \mid \kappa(u, C) = |C| - 1\}.$$

From these definitions, $C_0 = S_{add}$ holds when $C > 0$, and $C_1 = S_{swap}$ holds when $|C| > 1$. Therefore the previous method in [21] manages C_0 and C_1 using $\kappa(v, C), N(C), S_{add}, S_{swap}$. Although previous method does not consider the cases of $|C| \leq 1$, the cases are only in very small or very sparse graphs.

Hereafter we assume that the time complexity of adding an element to a set and deleting an element from a set is $O(1)$. It can be implemented by hash tables, whose expected time complexity of add operation and delete operation is $O(1)$, or it can be done by representing sets by doubly linked lists with an array of addresses that are used to find vertex position in the lists in constant time.

$\kappa(v, C), N(C), S_{add}$ and S_{swap} are updated in moving a clique C to a neighborhood. Let v be the vertex added to or deleted from C. For each $u \in N(v)$, $\kappa(u, C)$ must be updated and it takes $O(|N(v)|)$ time. After updating $\kappa(u, C)$, for each $u \in N(v)$, previous method adds u to $N(C)$ if $\kappa(u, C) > 0$ and $u \notin C$. Same as $\kappa(u, C)$, updating $N(C)$ takes $O(|N(v)|)$ time. Finally, it scans $N(C)$ to update S_{add} and S_{swap} in $O(|N(C)|)$ time. In summary, previous method in [21] updates neighborhoods in $O(|N(v)| + |N(C)|)$ time when it moves a clique C.

5.3 Proposed data structures for neighborhood management

In this section, we propose two data structures for neighborhood management. Usually, adjacency lists are adopted for large sparse graphs because of space complexity. Proposed method L is an efficient method to manage neighborhoods in such cases. In contrast, for small graphs represented by adjacency matrices, proposed method M constructs non-adjacency lists and uses the list to reduce time complexity. Table 5.1 summarizes these neighborhood management methods.

For MEWCP, especially for graphs represented by adjacency lists, calculating weights of neighborhoods takes time because of the time to refer to edge weights. Hence we propose a data structure to calculate weights of neighborhoods with proposed method L.

Table 5.1: Summary of methods for neighborhood management

method	graph representation	time complexity to update	space complexity of data structures
Previous [21]	adjacency list	$O(\lvert N(v)\rvert + \lvert N(C)\rvert)$	$O(\lvert V\rvert)$
Proposal L	adjacency list	$O(\lvert N(v)\rvert)$	$O(\lvert V\rvert)$
Proposal M	adjacency matrix + non-adjacency list	$O(\lvert V \setminus N(v)\rvert)$	$O(\lvert V\rvert)$

5.3.1 Proposed method L

Same as [21], for a clique C and each vertex $v \in V$, let $\kappa(v, C) = |C \cap N(v)|$. Using $\kappa(v, C)$, proposed method L defines a vertex set $S(i, C)$ as :

$$S(i, C) = \{v \in V \setminus C \mid \kappa(v, C) = i\}.$$

Hereafter, $\kappa(v)$ denotes $\kappa(v, C)$ and $S(i)$ denotes $S(i, C)$ when C can be obviously identified. By the definition, $C_0 = S(|C|)$ and $C_1 = S(|C| - 1)$. Proposed method L manages neighborhoods by updating sets $S(\cdot)$ implemented in doubly linked list. It manages $\Delta + 1$ doubly linked lists $S(0), S(1), \ldots, S(\Delta)$ because $\kappa(v) \leq |N(v)|$ holds for any vertex v, where Δ is a maximum degree of all vertices. Since the total size of the all doubly linked lists is $|V|$ at most, the space complexity is $O(|V|)$. In the initial state where $C = \emptyset$, $\kappa(v) = 0$ for all $v \in V$ and all vertices are included in $S(0)$. $S(1), S(2), \ldots, S(\Delta)$ are initialized by empty doubly linked lists that have only a head element. The relationship between the sets S_{add}, S_{swap} used in previous method, the sets $S(i)$ used in proposed method L, C_0 and C_1 is summarized as follows.

S_{add} : $S_{add} = C_0$ holds when $|C| \neq 0$. $S_{add} \subseteq N(C)$ holds. When C is moved to a neighborhood, previous method scans $N(C)$ and selects vertices of $\kappa(v) = |C|$ to update S_{add}.

S_{swap} : $S_{swap} = C_1$ holds when $|C| > 1$. $S_{swap} \subset N(C)$ holds. When C is moved to a neighborhood, previous method scans $N(C)$ and selects vertices of $\kappa(v) = |C| - 1$ to update S_{swap}.

$S(i)$: A family of $\Delta + 1$ sets ($i = 0, \ldots, \Delta$). $S(|C|) = C_0$ and $S(|C| - 1) = C_1$ hold. When C is moved to a neighborhood, necessity minimum vertices are moved to appropriate $S(i)$ as described bellow.

To remove a vertex from the doubly linked list in constant time, proposed method L uses an array *pos*(v) that have address of each vertex v. In removing a vertex $v \in S(\kappa(v))$ from the doubly linked list, it can refer to the address of v in constant time. Hence it can remove v in constant time. In the initial state, $C = \emptyset$, for each $v \in V$, *pos*(v) is initialized by the address of v in $S(0)$.

In addition, to calculate the weights of neighborhoods efficiently, proposed method L defines $\sigma(v, C)$ for a clique C and each vertex $v \in V$ as follows :

$$\sigma(v, C) = \sum_{u \in C \cap N(v)} w(u, v).$$

Hereafter $\sigma(v)$ denotes $\sigma(v, C)$. In the initial state, $C = \emptyset$, they are initialized by $\sigma(v) = 0$ for all $v \in V$. Given the values of $\sigma(v)$, the weight of a neighborhood can be calculated in $O(1)$ time as follows. For a clique C, the weight of a clique $C \cup \{v\}$ in add-neighborhoods is $w_e(C) + \sigma(v)$. The weight of a clique $C \setminus \{u\}$ in drop-neighborhood is $w_e(C) - \sigma(u)$. For a clique $(C \setminus \{u\}) \cup \{v\}$ in swap-neighborhood, u and v are not adjacent to each other. Therefore moving to swap-neighborhood is equivalent to the sequential movement to drop-neighborhood and then to add-neighborhood. Hence the weight of a clique $(C \setminus \{u\}) \cup \{v\}$ in swap-neighborhood is $w_e(C) - \sigma(u) + \sigma(v)$. For the graph G_{ex} shown in Figure 5.1 and a clique $C = \{v_4, v_6, v_7\}$, Figure 5.2 shows the data structure of proposed method L. In the case, $\Delta = 5$ and proposed method L manages 6 lists $S(0), S(1), \ldots, S(5)$. $C_0 = S(3) = \{v_5\}$ and $C_1 = S(2) = \{v_3, v_8\}$ because $|C| = 3$.

