TCS-TR-A-14-76

TCS Technical Report

Frontier-based Search for Enumerating All Constrained Subgraphs with Compressed Representation

by

Jun Kawahara, Takeru Inoue, Hiroaki Iwashita and Shin-ichi Minato

Division of Computer Science

Report Series A

September 30, 2014

Hokkaido University
Graduate School of
Information Science and Technology

Email: minato@ist.hokudai.ac.jp

Phone: +81-011-706-7682

Fax: +81-011-706-7682

Frontier-based Search for Enumerating All Constrained Subgraphs with Compressed Representation

Jun Kawahara[1] Takeru Inoue[2] Hiroaki Iwashita[3]
Shin-ichi Minato[4,5]

[1] Graduate School of Information Science, Nara Institute of Science and Technology
jkawahara@is.naist.jp
[2] NTT Network Innovation Laboratories, NTT Corporation
inoue.takeru@lab.ntt.co.jp
[3] Knowledge Platforms Laboratory, Fujitsu Laboratories Limited
iwashita.hiroak@jp.fujitsu.com
[4] Graduate School of Information Science and Technology, Hokkaido University, Japan
minato@ist.hokudai.ac.jp
[5] ERATO MINATO Project, Japan Science and Technology Agency, Japan

Abstract

For the subgraph enumeration problem, there have been proposed very efficient algorithms whose time complexities are far less than the number of subgraphs. Although the number of subgraphs can increase exponentially with the graph size, they exploit compressed representations to maintain enumerated subgraphs compactly so as to reduce the time complexity. However, they are designed for enumerating a specific type of subgraphs only, e.g., paths or trees. In this paper, we propose a novel algorithm framework, called frontier-based search, which generalizes these specific algorithms without losing their efficiency. We believe that our frontier-based search will be used to resolve various practical problems that include complicated subgraph enumeration.

1 Introduction

Enumerating all the subgraphs of a graph (or, more generally, object) satisfying given conditions is one of the most fundamental problems in combinatorics and graph theory. A few decades ago, several well-known algorithms were proposed for subgraph enumeration problems [16, 19, 1]. Since these traditional enumeration algorithms output subgraphs one by one, their time complexities are proportional to the output size. The output size, however, can increase exponentially with the graph size. Recently, several algorithms whose time complexities are significantly smaller than the output size have been proposed [18, 12, 9]; these algorithms exploit compressed representations such as binary decision diagrams (BDDs) [3] in order to output the subgraphs in a compressed manner.

Figure 1 is an example of *s*-*t* path enumeration that shows the key idea on the use of compressed representation. Since traditional naïve algorithms output *s*-*t* paths one by one (Figure 1 (b)), the time complexity of the algorithms depends on the number of *s*-*t* paths, which exponentially increases as the size of the graph grows. Regarding a path as a set of edges constituting the path, we find some paths share a common subset of edges; e.g., the first and second paths share edges 12, 23, 36, 56, and 89 in Figure 1 (b). We merge such common edges if possible, as shown in Figure 1 (c); in this compressed diagram, each path can be retrieved walking from the left to the right. As shown in the figure, compressed representation requires only 30 edges to represent all the paths (Figure 1 (c)), while naïve enumeration includes 64 edges with many duplications (Figure 1 (b)). Recent enumeration algorithms directly construct the compressed representation from the left to the right, without detouring the naïve representation. If the compressed representation is much smaller than the naïve representation, the time complexity can be reduced a lot.

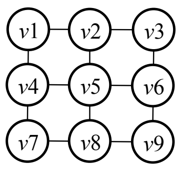
Subgraph enumeration algorithms can play a key role in practical problems including logic puzzles [21], reliability evaluation [4], and network optimization [7]. Unfortunately, subgraph enumeration algorithms used in these problems are dedicated for their own problems, and so we have to develop yet another graph enumeration algorithm to deal with every new problem. In order to resolve several practical problems, enumeration algorithms must be so flexible to search for variety of complicated subgraphs, such as "forests such that each tree contains exactly one of the specified vertices and the number of its edges is in the specified range." Our key contribution is generalization of algorithms that find common parts between subgraphs so as to construct the compressed representation for various types of constrained subgraphs, like Figure 1 (c).

In this paper, we propose a subgraph enumeration algorithm, called *frontier-based search*, which enables us to enumerate various types of constrained subgraphs. Frontier-based search supports several primitive constraints including a set of connected vertices, degree of each vertex, acyclic or not, and the number of edges used. Combinations of these primitive constraints allow us to specify various types of complicated subgraphs. To recap, the frontier-based search has the following features:

1. Unlike recent specific enumeration algorithms, frontier-based search supports several primitive constraints, and allows us to specify various types of subgraphs by their combinations.

2. Unlike traditional naïve enumeration algorithms, frontier-based search inherits the excellent properties of recent subgraph enumeration algorithms, such as great compression ability and efficient search strategy without backtracks (the worst case complexity is not excellent theoretically, but frontier-based search performs well practically if subgraphs have some structure).

(a) 3x3 grid graph



(b) Enumerating all paths naively

```
 1: 12    23    45 47 36 56    78    89
 2: 12    23          36 56 58       89
 3: 12    23          36          69
 4: 12       25 45 47          78    89
 5: 12       25          56       69
 6: 12       25             58       89
 7:    14 23 25 45    36          69
 8:    14 23 25    47 36    58 78 69
 9:    14       45       56       69
10:    14       45          58       89
11:    14          47    56 58 78 69
12:    14          47          78    89
```

(c) Enumeration with compressed representation

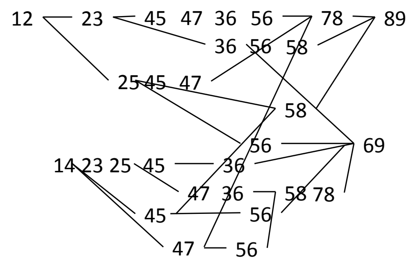


Figure 1: Enumerating all twelve v_1-v_9 paths on 3×3 grid graph naïvely and using compressed representation. A path is represented by a set of edges; e.g., the first path "12, 23, 45, 47, 36, 56, 78, and 89" means path of $\{\{v_1, v_2\}, \{v_2, v_3\}, \{v_4, v_5\}, \{v_4, v_7\}, \{v_3, v_6\}, \{v_5, v_6\}, \{v_7, v_8\}, \text{ and } \{v_8, v_9\}\}$.

The organization of the paper is as follows. In Section 2, we analyze the Simpath enumeration algorithm [9, exercise 225 in Section 7.1.4], in order to clarify requirements to frontier-based search. In Section 3, we propose the frontier-based search that enumerates various subgraphs, and also present some examples of subgraphs which can be treated. Section 4 shows several experimental results. Related work is shown in Section 5.