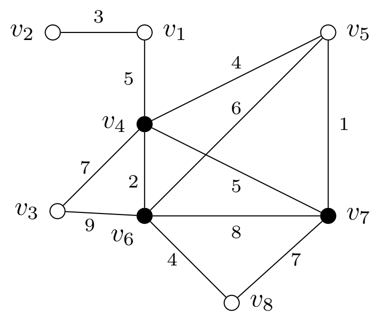
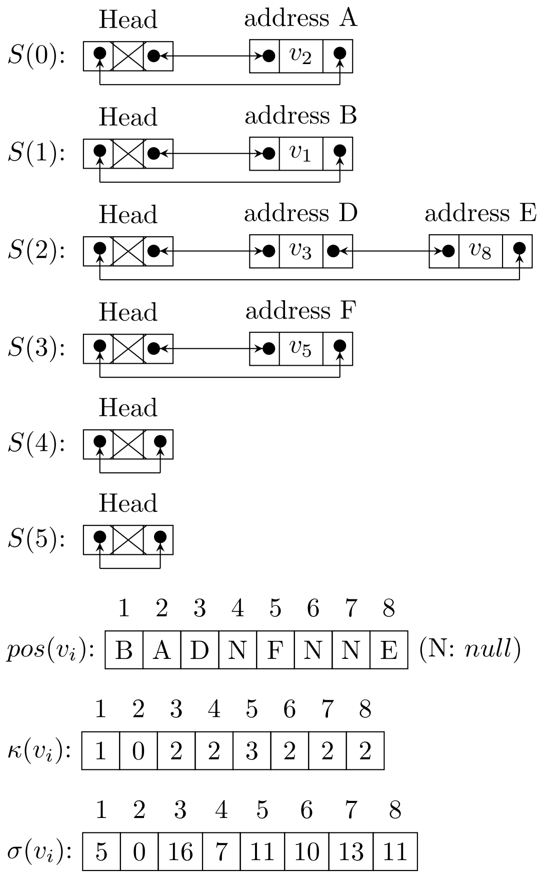


Figure 5.1: A graph G_{ex} (black vertices are in the clique C)



	1	2	3	4	5	6	7	8
$pos(v_i)$:	B	A	D	N	F	N	N	E
$\kappa(v_i)$:	1	0	2	2	3	2	2	2
$\sigma(v_i)$:	5	0	16	7	11	10	13	11

Figure 5.2: Data structures of proposed method L

Proposed method L updates $\kappa(\cdot)$, $S(\cdot)$, $pos(\cdot)$ and $\sigma(\cdot)$ in moving the clique C. Algorithm 15 shows the procedure to add a vertex v to the clique and the procedure to remove a vertex v from the clique. The procedures first update C, and then remove v from $S(|C|)$ in case of adding v to C or add v to $S(|C|)$ in case of removing v from C. Then for each $u \in N(v)$, procedures update $\kappa(u), \sigma(u)$ and move vertices $u \notin C$ to $S(\kappa(u))$. The time complexity of both procedures in Algorithm 15 is $O(|N(v)|)$. The reference to edge weights at lines 7 and 21 is done in constant time since they are in the **for** loops for each elements in adjacency lists. Since each vertex not included in the clique is in appropriate $S(i)$, proposed method L scans less vertices than previous method, and the time complexity of proposed method L is smaller than previous method [21].

Algorithm 15 Proposed method L: updating data structure

```
1: procedure AddToClique($v$)
2:     remove $v$ from $S(|C|)$                          ▷ Using $pos(v)$
3:     $pos(v) \leftarrow null$                           ▷ not in any $S(\cdot)$
4:     add $v$ to $C$
5:     for all $u \in N(v)$ do
6:         $\kappa(u) \leftarrow \kappa(u) + 1$
7:         $\sigma(u) \leftarrow \sigma(u) + w(v, u)$
8:         if $u \notin C$ then
9:             remove $u$ from $S(\kappa(u) - 1)$         ▷ Using $pos(u)$
10:            insert $u$ to $S(\kappa(u))$
11:            $pos(u) \leftarrow$ (address of $u$ in $S(\kappa(u))$)
12:        end if
13:    end for
14: end procedure

15: procedure DropFromClique($v$)
16:    remove $v$ from $C$
17:    add $v$ to $S(|C|)$                                ▷ $\kappa(v) = |C|$
18:    $pos(u) \leftarrow$ (address of $u$ in $S(|C|)$)
19:    for all $u \in N(v)$ do
20:        $\kappa(u) \leftarrow \kappa(u) - 1$
21:        $\sigma(u) \leftarrow \sigma(u) - w(v, u)$
22:        if $u \notin C$ then
23:            remove $u$ from $S(\kappa(u) + 1)$         ▷ Using $pos(u)$
24:            insert $u$ to $S(\kappa(u))$
25:            $pos(u) \leftarrow$ (address of $u$ in $S(\kappa(u))$)
26:        end if
27:    end for
28: end procedure
```

5.3.2 Proposed method M

First, to reduce the time complexity to update data structures, proposed method M constructs non-adjacency lists from adjacency matrix. For each v, non-adjacency list contains the elements of $V \setminus N(v)$. Adding non-adjacency lists does not increase the space complexity of $O(|V|^2)$. For a clique C and each vertex $v \in V$, proposed method M defines $s(v, C)$ as follows :

$$s(v, C) = |C \setminus N(v)|.$$

Hereafter $s(v)$ denotes $s(v, C)$. For $v \notin C$, v is included in C_0 if and only if $s(v) = 0$. For $v \notin C$, v is included in C_1 if and only if $s(v) = 1$. Same as $S(i)$ of proposed method L, proposed method M implements C_0 and C_1 by doubly linked lists and stores the address of each vertex v to an array $pos(v)$. By this implementation, a vertex can be added or removed in constant time. For the graph G_{ex} shown in Figure 5.1 and a clique $C = \{v_4, v_6, v_7\}$, Figure 5.3 shows the data structures of proposed method M. In the initial state, $C = \emptyset$, $s(v) = 0$ holds for all vertices $v \in V$, and all vertices are in C_0. For each $v \in V$, $pos(v)$ contains the address of v in C_0. C_1 is initialized by an empty doubly linked list that has only a head element.

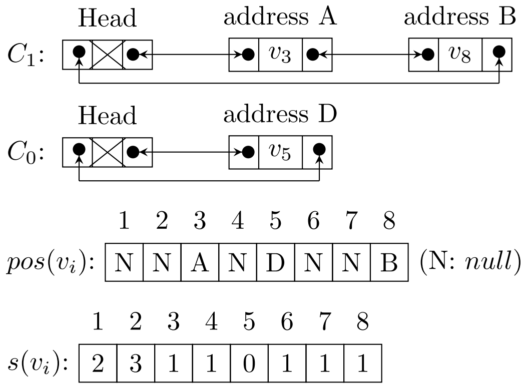


Figure 5.3: Data structures of proposed method M

Proposed method M updates $s(\cdot)$, C_0, C_1 and $pos(\cdot)$ in moving the clique C to a neighborhood. Algorithm 16 shows the procedure to add a vertex v to the clique and the procedure to remove a vertex v from the clique. Using non-adjacency lists, the time complexity of both procedures in Algorithm 16 is $O(|V \setminus N(v)|)$.

Proposed method M calculates clique weights as follows when they are required. For a clique C, the weight of a clique $C \cup \{v\}$ in add-neighborhoods is $w_e(C) + \sum_{s \in C} w(s, v)$. The weight of a clique $(C \setminus \{u\}) \cup \{v\}$ in swap-neighborhood is $w_e(C) - \sum_{t \in C \setminus \{u\}} w(u, t) + \sum_{s \in C \setminus \{u\}} w(s, v)$. The weight of a clique $C \setminus \{u\}$ in drop-neighborhood is $w_e(C) - \sum_{t \in C \setminus \{u\}} w(u, t)$. From the above, the weight of each clique in neighborhoods is calculated in $O(|C|)$.

Algorithm 16 Proposed method M: updating data structure

```
 1: procedure AddToClique($v$)
 2:     remove $v$ from $C_0$                                  ▷ Using $pos(v)$
 3:     $pos(u) \leftarrow null$                               ▷ not in $C_0$ or $C_1$
 4:     $s(v) \leftarrow 1$
 5:     add $v$ to $C$
 6:     for all $u \in V \setminus (N(v) \cup \{v\})$ do
 7:         $s(u) \leftarrow s(u) + 1$
 8:         if $s(u) = 1$ then
 9:             remove $u$ from $C_0$                          ▷ Using $pos(u)$
10:             insert $u$ to $C_1$
11:             $pos(u) \leftarrow$ (address of $u$ in $C_1$)
12:         end if
13:         if $s(u) = 2$ then
14:             remove $u$ from $C_1$                          ▷ Using $pos(u)$
15:             $pos(u) \leftarrow null$                       ▷ not in $C_0$ or $C_1$
16:         end if
17:     end for
18: end procedure

19: procedure DropFromClique($v$)
20:     remove $v$ from $C$
21:     $s(v) \leftarrow 0$
22:     add $v$ to $C_0$
23:     $pos(u) \leftarrow$ (address of $u$ in $C_0$)
24:     for all $u \in V \setminus (N(v) \cup \{v\})$ do
25:         $s(u) \leftarrow s(u) - 1$
26:         if $s(u) = 0$ then
27:             remove $u$ from $C_1$                          ▷ Using $pos(u)$
28:             insert $u$ to $C_0$
29:             $pos(u) \leftarrow$ (address of $u$ in $C_0$)
30:         end if
31:         if $s(u) = 1$ then
32:             insert $u$ to $C_1$
33:             $pos(u) \leftarrow null$                       ▷ not in $C_0$ or $C_1$
34:         end if
35:     end for
36: end procedure
```

5.4 Computer experiments

We implement two proposed data structures and previous data structure [21] by C++. We use PLS [50] for MEWCP to compare these data structures. In addition, we modify the multi neighborhood tabu search (MN/TS) for MWCP [66] to solve MEWCP and use it to compare data structures. MN/TS can be used for MEWCP only by changing the calculation of clique weights, from the sum of vertex weights to the sum of edge weights.

Although the previous data structure [21] is proposed for MWCP, we modify it to calculate edge weights as follows. For each $(u, v) \in E$, previous one [21] proposes to store the element in a hash table with a key calculated by some one-to-one function $f(u, v)$. It can check whether u, v are adjacent by checking whether the key $f(u, v)$ is in the hash table. Although it is not as fast as adjacency matrix because of overheads, the time complexity to check adjacency is $O(1)$ in average. In the experiments in this section, edge weight $w(u, v)$ is stored in the hash table with the key $f(u, v)$.

The CPU is Intel®Core$^{\text{TM}}$i7-6700 3.40 GHz. Memory is 16GB. The OS is Linux 4.4.0. The compiler is g++ 5.4.0 with optimization option O2.

In all experiments, for each instance, we apply each algorithm 10 times changing random seeds. We measure the best solution in 60 sec and the time to reach them. Since the neighborhood management is differ for each data structure, different vertices are picked from C_0 or C_1 by random selection in each compared method. Hence the reached solution maybe different even if the local search algorithm is same.

5.4.1 DIMACS

DIMACS[62] is a set of benchmarks for MCP. Although graphs in original DIMACS are not vertex-weighted or edge-weighted, they are used in experiments for MWCP and MEWCP by giving weights to vertices or edges [66, 50]. In this experiments, we give $(i + j) \bmod 200 + 1$ for each edge (v_i, v_j) as same as [50]. Table 5.2 shows the results for DIMACS. W_{best} is the best clique weight of all random seeds. The value suc is the number of reaches to W_{best} in all random seeds. The value time is the average time (second) of reaches to W_{best}. The symbol $< \epsilon$ means the algorithm reaches to W_{best} in shorter time than 0.01 sec that is possible minimum measurement time on the environment. Table 5.2 does not show the instances that all algorithms can reach to W_{best} in very short time for all random seeds. However the total suc shown in the bottom of the table includes them.

Comparing proposed method L and previous method [21] by total suc, we confirm that proposed method L obtains more suc than previous method for many instances. For many instances of same suc, proposed method L reaches W_{best} faster than previous method. This is because of the smaller time complexity of neighborhood update.

Comparing two proposed methods, proposed method M obtains more suc. Since many of DIMACS instances are dense graphs, the time complexity $O(|V \setminus N(v)|)$ of proposed method M is better.