2 Analyzing Simpath

Simpath is an enumeration algorithm designed for all the s-t paths on a given graph. It is proposed by Knuth, and the implementation is highly-optimized. This section unravels the secret of efficiency, and summarizes requirements to our frontier-based search.

2.1 Outline

Simpath enumerates all the s-t paths on a given undirected graph; this graph is called the *universe* and is denoted by $G_{\mathrm{uni}} = (V_{\mathrm{uni}}, E_{\mathrm{uni}})$, where V_{uni} is a set of vertices and E_{uni} is a set of edges. Let $n = |V_{\mathrm{uni}}|$, $m = |E_{\mathrm{uni}}|$, $V_{\mathrm{uni}} = \{v_1, \ldots, v_n\}$, and $E_{\mathrm{uni}} = \{e_1, \ldots, e_m\}$. An *$s$-$t$ path* is a subgraph $(\{v'_1, v'_2, \ldots, v'_{l+1}\}, \{e'_1, e'_2, \ldots, e'_l\})$ of G_{uni} such that $v'_1 = s$, $v'_{l+1} = t$, $v'_i \in V_{\mathrm{uni}}$ for all i, $e'_i = \{v'_i, v'_{i+1}\} \in E_{\mathrm{uni}}$ for all i, and $v_i \neq v_j$ for all i and j with $i \neq j$.

Simpath constructs the directed acyclic graph which is the compressed representation to compactly represent all the s-t paths on G_{uni} (see Figure 2). We call it the *decision diagram* (DD) [1] and denote by $\mathcal{D} = (N, A)$. We describe the property which $\mathcal{D}$ has to possess, and then we outline the construction process of $\mathcal{D}$. To avoid confusing G_{uni} with $\mathcal{D}$, we address elements in V_{uni} and E_{uni} as a *vertex* and an *edge*, respectively, whereas ones in N and A as a *node* and an *arc*, respectively. Also, we call a path on $\mathcal{D}$ a *route*.

$\mathcal{D}$ has a *root* node, denoted by n_{root}, which is labeled e_1 and which has just two arcs, called a 0-*arc* and a 1-*arc* (this property is different with Figure 1 (c), which has only a single type of arcs). The 0-arc and the 1-arc mean not selecting and selecting e_1, respectively. Each arc points at a node labeled e_2, or a special node, called a *terminal node*, which is labeled **0** or **1**. Similarly, for each $i = 1, \ldots, m-1$, a node labeled e_i has a 0-arc and a 1-arc, each of which points at a node labeled e_{i+1} or a terminal node. Both arcs of any node labeled e_m have to point at **0** or **1**. For each route from n_{root} to **1** in $\mathcal{D}$, labels of nodes whose 1-arc is included in the route have to constitute an s-t path on G_{uni}.

Although arbitrary binary tree with appropriate terminals satisfies the requirements of the DD described above (e.g., Figure 2 (b)), the number of nodes of the binary tree for a graph with m edges can increase exponentially as m grows. If there exist two nodes labeled e_i whose subtrees are identical, we need not to maintain both of them. More precisely, for nodes n_1 and n_2 in a DD, n_1 and n_2 are *identical* if $R(n_1) = R(n_2)$ holds, where $R(\hat{n})$ is the set of edge sets corresponding to all the route from $\hat{n}$ to **1**. In this case, the operation of keeping n_1, discarding n_2 and modifying the destination of each arc which points at n_2 into n_1 does not change the set of s-t paths which the DD represents. We call this operation *node sharing.* The DD in Figure 2 (c) is obtained by repeatedly applying node sharing to the binary tree in Figure 2 (b) as far as possible. Observe that it satisfies the requirements of the DD.

We directly construct the DD such as Figure 2 (c) in a top-down and breadth-first manner without making the broad binary tree such as Figure 2 (b). We first create n_{root}, a 0-arc and a node labeled e_2, and connect n_{root} with the node using the 0-arc. We also create a 1-arc and a node labeled e_2, and connect n_{root} with the node using the 1-arc. Similarly, for $i = 1, \ldots, m-1$, we create nodes labeled e_{i+1}

[1]The DD which we will construct is known as a zero-suppressed ordered binary decision diagram (ZDD) [11]. Note that it may not be reduced.

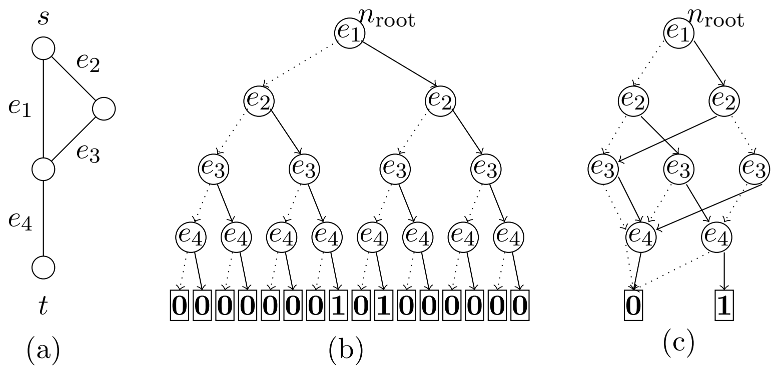


Figure 2: Example of the DDs representing all the s-t paths on a graph. (a) An example of an universe. There are two s-t paths, $\{e_1, e_4\}$ and $\{e_2, e_3, e_4\}$ on the graph. (b) The broad binary tree representing the set of the paths. 0-arcs and 1-arcs are depicted as dotted and solid lines, respectively. (c) The DD obtained from (b) by repeating node sharing. Observe that by selecting either 1-arc, 0-arc, 0-arc and 1-arc, or 0-arc, 1-arc, 1-arc and 1-arc in this order, we reach **1** from n_{root}.

after creating all the nodes labeled e_i. When making a node labeled e_{i+1}, we check whether there is an existing node which is identical with the node or not, and we perform node sharing if found. Note that deciding whether nodes n_1 and n_2 are identical has to be carried out without constructing children of n_1 and n_2, which will be discussed in Section 2.2. This is the key technique that makes SIMPATH very efficient.

The pseudocode of SIMPATH can be written as shown in Algorithm 1. In Line 5-13, for $x = 0, 1$, the x-arc of $\hat{n}$ is made to point at a new node, an existing node or a terminal, as follows. In Line 5, CHECKTERMINAL$(\hat{n}, i, x)$ decides whether the destination of the x-arc of $\hat{n}$ is a terminal $(\mathbf{0}, \mathbf{1})$ or not. In Line 8, UPDATEINFO(n', i, x) updates the information of n', as will be described later. Line 9 searches an existing node n'' which is identical to n'. If it is found, n' is discarded and let the x-arc of $\hat{n}$ point at n''. In Line 12, the new node n' is added to N_{i+1}, where N_j denotes the set of nodes labeled with e_j. In Algorithm 1, we put processes depending on s-t paths into UPDATEINFO and CHECKTERMINAL functions. By rearranging SIMPATH in this way, we find that other enumeration algorithms designed for different graph types also can be written in the same manner with different specifications of UPDATEINFO and CHECKTERMINAL.