Table 5.2: Experimental results for DIMACS

				MN/TS		MN/TS + Proposed method				PLS		PLS + Proposed method			
				[21]		L		M		[21]		L		M	
Instance	$\|V\|$	d	W_{best}	suc	time	suc	time	suc	time	suc	time	suc	time	suc	time
brock200_1	200	0.745	21230	9	15.27	10	1.99	10	0.42	10	0.05	10	0.03	10	$< \epsilon$
brock200_2	200	0.496	6542	8	13.00	10	5.11	10	1.37	10	0.04	10	0.02	10	$< \epsilon$
brock200_3	200	0.605	10303	9	20.42	10	3.34	10	2.13	10	$< \epsilon$	10	$< \epsilon$	10	$< \epsilon$
brock400_1	400	0.748	35257	0	-	1	21.18	5	25.75	10	3.12	10	2.01	10	0.13
brock400_2	400	0.749	40738	1	50.06	7	21.13	10	9.31	10	0.53	10	0.26	10	0.04
brock400_3	400	0.748	46785	7	23.82	10	2.81	10	0.67	10	0.15	10	0.12	10	0.02
brock400_4	400	0.749	54304	10	15.00	10	1.69	10	0.34	10	0.09	10	0.04	10	$< \epsilon$
brock800_1	800	0.649	25050	10	3.80	10	0.83	10	0.14	10	3.30	10	1.29	10	0.35
brock800_2	800	0.651	27932	0	-	0	-	0	-	9	16.18	9	25.02	10	2.67
brock800_3	800	0.649	30972	0	-	0	-	2	21.44	10	19.24	9	19.27	10	3.07
brock800_4	800	0.650	30950	0	-	1	59.37	0	-	10	7.65	10	6.86	10	0.67
C1000.9	1000	0.900	234013	1	41.79	8	13.96	10	10.79	2	21.62	3	30.94	10	10.38
C2000.5	2000	0.500	14927	6	18.76	9	22.01	10	5.87	4	31.33	4	27.71	8	17.89
C2000.9	2000	0.900	320715	0	-	0	-	1	41.44	0	-	0	-	0	-
C4000.5	4000	0.500	19304	0	-	2	37.09	3	44.13	0	-	0	-	1	43.79
C500.9	500	0.900	164953	10	4.89	10	0.14	10	0.07	10	4.36	10	0.98	10	0.17
c-fat500-10	500	0.374	804000	10	14.80	10	0.01	10	0.21	10	0.05	10	$< \epsilon$	10	$< \epsilon$
c-fat500-2	500	0.073	38350	10	1.14	10	$< \epsilon$	10	0.05	10	$< \epsilon$	10	$< \epsilon$	10	$< \epsilon$
c-fat500-5	500	0.186	205864	10	5.42	10	$< \epsilon$	10	0.10	10	$< \epsilon$	10	$< \epsilon$	10	$< \epsilon$
DSJC1000_5	1000	0.500	12054	10	2.01	10	0.55	10	0.13	10	4.96	10	1.94	10	0.43
gen400_p0.9_55	400	0.900	150981	10	2.80	10	0.09	10	0.07	10	1.26	10	0.41	10	0.10
hamming10-2	1024	0.990	13140816	3	9.48	10	0.07	10	1.11	10	3.74	10	0.03	10	0.04
hamming10-4	1024	0.829	83280	0	-	3	20.12	8	36.64	0	-	7	34.32	8	26.02
johnson32-2-4	496	0.879	16330	10	6.25	10	0.51	10	0.13	0	-	0	-	0	-
keller5	776	0.752	38901	8	22.42	10	3.83	10	0.62	6	22.28	7	22.09	10	5.78
keller6	3361	0.818	178189	0	-	1	29.26	0	-	0	-	0	-	0	-
MANN_a27	378	0.990	802575	0	-	0	-	0	-	0	-	0	-	2	20.92
MANN_a45	1035	0.996	5874190	0	-	0	-	0	-	0	-	0	-	1	20.44
MANN_a81	3321	0.999	59893215	0	-	0	-	0	-	0	-	1	28.63	0	-
p_hat1500-2	1500	0.506	211069	10	8.57	10	0.31	10	0.25	10	1.02	10	0.49	10	0.09
p_hat1500-3	1500	0.754	441998	5	33.06	10	1.69	10	1.80	10	3.04	10	0.67	10	0.05
san1000	1000	0.502	10661	2	43.96	2	12.94	9	19.00	4	39.28	7	24.45	10	13.06
san200_0.7_2	200	0.700	15073	10	6.53	10	0.37	10	0.15	10	0.35	10	0.04	10	0.01
san400_0.5_1	400	0.500	7442	9	23.24	10	5.50	10	1.78	10	1.11	10	0.18	10	0.08
san400_0.7_1	400	0.700	77719	10	19.65	10	2.16	10	0.52	10	4.37	10	0.76	10	0.11
san400_0.7_2	400	0.700	44155	10	2.09	10	0.15	10	0.08	10	1.66	10	0.32	10	0.04
san400_0.7_3	400	0.700	24727	10	1.28	10	0.22	10	0.03	10	0.53	10	0.17	10	0.04
san400_0.9_1	400	0.900	496874	10	3.72	10	0.08	10	0.04	10	1.03	10	0.07	10	0.02
total suc				638		684		708		695		707		730	

5.4.2 BHOSLIB

Same as DIMACS, BHOSLIB[67] is a set of benchmarks for MCP. Although graphs in original BHOSLIB are not vertex-weighted or edge-weighted too, we give $(i + j) \bmod 200 + 1$ for each edge (v_i, v_j) as same as [50]. Table 5.3 shows the results for BHOSLIB.

By comparing suc, we confirm proposed methods are better than previous one. Proposed method M obtains more suc than proposed method L. Although PLS obtains more suc on DIMACS and MN/TS obtains more suc on BHOSLIB, these tendencies of data structures are same as DIMACS.

Table 5.3: Experimental results for BHOSLIB

				MN/TS		MN/TS + Proposed method				PLS		PLS + Proposed method			
				[21]		L		M		[21]		L		M	
Instance	$\lvert V \rvert$	d	W_{best}	suc	time	suc	time	suc	time	suc	time	suc	time	suc	time
frb30-15-1	450	0.824	44069	10	0.59	10	0.10	10	0.03	10	0.44	10	0.55	10	0.05
frb30-15-2	450	0.823	44078	10	0.53	10	0.08	10	0.02	10	0.19	10	0.15	10	0.03
frb30-15-3	450	0.824	43414	6	30.14	10	5.21	10	1.54	10	9.67	10	7.92	10	1.73
frb30-15-4	450	0.823	43884	10	2.89	10	0.36	10	0.13	10	0.20	10	0.13	10	0.01
frb30-15-5	450	0.824	43675	9	35.72	10	4.44	10	1.01	10	9.58	10	5.53	10	0.79
frb35-17-1	595	0.842	59629	1	11.07	10	20.98	10	6.83	4	22.67	5	34.47	10	5.90
frb35-17-2	595	0.842	59973	8	30.54	10	3.00	10	1.53	10	15.86	10	14.31	10	0.92
frb35-17-3	595	0.842	60357	10	8.17	10	1.43	10	0.23	10	3.61	10	4.35	10	0.45
frb35-17-4	595	0.842	59653	2	24.67	7	28.16	10	7.64	6	39.71	4	10.31	10	11.55
frb35-17-5	595	0.841	60749	10	5.00	10	0.51	10	0.13	10	6.35	10	2.89	10	0.37
frb40-19-1	760	0.857	79800	4	31.94	10	9.43	10	1.90	2	17.94	3	26.58	10	9.61
frb40-19-2	760	0.857	79004	1	52.71	6	37.81	10	13.30	2	30.09	2	22.10	9	13.28
frb40-19-3	760	0.858	79457	6	15.04	10	2.35	10	1.20	2	9.54	9	9.71	10	4.03
frb40-19-4	760	0.856	79247	5	19.22	10	9.44	10	5.30	2	32.39	4	26.24	10	15.68
frb40-19-5	760	0.856	79223	2	34.64	3	29.76	8	30.48	0	-	0	-	4	34.85
frb45-21-1	945	0.867	99802	0	-	2	11.43	6	25.86	0	-	1	55.92	7	33.30
frb45-21-2	945	0.869	99838	0	-	0	-	2	40.94	0	-	3	19.20	1	47.50
frb45-21-3	945	0.869	100282	1	14.48	5	38.82	6	34.29	0	-	0	-	3	37.87
frb45-21-4	945	0.869	101182	0	-	7	23.96	10	19.61	0	-	2	28.82	7	33.00
frb45-21-5	945	0.869	99614	0	-	2	18.46	2	12.74	0	-	1	56.26	3	20.26
frb50-23-1	1150	0.879	122931	0	-	0	-	2	11.02	0	-	0	-	0	-
frb50-23-2	1150	0.878	123674	0	-	0	-	1	39.73	0	-	0	-	0	-
frb50-23-3	1150	0.877	123494	0	-	0	-	1	51.31	0	-	0	-	0	-
frb50-23-4	1150	0.879	123298	1	41.14	4	27.33	5	15.37	0	-	1	21.98	7	21.06
frb50-23-5	1150	0.879	122846	1	2.84	0	-	6	28.27	0	-	0	-	1	45.33
frb53-24-1	1272	0.883	135675	0	-	0	-	1	32.05	0	-	0	-	0	-
frb53-24-2	1272	0.883	140162	0	-	0	-	2	18.76	0	-	0	-	0	-
frb53-24-3	1272	0.884	140122	0	-	0	-	3	48.52	0	-	0	-	2	30.81
frb53-24-4	1272	0.883	138758	0	-	2	32.76	0	-	0	-	0	-	0	-
frb53-24-5	1272	0.883	139614	0	-	1	11.67	0	-	0	-	0	-	0	-
frb56-25-1	1400	0.888	151823	0	-	1	9.41	1	30.04	0	-	0	-	1	17.36
frb56-25-2	1400	0.888	151377	0	-	0	-	1	34.34	0	-	0	-	0	-
frb56-25-3	1400	0.888	150509	0	-	0	-	1	57.62	0	-	0	-	0	-
frb56-25-4	1400	0.888	156615	0	-	0	-	3	16.69	0	-	0	-	0	-
frb56-25-5	1400	0.888	155630	0	-	0	-	0	-	0	-	0	-	1	32.29
frb59-26-1	1534	0.892	170472	0	-	0	-	1	52.20	0	-	0	-	0	-
frb59-26-2	1534	0.893	170232	0	-	1	38.01	1	54.10	0	-	0	-	0	-
frb59-26-3	1534	0.893	168343	0	-	0	-	0	-	1	59.36	0	-	0	-
frb59-26-4	1534	0.892	168397	0	-	0	-	1	21.40	0	-	0	-	0	-
frb59-26-5	1534	0.893	172795	0	-	0	-	2	14.70	0	-	1	20.07	0	-
	total suc			97		161		206		99		116		176	

5.4.3 higgs-twitter data set

We compare data structures on higgs-twitter data set of large sparse graphs. Graphs of this data set are constructed from twitter user activity about higgs boson published at [37].

In this data set, each vertex corresponds to an user. Activities from user v to user u is represented by a directed edge (v, u), and its weight is the count of activities. In the experiments, we use the following three graphs :

retweet-network A graph constructed from retweets.

reply-network A graph constructed from replies.

mention-network A graph constructed from mentions.

These are huge graphs whose number of vertices is several tens or hundreds of thousand. To use them as benchmarks of MEWCP, we obtain undirected simple edge-weighted graphs as follows :

- Remove loops.
- For two vertices u, v that have directed edges between them, replace all of those edges by one undirected edge (u, v).
- The weight $w(u, v)$ of an undirected edge (u, v) is the total weight of directed edges between u and v of the original directed graph.

Since all of activities indicate communications between users, we can find a group of strong ties by extracting a maximum edge-weight clique from processed graphs. We regard that processing described above is acceptable for extracting such groups.

Memory usage of data structures

For the processed higgs-twitter graphs, we estimate memory usage of adjacency matrix and adjacency lists as follows. We assume that one word length is 4byte based on the CPU used in experiments. For adjacency lists, the number of elements in the lists is $2|E|$, and each element is a tuple of 8 byte that contains a connected vertex and edge weight. Therefore we estimate the memory usage of adjacency lists at $16|E|$ byte. For an adjacency matrix, the number of elements in the matrix is $|V|^2$, and each element is edge weight of 4 byte. Hence we estimate the memory usage of an adjacency matrix at $4|V|^2$ byte. Table 5.4 shows the memory usage estimation.

Table 5.4: Memory usage estimation for higgs-twitter graphs

graph	$\|V\|$	$\|E\|$	adjacency matrix	adjacency lists
higgs-reply	38683	29552	5.57GB	0.45MB
higgs-mention	115684	140421	49.85GB	2.14MB
higgs-retweet	256491	327374	245.07GB	5.00MB

Table 5.4 shows that the adjacency matrix requires huge memory. On the other hand, adjacency lists can represent these graphs with much less memory usage. Hence we compare only methods for adjacency lists for higgs-twitter graphs.