2.2 Finding Child Nodes

In this subsection, we analyze the two functions, UPDATEINFO and CHECKTERMINAL, which are specialized to handle s-t paths. In order to find child nodes without constructing the subtree, each node maintains some values with respect to which edges

Algorithm 1: ConstructDD

```
1  $N_1 \leftarrow \{n_{\mathrm{root}}\}$. $N_i \leftarrow \emptyset$ for $i = 2, \ldots, m+1$.
2  for $i \leftarrow 1$ to $m$ do
3      foreach $\hat{n} \in N_i$ do
4          foreach $x \in \{0, 1\}$ do                         // process for the 0/1-arc
5              $n' \leftarrow$ CheckTerminal($\hat{n}, i, x$)              // Returns 0, 1, or nil.
6              if $n' =$ nil then                              // $n'$ is neither 0 nor 1.
7                  Copy $\hat{n}$ to $n'$.
8                  UpdateInfo($n', i, x$)
9                  if there exists $n'' \in N_i$ s.t. $n''$ is identical to $n'$ then
10                     $n' \leftarrow n''$
11                 else
12                     $N_{i+1} \leftarrow N_{i+1} \cup \{n'\}$
13             Create the $x$-arc of $\hat{n}$ and make it point at $n'$.
```

have been selected from the root. We design the values as follows. For node $\hat{n}$, let E be the set of edges corresponding to one route from n_{root} to $\hat{n}$ and let G be the subgraph induced by E. We store the degree of v in G to the variable $\hat{n}.\mathtt{deg}[v]$, and also store the component identifier of v to the variable $\hat{n}.\mathtt{comp}[v]$, where the component identifier is an integer in $\{1, 2, \ldots, n\}$ such that $\hat{n}.\mathtt{comp}[v] = \hat{n}.\mathtt{comp}[w]$ if and only if v and w are in the same connected component in G (in the original Simpath, the degree and component identifier are implicitly encoded into a single integer named "mate", but we describe them explicitly for generalization).

We maintain these values just for verticies incident to both a processed edge and an unprocessed one. We call a set of these vertices *frontier* (Figure 3). More precisely, for each $i = 1, \ldots, m-1$, we define the i-th frontier F_i by $F_i = \left(\bigcup_{j=1,\ldots,i} e_j\right) \cap \left(\bigcup_{j=i+1,\ldots,m} e_j\right)$[2]. We also define $F_0 = F_m = \emptyset$. For a node labeled with e_i ($i = 1, \ldots, m$), we maintain $\hat{n}.\mathtt{deg}[v]$ and $\hat{n}.\mathtt{comp}[v]$ for only each vertex v in F_{i-1}. Two nodes n_1 and n_2 labeled e_i are considered as identical if $n_1.\mathtt{deg}[v] = n_2.\mathtt{deg}[v]$ and $n_1.\mathtt{comp}[v] = n_2.\mathtt{comp}[v]$ for every $v \in F_{i-1}$ [21].

The pseudocode of UpdateInfo(n', i, x) is shown as Algorithm 2. It updates $\hat{n}.\mathtt{deg}$ and $\hat{n}.\mathtt{comp}$. In Line 15, we also update the component identifier in comp for all nodes of e_i as follows. For $i \in \{1, \ldots, n\}$, let $c(i)$ be the minimum vertex index in the ith connected component (i.e., its component identifier is i). If the ith component does not exist, $c(i) = \infty$. Then, we update them so that $c(1), c(2), \ldots, c(n)$ are sorted in the ascending order. Although any node does not have the values of $\hat{n}.\mathtt{deg}[v]$ and $\hat{n}.\mathtt{comp}[v]$ if v is not in the frontier, UpdateInfo works because it references the values of $\hat{n}.\mathtt{deg}$ and $\hat{n}.\mathtt{comp}$ only on the frontier.

CheckTerminal($\hat{n}, i, x$) is shown as Algorithm 3. When the destination of

[2]In the graph theory, an edge is defined by the set of vertices to which the edge is incident, i.e., e_i is equivalent to, for example, $\{v, w\}$. Thus, $\bigcup_{j=1,\ldots,i} e_j$ represents the set of vertices to which at least one of $e_1, \ldots, e_i$ is incident.

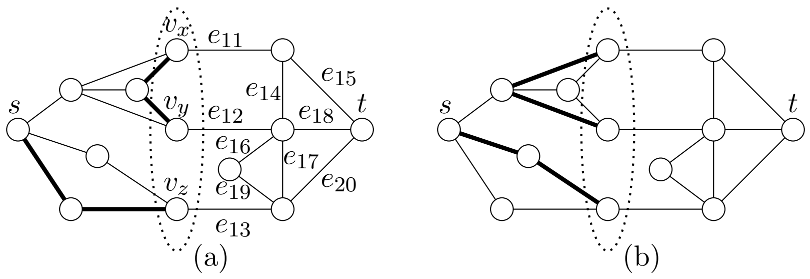


Figure 3: Two identical states (a) and (b), in which edges in the left-hand side of dotted oval have been processed and bold edges have been determined to use.

an arc is being decided (in Line 5-13 in Algorithm 1), if any s-t path cannot be constructed, the destination should be **0**. Considering subgraph G, once one of the following conditions is satisfied, G cannot be an s-t path even if adding some edges to G: (i-1) the degree of s or t in G exceeds 1, (i-2) a branch of the s-t path occurs (i.e., the degree of a vertex except for s and t exceeds 2) in G, and (ii) a cycle occurs in G. In Line 2-4, we check whether (ii) holds or not. We also check whether (i-1) and (i-2) hold or not in Line 7-11. If a vertex u is an endpoint of e_i and $u \notin F_i$, u is no longer referred after processing e_i by the definition of the frontier. Therefore, the degree of u is fixed. In Line 12-16, we check the degree constraint of s-t path, i.e., whether the degrees of s and t are exactly one or not, and whether those of the other vertices are zero or two, or not. If we are processing the last edge e_m and (i-1), (i-2) and (ii) do not hold, an s-t path has been completed (Line 18).

The UpdateInfo and CheckTerminal functions are performed locally, that is, they do not need to traverse the DD. Our frontier-based search must inherit this property even if it will be generalized. The computation time of them is proportional to the number of vertices in the frontier. That of finding an identical node is also proportional to the frontier size if nodes are stored using a hash. Therefore, Line 5-13 in Algorithm 1 can be executed in $O(w)$ time, where $w = \max_i |F_i|$. We denote the number of nodes in the constructed DD $\mathcal{D}$ by $|\mathcal{D}|$. Hence, the computation time of Simpath is $O(w \cdot |\mathcal{D}|)$, which may be exponential in n theoretically. However, our experiments in Sec. 4 show that $\mathcal{D}$ often represents $O(a^{|\mathcal{D}|})$ s-t paths for some constant a, which indicates that the computation time of Simpath may be significantly smaller than the number of s-t paths.

2.3 Requirements to Frontier-based Search

At the last of this section, we summarize the requirements to our frontier-based search.

1. The frontier-based search can follow Algorithm 1 even if it is generalized,

Algorithm 2: UpdateInfo($\hat{n}, i, x$) for Simpath

```
1  Let $e_i = \{v, w\}$.
2  foreach $u \in \{v, w\}$ such that $u \notin F_{i-1}$ do      // $u$ is entering the frontier.
3      $\hat{n}$.deg[$u$] $\leftarrow 0$
4      $\hat{n}$.comp[$u$] $\leftarrow j$ such that $u = v_j$   // The initial component identifier is
           the index of $u$.
5  if $x = 1$ then
       // Increment the degrees of vertices $v$ and $w$.
6      $\hat{n}$.deg[$v$] $\leftarrow \hat{n}$.deg[$v$] $+ 1$
7      $\hat{n}$.deg[$w$] $\leftarrow \hat{n}$.deg[$w$] $+ 1$
       // The two components are connected.
8      $c_{\min} \leftarrow \min\{\hat{n}$.comp[$v$]$, \hat{n}$.comp[$w$]$\}$
9      $c_{\max} \leftarrow \max\{\hat{n}$.comp[$v$]$, \hat{n}$.comp[$w$]$\}$
10     foreach $u \in F_i$ do
11         if $\hat{n}$.comp[$u$] $= c_{\max}$ then
12             $\hat{n}$.comp[$u$] $\leftarrow c_{\min}$
13 foreach $u \in \{v, w\}$ such that $u \notin F_i$ do           // $u$ is leaving the frontier.
14     Forget $\hat{n}$.deg[$u$] and $\hat{n}$.comp[$u$].
15 Renumber $\hat{n}$.comp so that $c(1), c(2), \ldots, c(n)$ are sorted in the ascending order,
   where $c(i)$ is the minimum vertex $v'$ satisfying $\hat{n}$.comp[$v'$] $= i$.
```

since we find that it is a common framework seen in several enumeration algorithms using compressed representation. It is enough to generalize UpdateInfo and CheckTerminal functions

2. UpdateInfo and CheckTerminal functions have to perform their operations locally without traversing DDs. Because this is the key property that makes the enumeration algorithms efficient.

3 Frontier-based Search for Various Subgraphs

3.1 Definition

The main reason why Simpath can quickly construct a DD is to efficiently decide identicalness of nodes using the values `deg` and `comp` stored in nodes. It is important that an *s*-*t* path is a graph characterized only by the degree of each vertex and whether having a cycle or not, and that these properties can be locally checked during the edge selection process.

Although Simpath is designed for enumerating *s*-*t* paths, the property of *s*-*t* path appears only in the CheckTerminal and UpdateInfo functions. Therefore, by changing the behavior of these functions we can construct DDs which represent various subgraphs such as spanning trees and matchings as long as they perform locally. We focus on subgraphs of G_{uni} induced by some edge set, that is,

Algorithm 3: CHECKTERMINAL($\hat{n}, i, x$) for SIMPATH

```
Let $e_i = \{v, w\}$.
if $x = 1$ then
    if $\hat{n}$.comp[$v$] = $\hat{n}$.comp[$w$] then   // $v$ and $w$ belong to the same connected component.
        return 0                                      // A cycle occurs (check (ii)).
Copy $\hat{n}$ to $n'$.
UPDATEINFO($n', i, x$).
foreach $u \in \{v, w\}$ do
    if ($u = s$ or $u = t$) and $n'$.deg[$u$] $> 1$ then                  // Check (i-1).
        return 0
    else if ($u \neq s$ and $u \neq t$) and $n'$.deg[$u$] $> 2$ then      // Check (i-2).
        return 0
foreach $u \in \{v, w\}$ such that $u \notin F_i$ do        // $u$ is leaving the frontier.
    if ($u = s$ or $u = t$) and $n'$.deg[$u$] $\neq 1$ then               // Check (i-1).
        return 0
    else if ($u \neq s$ and $u \neq t$) and $n'$.deg[$u$] $\neq 0$ and $n'$.deg[$u$] $\neq 2$ then   // Check (i-2).
        return 0
if $i = m$ then          // The processing of the last edge has been done.
    return 1             // An $s$–$t$ path has been completed.
return nil               // Not terminal
```

subgraphs G such that $G = (\mathrm{dom}(E), E)$ with an edge set $E \subseteq E_{\mathrm{uni}}$, where for a set of pairs of vertices X, $\mathrm{dom}(X) = \bigcup_{\{x,y\} \in X} \{x, y\}$.

Using `deg` and `comp`, the DD representing subgraphs G of G_{uni} specified by the following properties can be constructed as same as shown in Sec. 2:

(i) the degrees of each vertex in G,

(ii) whether G has a cycle or not.

We make a node $\hat{n}$ maintain the value $\hat{n}$.`cycle` which represents whether graphs correspond to $\hat{n}$ have a cycle. If $\hat{n}$.`cycle` $= 1$, the graphs have a cycle; otherwise (if $\hat{n}$.`cycle` $= 0$) not.

In addition to the above, we can treat two properties related to connected components using `comp`. One is whether for each given pair of vertices (v, w), v and w belong to the same component or not. To decide it, for a node $\hat{n}$, we store the set of vertices which is included in the connected component whose number is x into $\hat{n}$.`vset`$[x]$. Note that we store the set of only vertices that we would like to pay attention to into $\hat{n}$.`vset`$[x]$. By introducing `vset`, we can specify the following property:

(iii) whether for each specified pair of vertices (v, w), v and w belong to the same component or not.

The other is the number of connected components. Line 12 in Algorithm 3 is a conditional branch for deciding whether u is no longer incident to any unprocessed edge. Then, if the connected component identifier which u belongs to is different from that of any vertex in the frontier F_i, there is no chance in which it is connected with vertices in the frontier after this moment. That is, the connected component is decided. For a node $\hat{n}$, let $\hat{n}$.`cc` be the number of connected component. By introducing `cc`, we can specify the following property:

(iv) the number of connected components which G has.