Results for higgs-twitter graphs

Table 5.5 shows the results for higgs-twitter graphs. From the results, we confirm that proposed method L reaches W_{best} in shorter time than previous method [21].

Table 5.5: Experimental results for higgs-twitter graphs

				MN/TS				PLS			
				Previous [21]		Proposed L		Previous [21]		Proposed L	
	$\|V\|$	$\|E\|$	W_{best}	suc	time	suc	time	suc	time	suc	time
higgs-reply	38683	29552	28	9	25.09	10	0.76	10	1.46	10	0.13
higgs-mention	115684	140421	430	10	0.08	10	0.01	10	0.13	10	$< \epsilon$
higgs-retweet	256491	327374	101	10	0.20	10	0.03	10	0.44	10	0.02

5.5 Conclusion

We proposed two data structures for neighborhood management on local search algorithms for MEWCP. A technique to calculate clique weights of MEWCP efficiently is also proposed. With two local search algorithms, we compared all combinations of data structures and local search algorithms in computer experiments.

We use DIMACS, BHOSLIB and higgs-twitter data set as benchmarks. From the experimental result, we confirmed that proposed methods are better than previous one. For dense graphs of DIMACS and BHOSLIB, proposed method M with adjacency matrix and non-adjacency lists is better. On the other hand, for large sparse graphs of higgs-twitter data set, proposed method L with adjacency lists is better. From the results, we confirmed that local search algorithms works efficiently using one of proposed data structures properly by memory capacity or graph property such as number of vertices or edge density.

For MEWCP, MCP and MWCP, proposed methods can be used to manage a set of candidate vertices to add to clique in heuristics such as [17]. The performance analysis of proposed data structures on such algorithms is a future work.

Chapter 6

Conclusion

In this paper, we proposed two branch-and-bound based exact algorithms for MWCP, a mathematical programming technique for MEWCP, a branch-and-bound based exact algorithm for MEWCP, two greedy algorithms for MWVCP and two data structures used in local search algorithms for MEWCP.

In the chapter 2, we proposed two branch-and-bound algorithms for MWCP. Proposed algorithm VCTable uses vertex coloring to calculate upper bounds. VCTable calculates upper bounds of vertex coloring before branch-and-bound and store them to upper bound tables. In branch-and-bound, VCTable calculates upper bounds in short time by using upper bound tables. The other algorithm OTClique is also proposed. Before branch-and-bound, OTClique calculates the weights of optimal solutions for a lot of small subgraphs and stores the values to optimal tables. Upper bounds are calculated using optimal tables. By some computer experiments for some benchmarks, we confirmed that they are better than previous algorithms.

In the chapter 3, we proposed a mathematical programming formulation technique and a branch-and-bound based exact algorithm for MEWCP. For the MIP formulation of MEWCP, we proposed vertex renumbering technique. By the vertex renumbering, the range of each variables becomes smaller, and the value of some variables are fixed. This improves the performance of mathematical programming solvers. In addition, we proposed a branch-and-bound algorithm called EWCLIQUE. EWCLIQUE decomposes edge weights of each subproblem into three components. It calculates an upper bound for each component, and uses the sum of them as an upper bound for the subproblem. By some experiments, we compared algorithms and confirmed that EWCLIQUE is faster than others in all of benchmarks.

In the chapter 4, two heuristic algorithms are proposed. They are based on a simple linear time greedy algorithm which removes vertices from the set of all vertices. Since the base algorithm is very simple, it is faster than previous approximate algorithms. However it finds worse solutions. Proposed algorithms create a vertex list and then construct multiple solutions from the list. To construct multiple solutions, one adopts the rotating technique and the other adopts the branching technique. They output the best solution among them. Comparing to approximation algorithms for MWVCP, proposed algorithms can find better solutions in shorter time.

In the chapter 5, two data structures used in local search algorithms for MEWCP are proposed. Local search algorithms start from a solution and continuously move it to one of neighborhoods. The neighborhoods are scanned in each movement. Hence the computation time to calculate neighborhood for each solution is important. One of proposed algorithms can be used with graphs represented by adjacency lists. The other is for graphs represented by adjacency matrix. With local search algorithms for MEWCP, we compare proposed methods and previous one by computer experiments. Proposed methods show better performance than previous one.

Acknowledgements

The completion of this thesis was made by the contributions, knowledge, kindness, and support of many individuals. First of all, I would like to express my sincere gratitude to my supervisor Associate Professor Kazuaki Yamaguchi, and Professor Sumio Masuda at Kobe University, for their patient guidance, support, encouragement, and excellent advice throughout my study period.

I would like to appreciate Professor Masahiro Numa and Professor Tsutomu Terada at Kobe University. They took time to review my doctoral thesis and gave me valuable comments and advices.

I am thankful to Associate Professor Toshiki Saitoh at Kyushu Institute of Technology. When he taught at Kobe University, he gave me lots of advices and motivated me to study more.

I am grateful to Mr. Ryota Ishihara. He helps me on some of computer experiments, and provides some of benchmark instances used in experiments.

I would like to thank for the help and support for the teachers and staffs at Departments of Electrical and Electronic Engineering of Kobe University.

Finally, I would like to thank my family for their sincere supports and warm encouragement.

Bibliography

[1] Bahram Alidaee, Fred Glover, Gary Kochenberger, and Haibo Wang. Solving the maximum edge weight clique problem via unconstrained quadratic programming. *European Journal of Operational Research*, 181(2):592–597, 2007.

[2] Luitpold Babel. A fast algorithm for the maximum weight clique problem. *Computing*, 52(1):31–38, 1994.

[3] KC Bahadur, Tatsuya Akutsu, Etsuji Tomita, and Tomokazu Seki. Protein side-chain packing problem: a maximum edge-weight clique algorithmic approach. In *Proceedings of the second conference on Asia-Pacific bioinformatics-Volume 29*, pages 191–200. Australian Computer Society, Inc., 2004.

[4] Egon Balas and Jue 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):209–221, 1991.

[5] Egon Balas and Jue Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15(5):397–412, 1996.

[6] Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986.

[7] Reuven Bar-Yehuda and Shimon Even. A linear-time approximation algorithm for the weighted vertex cover problem. *Journal of Algorithms*, 2(2):198–203, 1981.