Furthermore, by making a node $\hat{n}$ maintain the number of edges $\hat{n}$.`noe`, we can specify the following:

(v) the number of edges which G has.

To describe the above conditions, we define some notation for $G = (\mathrm{dom}(E), E)$. We define by $d_G(v)$ the degree of v in G. If $v \in V_{\mathrm{uni}} \setminus \mathrm{dom}(E)$, we define $d_G(v) = 0$. Let c_G be the number of connected components in G except for isolated vertices, and q_G be the 0/1-variable that represents G having a cycle or not. That is, if G has a cycle, $q_G = 1$; otherwise, $q_G = 0$. The subscript of G is omitted when it is clear from the context.

We introduce the notation D, P, S, C, Q, T to formally describe the condition of (i) through (v). For every vertex v, $D(v)$ denotes the set of possible values of $d_G(v)$. C and Q denote the sets of possible values of c_G and q_G, respectively. Note that $C \subseteq \{0, 1, \ldots, n\}$. P denotes the set of vertex pairs each of which has to be included in the same component. S denotes the set of vertex pairs each of which must not be included in the same component. T denotes the set of possible values of the number of edges in G. By using the above notation, we define $\mathcal{G}(D, P, S, C, Q, T)$ by

$$\begin{aligned}\mathcal{G}(D, P, S, C, Q, T) \quad = \quad & \{G = (\mathrm{dom}(E), E) \mid E \subseteq E_{\mathrm{uni}},\ d_G(v) \in D(v) \text{ for each } v, \\ & c_G \in C,\ q_G \in Q,\ v \text{ and } w \text{ are included in the same} \\ & \text{component for each } \{v, w\} \in P,\ x \text{ and } y \text{ are included} \\ & \text{in distinct components for each } \{x, y\} \in S,\ |E| \in T\}.\end{aligned}$$

Let us describe our algorithm, called *frontier-based search*, to construct the DD representing $\mathcal{G}(D, P, S, C, Q, T)$. The framework of the method is similar to SIMPATH. In particular, the pseudocode of the algorithm is the same as Algorithm 1 except for the behavior of the UPDATEINFO and CHECKTERMINAL functions. The UPDATEINFO and CHECKTERMINAL functions for frontier-based search are shown in Algorithm 4 and 5, respectively. The details are described as the comments in the pseudocode.

If some of D, P, S, C, Q, T are specific values, some variables in nodes are not needed. Specifically, if for all $v \in V_{\mathrm{uni}}$, $D(v) = \{0, 1, \ldots, n\}$, then we need not to remember $\hat{n}.\mathtt{deg}$. If $Q = \{0, 1\}$, $P = S = \emptyset$ and $C = \{0, 1, \ldots, n\}$, then we need not to remember $\hat{n}.\mathtt{comp}$. If $P = S = \emptyset$, then we need not to remember $\hat{n}.\mathtt{vset}$. If $C = \{0, 1, \ldots, n\}$, then we need not to remember $\hat{n}.\mathtt{cc}$. If $0 \in Q$ (i.e., we allow an acyclic graph), we need not to remember $\hat{n}.\mathtt{cycle}$. If $T = \{1, \ldots, m\}$, we need not to remember $\hat{n}.\mathtt{noe}$.

The generalized versions of UpdateInfo and CheckTerminal are still performed locally. Line 5 and 21 in Algorithm 5 need $O(|S|)$ and $O(|P|)$ computation time by a straightforward implementation, respectively. However, we can reduce it to $O(|F_i|)$ by storing pairs of the connected component identifiers $\{\mathrm{comp}[v], \mathrm{comp}[w]\}$ for each pair in $\{v, w\} \in P \cup S$ into each node. Hence, the computation time of frontier-based search is still $O(w \cdot |\mathcal{D}|)$. Note that the number of nodes in $\mathcal{D}$ constructed by frontier-based search may be larger than that of nodes constructed by Simpath because more variables are stored in nodes and hence distinct nodes increase.

3.2 Several examples

In this section, we show that frontier-based search can deal with a various kind of subgraphs by specifying D, P, S, C, Q and T in the previous section. We describe several examples of subgraphs and how to obtain its DD by giving D, P, S, C, Q and T.

s-t path and Hamiltonian s-t path

Although the DD representing all s-t paths can be constructed by Simpath described in Sec. 2, we construct it also by frontier-based search shown in Sec. 3.

If D, P, S, C, Q, T are given as follows, $\mathcal{G}(D, P, S, C, Q, T)$ corresponds to all the s-t paths on G_{uni}, where s and t are vertices on G_{uni}:

- $D(s) = D(t) = \{1\}$, $D(v) = \{0, 2\}$ for any $v \in V_{\mathrm{uni}} \setminus \{s, t\}$,
- $P = \{\{s, t\}\}$, $S = \emptyset$,
- $C = \{1, \ldots, n\}$,
- $Q = \{0\}$,
- $T = \{1, \ldots, m\}$.

This means that the degrees of s and t have to be one, that of all the vertices except for s and t have to be zero or two, q_G has to be zero (i.e., there is no cycle),

Algorithm 4: UPDATEINFO$(\hat{n}, i, x)$ for frontier-based search

```
1  Let $e_i = \{v, w\}$.
2  foreach $u \in \{v, w\}$ such that $u \notin F_{i-1}$ do      // $u$ is entering the frontier.
3      $\hat{n}$.deg$[u] \leftarrow 0$
4      $\hat{n}$.comp$[u] \leftarrow j$ such that $u = v_j$    // The initial component identifier is
       the index of $u$.
5      if $u \in \mathrm{dom}(P) \cup \mathrm{dom}(S)$ then
6          $\hat{n}$.vset$[j] \leftarrow \{u\}$ such that $u = v_j$
7  if $x = 1$ then
8      $\hat{n}$.noe $\leftarrow \hat{n}$.noe $+ 1$                 // Increment the number of edges
       // Increment the degrees of vertices $v$ and $w$.
9      $\hat{n}$.deg$[v] \leftarrow \hat{n}$.deg$[v] + 1$
10     $\hat{n}$.deg$[w] \leftarrow \hat{n}$.deg$[w] + 1$
11     if $\hat{n}$.comp$[v] = \hat{n}$.comp$[w]$ then               // A cycle is detected.
12         $\hat{n}$.cycle $\leftarrow$ true
13     else
           // The two components are connected.
14         $c_{\min} \leftarrow \min\{\hat{n}$.comp$[v], \hat{n}$.comp$[w]\}$
15         $c_{\max} \leftarrow \max\{\hat{n}$.comp$[v], \hat{n}$.comp$[w]\}$
16         foreach $u \in F_i$ do          // $u$ is included in the next frontier.
17             if $\hat{n}$.comp$[u] = c_{\max}$ then
18                 $\hat{n}$.comp$[u] \leftarrow c_{\min}$
19         $\hat{n}$.vset$[c_{\min}] \leftarrow \hat{n}$.vset$[c_{\min}] \cup \hat{n}$.vset$[c_{\max}]$
20 foreach $u \in \{v, w\}$ such that $u \notin F_i$ do       // $u$ is leaving the frontier.
21     $f \leftarrow$ false
22     foreach $z \in F_i$ do
23         if $\hat{n}$.comp$[u] = \hat{n}$.comp$[z]$ then
24             $f \leftarrow$ true
25     if $f =$ false then     // There is no vertex which belongs to the same
       component as $u$ in the next frontier.
26         $\hat{n}$.cc $\leftarrow \hat{n}$.cc $+ 1$
27     Forget $\hat{n}$.deg$[u]$ and $\hat{n}$.comp$[u]$.
28 Renumber $\hat{n}$.comp so that $c(1), c(2), \ldots, c(n)$ are sorted in the ascending order,
   where $c(i)$ is the minimum vertex $v'$ satisfying $\hat{n}$.comp$[v'] = i$.
```

and s and t have to belong to the same connected component. c_G and the number of edges can take arbitrary values, which means that the nodes in DD need not maintain `cc` and `noe`. The variable `cycle` is not also needed because $1 \notin Q$. Each node maintains only variables `deg` and `comp`.

A Hamiltonian s-t path is an s-t path which includes all the vertices on G_{uni}. Only the degree constraints of the vertices except for s and t are different from the case of s-t path. That is,

- $D(s) = D(t) = \{1\}$, $D(v) = \{2\}$ for any $v \in V_{\mathrm{uni}} \setminus \{s, t\}$.

Algorithm 5: CheckTerminal($\hat{n}, i, x$) for frontier-based search

```
1  Let $e_i = \{v, w\}$.
2  if $x = 1$ then
3      if $1 \notin Q$ and $\hat{n}$.comp[$v$] = $\hat{n}$.comp[$w$] then  // A cycle is forbidden and $v$ and $w$ belongs to the same connected component.
4          return 0
5      if There exists $\{x, y\} \in S$ such that $x \in \hat{n}$.vset[$\hat{n}$.comp[$v$]] and $y \in \hat{n}$.vset[$\hat{n}$.comp[$w$]] then  // $x$ and $y$ will be in the same component.
6          return 0
7  Copy $\hat{n}$ to $n'$.
8  UpdateInfo($n', i, x$).
9  if $n'$.deg[$v$] > $\max\{j \mid j \in D(v)\}$ or $n'$.deg[$w$] > $\max\{j \mid j \in D(w)\}$ then  // The degree of $v$ or $w$ exceeds all of the possible values.
10     return 0
11 foreach $u \in \{v, w\}$ such that $u \notin F_i$ do  // $u$ is leaving the frontier.
12     if $n'$.deg[$u$] $\notin D(u)$ then  // The degree condition is not satisfied.
13         return 0
14     $f \leftarrow$ false
15     foreach $z \in F_i$ do
16         if $n'$.comp[$u$] = $n'$.comp[$z$] then
17             $f \leftarrow$ true
18     if $f$ = false then  // There is no vertex which belongs to the same component as $u$ in the next frontier.
19         if $n'$.cc > $\max\{j \mid j \in C\}$ then  // The number of connected components exceeds all of the possible values.
20             return 0
21         if There exist $\{v', v''\} \in P$ such that $v' \in n'$.vset[$n'$.comp[$u$]] and $v'' \notin n'$.vset[$n'$.comp[$u$]] then  // $u$ and $v'$ are in the same component, but $v'$ and $v''$ are in distinct components.
22             return 0
23 if $i = m$ then  // The processing of the last edge has been done.
24     if $n'$.cc $\notin C$ then  // The component condition is not satisfied.
25         return 0
26     else if $n'$.noe $\notin T$ then  // Check the number of edges.
27         return 0
28     else if $0 \notin Q$ and $n'$.cycle = false then
29         return 0
30     else
31         return 1  // All the constraints are satisfied.
32 return nil  // It is not the terminal.
```

Trees and forests

A spanning tree on G_{uni} is a subgraph of G_{uni} such that all the vertices belong to the same connected component and it has no cycle. If D, P, S, C, Q, T are given as follows, $\mathcal{G}(D, P, S, C, Q, T)$ corresponds to all the spanning trees on G_{uni}:

- $D(v) = \{1, \ldots, n\}$ for any $v \in V_{\text{uni}}$,
- $P = \{\{v, w\} \mid v, w \in V_{\text{uni}}, v \neq w\}$, $S = \emptyset$,
- $C = \{1\}$,
- $Q = \{0\}$,
- $T = \{1, \ldots, m\}$.

This means that for any v, the degree of v is at least one, every pair of vertices belongs to the same connected component, $c(G)$ has to be one, and $q(G)$ has to be zero.

Let $r_1, \ldots, r_\ell$ be vertices on G_{uni}. A rooted spanning forest whose roots are $r_1, \ldots, r_\ell$ is a forest of G_{uni} such that it has no cycle, the number of connected components is m, each connected component has exactly one of $r_1, \ldots, r_\ell$, and each vertex belongs to one of m components. If D, P, S, C, Q, T are given as follows, $\mathcal{G}(D, P, S, C, Q, T)$ corresponds to all the rooted spanning forests whose roots are $r_1, \ldots, r_\ell$ on G_{uni}:

- $D(v) = \{1, \ldots, n\}$ for any $v \in V_{\text{uni}}$,
- $P = \emptyset$, $S = \{\{r_i, r_j\} \mid 1 \leq i < j \leq \ell\}$,
- $C = \{\ell\}$,
- $Q = \{0\}$,
- $T = \{1, \ldots, m\}$.

Let R be a set of vertices on G_{uni}. An R-Steiner tree is a tree including all the vertices in R. If D, P, S, C, Q, T are given as follows, $\mathcal{G}(D, P, S, C, Q, T)$ corresponds to all the R-Steiner trees on G_{uni}:

- $D(v) = \{0, \ldots, n\}$ for any $v \in V_{\text{uni}}$,
- $P = \{\{r, r'\} \mid r \in R, r' \in R\}$, $S = \emptyset$,
- $C = \{1\}$,
- $Q = \{0\}$,
- $T = \{1, \ldots, m\}$.

Since on the R-Steiner tree, all the vertices in R belong to the same connected component, we specify $P = \{\{r, r'\} \mid r \in R, r' \in R\}$. $C = \{1\}$ means that the number of connected components has to be just one, the Steiner tree itself. Note that since the isolated vertices should be ignored to count the number of connected components in this case, we must modify Algorithm 5 slightly (we omit the detail).