[8] Reuven Bar-Yehuda and Shimon Even. A local-ratio theorem for approximating the weighted vertex cover problem. *North-Holland Mathematics Studies*, 109:27–45, 1985.

[9] Mikhail Batsyn, Boris Goldengorin, Evgeny Maslov, and Panos M Pardalos. Improvements to mcs algorithm for the maximum clique problem. *Journal of Combinatorial Optimization*, 27(2):397–416, 2014.

[10] Una Benlic and Jin-Kao Hao. Breakout local search for maximum clique problems. *Computers & Operations Research*, 40(1):192–206, 2013.

[11] Galina T Bogdanova, Andries E Brouwer, Stoian N Kapralov, and Patric RJ Östergård. Error-correcting codes over an alphabet of four elements. *Designs, Codes and Cryptography*, 23(3):333–342, 2001.

[12] Salim Bouamama, Christian Blum, and Abdellah Boukerram. A population-based iterated greedy algorithm for the minimum weight vertex cover problem. *Applied Soft Computing*, 12(6):1632–1639, 2012.

[13] JB Brown, KC Dukka Bahadur, Etsuji Tomita, and Tatsuya Akutsu. Multiple methods for protein side chain packing using maximum weight cliques. *Genome Informatics*, 17(1):3–12, 2006.

[14] Kevin Leyton Brown. Combinatorial auction test suite (CATS), 2000. `http://www.cs.ubc.ca/~kevinlb/CATS/`.

[15] Randy Carraghan and Panos M Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9(6):375–382, 1990.

[16] Luís Cavique. A scalable algorithm for the market basket analysis. *Journal of Retailing and Consumer Services*, 14(6):400–407, 2007.

[17] Carmine Cerrone, Raffaele Cerulli, and Bruce Golden. Carousel greedy: a generalized greedy algorithm with applications in optimization. *Computers & Operations Research*, 85:97–112, 2017.

[18] Kenneth L Clarkson. A modification of the greedy algorithm for vertex cover. *Information Processing Letters*, 16(1):23–25, 1983.

[19] Steve R Corman et al. Pajek datasets: Reuters terror news network. `http://vlado.fmf.uni-lj.si/pub/networks/data/CRA/terror.htm`.

[20] Steven R Corman, Timothy Kuhn, Robert D McPhee, and Kevin J Dooley. Studying complex discursive systems. *Human communication research*, 28(2):157–206, 2002.

[21] Yi Fan, Chengqian Li, Zongjie Ma, Lian Wen, Abdul Sattar, and Kaile Su. Local search for maximum vertex weight clique on large sparse graphs with efficient data structures. In *Advances in Artificial Intelligence: 29th Australasian Joint Conference*, pages 255–267. Springer, 2016.

[22] Zhiwen Fang, Chu-Min Li, and Ke Xu. An exact algorithm based on maxsat reasoning for the maximum weight clique problem. *Journal of Artificial Intelligence Research*, 55:799–833, 2016.

[23] Michael R Garey and David S Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, New York, 1979.

[24] Luis Gouveia and Pedro Martins. Solving the maximum edge-weight clique problem in sparse graphs with compact formulations. *EURO Journal on Computational Optimization*, 3(1):1–30, 2015.

[25] Johan Håstad. Clique is hard to approximate withinn1- ε. *Acta Mathematica*, 182(1):105–142, 1999.

[26] Radu Horaud and Thomas Skordas. Stereo correspondence through feature grouping and maximal cliques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1168–1180, 1989.

[27] Yan Jin and Jin-Kao Hao. General swap-based multiple neighborhood tabu search for the maximum independent set problem. *Engineering Applications of Artificial Intelligence*, 37:20–33, 2015.

[28] Raka Jovanovic and Milan Tuba. An ant colony optimization algorithm with improved pheromone correction strategy for the minimum weight vertex cover problem. *Applied Soft Computing*, 11(8):5360–5366, 2011.

[29] George Karakostas. A better approximation ratio for the vertex cover problem. *ACM Transactions on Algorithms (TALG)*, 5(4):41, 2009.

[30] Richard Manning Karp. Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer, 1972.

[31] D Bahadur KC, Tatsuya Akutsu, Etsuji Tomita, Tomokazu Seki, and Asao Fujiyama. Point matching under non-uniform distortions and protein side chain packing based on efficient maximum clique algorithms. *Genome Informatics*, 13:143–152, 2002.

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

[33] Deniss 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*, pages 202–208. Citeseer, 2004.

[34] Deniss Kumlander. Improving the maximum-weight clique algorithm for the dense graphs. In *Proceedings of the 10th WSEAS International Conference on COMPUTERS*, pages 938–943, 2006.

[35] Deniss 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):32–49, 2006.

[36] Deniss 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*, pages 165–174, 2008.

[37] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[38] Ruizhi Li, Shuli Hu, Haochen Zhang, and Minghao Yin. An efficient local search framework for the minimum weighted vertex cover problem. *Information Sciences*, 2016.

[39] Rafael Martí, Micael Gallego, and Abraham Duarte. A branch and bound algorithm for the maximum diversity problem. *European Journal of Operational Research*, 200(1):36–44, 2010.

[40] Pedro Martins. Cliques with maximum/minimum edge neighborhood and neighborhood density. *Computers & Operations Research*, 39(3):594–608, 2012.

[41] George L Nemhauser and Leslie E Trotter Jr. Vertex packings: structural properties and algorithms. *Mathematical Programming*, 8(1):232–248, 1975.

[42] Patric RJ Östergård. Cliquer homepage. `http://users.tkk.fi/pat/cliquer.html`.

[43] Patric RJ Östergård. A new algorithm for the maximum-weight clique problem. *Nordic Journal of Computing*, 8(4):424–436, 2001.

[44] Patric RJ Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120(1):197–207, 2002.

[45] Panos M Pardalos and Nisha Desai. An algorithm for finding a maximum weighted independent set in an arbitrary graph. *International Journal of Computer Mathematics*, 38(3-4):163–175, 1991.

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

[47] Panos M Pardalos and Jue Xue. The maximum clique problem. *Journal of global Optimization*, 4(3):301–328, 1994.

[48] Leonard Pitt. A simple probabilistic approximation algorithm for vertex cover. Technical report, Yale University, June 1985.

[49] Wayne Pullan. Phased local search for the maximum clique problem. *Journal of Combinatorial Optimization*, 12(3):303–323, 2006.

[50] Wayne Pullan. Approximating the maximum vertex/edge weighted clique using local search. *Journal of Heuristics*, 14(2):117–134, 2008.