Matching

A matching on G_{uni} is a subgraph such that the number of edges incident to each vertex is at most one. If D, P, S, C, Q, T are given as follows, $\mathcal{G}(D, P, S, C, Q, T)$ corresponds to all the matchings on G_{uni}:

- $D(v) = \{0, 1\}$ for any $v \in V_{\mathrm{uni}}$,
- $P = \emptyset$, $S = \emptyset$,
- $C = \{1, \ldots, n\}$,
- $Q = \{0\}$,
- $T = \{1, \ldots, m\}$.

In addition to these subgraphs, we can obtain the DD representing various subgraphs by giving D, P, C, S, Q, T. More examples are shown in Table 1.

4 Experiments

We show that frontier-based search significantly outperforms traditional naïve algorithms in enumerating spanning trees by numerical experiments. Then, we present that the running time of frontier-based search is almost the same as that of Knuth's SIMPATH algorithm, which is dedicated for s-t path. Some other experiments were carried out to show that frontier-based search can treat many kinds of subgraphs.

We introduce three classes of graphs G_ℓ, M_ℓ and W_ℓ, which is given as the universes. G_ℓ is the $\ell \times \ell$ grid graph. M_ℓ is a graph generated from a real-world map of Nara city by the following way: First, we obtained a map near JR Nara station from OpenStreetMap [15], and converted it to the well-known text format using `osm2wkt` [10]. The well-known text format can be easily converted to the inner format of our program. Then, we cut it so that both its width and height are ℓ meters and JR Nara station are located at the center. We remove all the vertices whose degree is at most two. W_ℓ is a small-world graph generated from a random small-world graph generation model, called the Watts-Strogatz model [20], with parameter $N = \ell$, $K = 4$ and $\beta = 0.2$. We made use of the `connected_watts_strogatz_graph` method in NetworkX library [14], which tries to repeatedly generate a graph until connected one is obtained. The numbers of their vertices and edges are shown in Table 2.

Table 1: Subgraphs

Type	Parameters
s-t path	$D(s) = D(t) = \{1\}$, $D(v) = \{0, 2\}$ for any $v \in V_{\mathrm{uni}} \setminus \{s, t\}$ $P = \{\{s, t\}\}$, $S = \emptyset$, $C = \{1\}$, $Q = \{0\}$, $T = \{1, \ldots, m\}$
Hamiltonian s-t path	$D(s) = D(t) = \{1\}$, $D(v) = \{2\}$ for any $v \in V_{\mathrm{uni}} \setminus \{s, t\}$ $P = \{\{v, w\} \mid v, w \in V_{\mathrm{uni}}, v \neq w\}$, $S = \emptyset$ $C = \{1\}$, $Q = \{0\}$, $T = \{1, \ldots, m\}$
Multiple paths	$D(v) = \{0, 1, 2\}$ for any $v \in V_{\mathrm{uni}}$, $P = \emptyset$, $S = \emptyset$, $C = \{1, \ldots, n\}$, $Q = \{0\}$, $T = \{1, \ldots, m\}$
Single cycle	$D(v) = \{0, 2\}$ for any $v \in V_{\mathrm{uni}}$, $P = \{\{v, w\} \mid v, w \in V_{\mathrm{uni}}, v \neq w\}$, $S = \emptyset$, $C = \{1\}$, $Q = \{1\}$, $T = \{1, \ldots, m\}$
Multiple cycles	$D(v) = \{0, 2\}$ for any $v \in V_{\mathrm{uni}}$, $P = \emptyset$, $S = \emptyset$, $C = \{1, \ldots, n\}$, $Q = \{1\}$, $T = \{1, \ldots, m\}$
Hamiltonian cycle	$D(v) = \{2\}$ for any $v \in V_{\mathrm{uni}}$, $P = \{\{v, w\} \mid v, w \in V_{\mathrm{uni}}, v \neq w\}$, $S = \emptyset$, $C = \{1\}$, $Q = \{1\}$, $T = \{1, \ldots, m\}$
Spanning tree	$D(v) = \{1, \ldots, n\}$ for any $v \in V_{\mathrm{uni}}$, $P = \{\{v, w\} \mid v, w \in V_{\mathrm{uni}}, v \neq w\}$, $S = \emptyset$, $C = \{1\}$, $Q = \{0\}$, $T = \{1, \ldots, m\}$
Rooted spanning forest	$D(v) = \{1, \ldots, n\}$ for any $v \in V_{\mathrm{uni}}$, $P = \emptyset$, $S = \{\{r_i, r_j\} \mid 1 \leq i < j \leq \ell\}$ with roots $r_1, \ldots, r_\ell$, $C = \{1, \ldots, \ell\}$, $Q = \{0\}$, $T = \{1, \ldots, m\}$
R-Steiner tree	$D(v) = \{1, \ldots, n\}$ for any $v \in V_{\mathrm{uni}}$, $P = \{\{r_i, r_j\} \mid r_i, r_j \in P\}$, $Q = \emptyset$ $C = \{1\}$, $Q = \{0\}$, $T = \{1, \ldots, m\}$
Matching	$D(v) = \{0, 1\}$ for any $v \in V_{\mathrm{uni}}$, $P = \emptyset$, $S = \emptyset$, $C = \{1, \ldots, n\}$, $Q = \{0\}$, $T = \{1, \ldots, m\}$
Perfect matching	$D(v) = \{1\}$ for any $v \in V_{\mathrm{uni}}$, $P = \emptyset$, $S = \emptyset$, $C = \{1, \ldots, n\}$, $Q = \{0\}$, $T = \{1, \ldots, m\}$
Edge cover	$D(v) = \{1, \ldots, n\}$ for any $v \in V_{\mathrm{uni}}$, $P = \emptyset$, $S = \emptyset$, $C = \{1, \ldots, n\}$, $Q = \{0, 1\}$, $T = \{1, \ldots, m\}$
k-clique	$D(v) = \{0, k-1\}$ for any $v \in V_{\mathrm{uni}}$, $P = \emptyset$, $S = \emptyset$, $C = \{1\}$, $Q = \{0, 1\}$, $T = \{k(k-1)/2\}$

Table 2: Given universes and the numbers of their vertices and edges.

Graph	# of Vertices	# of Edges
G_ℓ	ℓ^2	$2\ell^2 - 2\ell$
M_{200}	7	12
M_{300}	19	33
M_{400}	27	43
M_{500}	56	89
M_{600}	78	124
M_{700}	100	161
M_{800}	123	199
W_ℓ	ℓ	2ℓ

Table 3: Experiment for spanning tree. Read is the algorithm in [16].