[51] Steffen Rebennack, Gerhard Reinelt, and Panos M Pardalos. A tutorial on branch and cut algorithms for the maximum stable set problem. *International Transactions in Operational Research*, 19(1-2):161–199, 2012.

[52] Fabrizio Rossi and Stefano Smriglio. A branch-and-cut algorithm for the maximum cardinality stable set problem. *Operations Research Letters*, 28(2):63–74, 2001.

[53] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers & Operations Research*, 38(2):571–581, 2011.

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

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

[56] Michael M Sørensen. New facets and a branch-and-cut algorithm for the weighted clique problem. *European Journal of Operational Research*, 154(1):57–70, 2004.

[57] S. Sorour and S. Valaee. Minimum broadcast decoding delay for generalized instantly decodable network coding. In *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*, pages 1–5, Dec 2010.

[58] Satoshi Taoka, Daisuke Takafuji, and Toshimasa Watanabe. Computing-based performance analysis of approximation algorithms for the minimum weight vertex cover problem of graphs. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 96(6):1331–1339, 2013.

[59] Etosuji Tomita, Yoichi Sutani, Takanori Higashi, Shinya Takahashi, and Mitsuo Wakatsuki. A simple and faster branch-and-bound algorithm for finding a maximum clique. In *WALCOM: Algorithms and computation*, pages 191–203. Springer, 2010.

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

[61] Etsuji Tomita and Tomokazu Seki. An efficient branch-and-bound algorithm for finding a maximum clique. In *Discrete Mathematics and Theoretical Computer Science*, pages 278–289. Springer, 2003.

[62] Michael Trick, Vavsek Chvatal, Bill Cook, David Johnson, Cathy McGeoch, Bob Tarjan, et al. DIMACS implementation challenges. `http://dimacs.rutgers.edu/Challenges/`.

[63] Yiyuan Wang, Shaowei Cai, and Minghao Yin. Two efficient local search algorithms for maximum weight clique problem. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 805–811, 2016.

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

[65] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[66] Qinghua Wu, Jin-Kao Hao, and Fred Glover. Multi-neighborhood tabu search for the maximum weight clique problem. *Annals of Operations Research*, 196(1):611–634, 2012.

[67] Ke Xu. BHOSLIB: Benchmarks with hidden optimum solutions for graph problems (maximum clique, maximum independent set, minimum vertex cover and vertex coloring). `http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm`.

[68] Kazuaki Yamaguchi and Sumio Masuda. A new exact algorithm for the maximum weight clique problem. In *23rd International Conference on Circuits/Systems, Computers and Communictions (ITC-CSCC'08)*, pages 317–320, 2008.

[69] Taoqing Zhou, Zhipeng Lü, Yang Wang, Junwen Ding, and Bo Peng. Multi-start iterated tabu search for the minimum weight vertex cover problem. *Journal of Combinatorial Optimization*, pages 1–17, 2015.

List of publications

Journals (Refereed)

1. Satoshi Shimizu, Kazuaki Yamaguchi, Toshiki Saitoh and Sumio Masuda,
   "Fast maximum weight clique extraction algorithm: optimal tables for branch-and-bound,"
   Discrete Applied Mathematics, Vol.223, pp.120-134, 2017.

2. 清水 悟司，山口 一章，増田 澄男，
   "数理計画問題による最大辺重みクリーク問題の定式化，"
   電子情報通信学会論文誌，Vol.100-A No.8, pp.313-315, 2017.

3. 清水 悟司，石原 諒大，山口 一章，増田 澄男，
   "最大辺重みクリーク問題に対する局所探索法のためのデータ構造，"
   情報処理学会論文誌，Vol.59 No.7, pp.1415-1424, 2018.

Conferences (Refereed)

1. Satoshi Shimizu, Kazuaki Yamaguchi and Sumio Masuda,
   "A branch-and-bound based exact algorithm for the maximum edge-weight clique problem,"
   Computational Science/Intelligence & Applied Informatics, Accepted.

2. Satoshi Shimizu, Kazuaki Yamaguchi, Toshiki Saitoh and Sumio Masuda,
   "A fast heuristic for the minimum weight vertex cover problem,"
   Proceedings of the IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), pp.341-345, 2016.

3. Satoshi Shimizu, Kazuaki Yamaguchi, Toshiki Saitoh and Sumio Masuda,
   "Optimal Table Method for Finding the Maximum Weight Clique,"
   Proceedings of the 13th International Conference on Applied Computer Science (ACS), pp.84-90, 2013.

4. Satoshi Shimizu, Kazuaki Yamaguchi, Toshiki Saitoh and Sumio Masuda,
   "Some improvements on Kumlander's maximum weight clique extraction algorithm,"
   Proceedings of the International Conference on Electrical, Computer, Electronics and Communication Engineering (ICECECE), pp.307-311, 2012.

Workshops (No peer review)

1. 清水 悟司，山口 一章，増田 澄男，
   "分枝限定法による最大辺重みクリーク抽出法，"
   情報処理学会 第160回アルゴリズム研究会，2016.

2. 清水 悟司，石原 諒大，山口 一章，増田 澄男，
   "最大辺重みクリーク問題に対する局所探索法の実験的評価，"
   平成28年度情報処理学会関西支部 支部大会，2016.

3. 清水 悟司，山口 一章，斎藤 寿樹，増田 澄男，
“最小重み頂点被覆問題に対する高速な発見的手法の提案，”
電子情報通信学会コンピュテーション研究会，2016.

4. 清水 悟司，
“最大重みクリークに対する最適解テーブル法，”
ERATO湊離散構造処理系プロジェクト「2013年度 初夏のワークショップ」.

5. 清水 悟司，山口 一章，斎藤 寿樹，増田 澄男，
“動的計画法を用いた上界計算法による最大重みクリーク抽出アルゴリズムの提案，”
電子情報通信学会コンピュテーション研究会，2013.

6. 清水悟司，森中 諒太，山口 一章，増田 澄男，
“ある最大重みクリーク抽出法における頂点集合の効率的な実装方法の提案，”
平成24年度情報処理学会関西支部 支部大会，2012.

7. 森中 諒太，清水悟司，山口 一章，増田 澄男，
“最大重みクリーク抽出法における分枝順序の検討，”
平成24年度情報処理学会関西支部 支部大会，2012.

Doctor Thesis, Kobe University ”Research on Advanced Human Navigation Systems”, 80 pages.
Submitted on July, 13th, 2018
The date of publication is printed in cover of repository version published in Kobe University Repository Kernel.