Graph	# of Nodes FBS	FBS Time	Read Time	# of subgraphs
G_2	16	0.00	0.00	4
G_3	270	0.01	0.01	192
G_4	2303	0.01	0.184	100352
G_5	15704	0.02	1082.88	5.57568×10^8
G_6	97265	0.03	T/O	3.25655×10^{13}
G_7	571576	1.06	T/O	1.98724×10^{19}
G_8	3251489	10.00	T/O	1.26231×10^{26}
G_9	18093572	67.06	T/O	8.32663×10^{33}
G_{10}	99101715	435.35	T/O	5.69432×10^{42}
M_{200}	629	0.01	0.00	392
M_{300}	4789	0.02	27.84	1.76180×10^7
M_{400}	37290	0.02	493.24	2.33645×10^8
M_{500}	343769770	1096.74	T/O	2.43278×10^{18}
W_{20}	44476	0.02	1775.10	1.10039×10^9
W_{30}	1198102	2.05	T/O	5.56887×10^{13}
W_{40}	777244510	2186.42	T/O	2.12187×10^{18}

We implemented frontier-based search (Algorithm 1, 4 and 5) in C++, and compiled it using `g++` compiler with `-O3` optimize option. The source code is available in `https://github.com/junkawahara/frontier-experiment`. All the experiments were performed by a 2.3GHz Xeon E5-2630 computer with 128GB memory.

We first compared frontier-based search with a conventional back-track algorithm [16] for enumerating all the spanning trees on a given universe, whose computation time is proportional to the number of solutions. The experimental results for the cases where the universes are G_ℓ, M_ℓ and W_ℓ for some ℓ's are shown in Table 3. The back-track algorithm could not count the number of the spanning trees of G_6, which is about 3×10^{13}, whereas our algorithm succeeded in counting that of G_9, which is about 8×10^{33}.

Secondly, we compared frontier-based search with SIMPATH for s-t path with the universes G_ℓ, M_ℓ and W_ℓ. SIMPATH is written in C language and compiled

Table 4: Number of ZDD nodes, the running time and the number of solutions of frontier-based search (FBS) and Simpath for s-t path. Note that both algorithms can calculate the exact number of solutions although it is shown as the floating point in the table.

Graph	# of Nodes FBS	FBS Time	# of Nodes Simpath	Simpath Time	# of subgraphs
G_8	78640	0.00	64894	0.07	7.89360×10^{11}
G_9	274740	0.01	224147	0.16	3.26660×10^{15}
G_{10}	937672	0.02	757715	0.44	4.10442×10^{19}
G_{11}	3143949	0.07	2519890	1.38	1.56876×10^{24}
G_{12}	10396094	2.03	8274264	4.59	1.82413×10^{29}
G_{13}	33997610	8.03	26894642	15.31	6.45280×10^{34}
G_{14}	110180128	29.08	86698793	51.36	6.94507×10^{40}
G_{15}	354414745	106.72	277581570	167.73	2.27450×10^{47}
M_{400}	685	0.00	2422	0.06	8288
M_{500}	2548855	0.07	1439642	2.06	7.17999×10^{7}
M_{600}	6913381	4.09	14321860	10.03	2.01115×10^{11}
M_{700}	626023359	406.05	851773581	528.07	8.30693×10^{14}
W_{40}	4257554	1.01	2463005	1.43	1.26506×10^{9}
W_{50}	42777545	16.01	40631960	24.49	1.60426×10^{11}
W_{60}	373660835	166.06	185212263	113.84	4.21309×10^{13}
W_{70}	2152647207	1495.73	2080507887	1394.71	6.76810×10^{15}

using `gcc` compiler with `-O3` option. In the case of G_ℓ, we let the start of the path s be a corner of the grid, in the case of M_ℓ, we let s be the node corresponding to the JR Nara station, and in the case of W_ℓ, we let s be a node randomly chosen. We let the goal of the path t be the farthest node from s (if there are the two or more farthest nodes, one of them is randomly chosen). The edges in each universe are ordered in a breadth-first manner from s.

Table 4 shows the running times of both algorithms. This result shows that our algorithm is comparable with Simpath; surprisingly, it is slightly faster for other than W_{60} and W_{70}. We can conclude that frontier-based search has been generalized from Knuth's algorithm without greatly losing the efficiency. Note that since frontier-based search directly stores the values as `deg` and `comp` into nodes, while Simpath encodes such values as "mate," the memory requirement of frontier-based search during the computation is larger than that of Simpath.

Finally, we show the results for Hamiltonian s-t path, rooted spanning forest, R-Steiner tree and matching. In the case of rooted spanning forest and R-Steiner tree, we let the roots of G_ℓ be the four corner vertices on the grid, and those of M_ℓ and W_ℓ be four nodes randomly chosen. The results are shown in Tables 5-8. Note that these computation can be done by only changing parameters.

Each node in DD requires 16 bytes memory space to store it if we erase variables in the node after its children have been created. Therefore, the total required memory for constructed DD can be approximately estimated by the number of nodes multiplied by 16.

Table 5: Experiment for Hamiltonian *s*-*t* path. Note that there is no Hamiltonian *s*-*t* path on G_ℓ when ℓ is even.

Graph	# of Nodes FBS	FBS Time	# of subgraphs
G_9	103627	0.00	2.68830×10^9
G_{11}	1149169	0.03	1.44578×10^{15}
G_{13}	11430100	3.02	1.73376×10^{22}
G_{15}	117635724	39.04	4.62865×10^{30}
M_{400}	351	0.00	60
M_{500}	95949	0.01	576
M_{600}	59215	0.02	12528
M_{700}	26778648	30.41	214020
W_{40}	2209511	0.08	102740
W_{50}	8266019	7.09	2.68762×10^6
W_{60}	64801681	75.37	3.135491×10^7
W_{70}	47685927	130.08	1.794379×10^8

Table 6: Experiment for rooted spanning forest.

Graph	# of Nodes FBS	FBS Time	# of subgraphs
G_6	228871	0.06	2.95470×10^{15}
G_7	1377790	3.07	3.59946×10^{21}
G_8	7946770	25.09	4.22129×10^{28}
G_9	44595187	172.06	4.84433×10^{36}
M_{300}	11089	0.02	2.69631×10^8
M_{400}	92536	0.03	3.10111×10^{10}
M_{500}	859422296	2952.04	1.47339×10^{21}
W_{20}	106542	0.03	9.92928×10^{09}
W_{30}	2977619	6.03	1.54711×10^{15}

5 Related Work

An algorithm for constructing the DD representing all the spanning trees has already been proposed by Sekine et al. [18], which is similar to our frontier-based search. The purpose of their paper is not to enumerate all the spanning tree but to compute the Tutte polynomial. However, the constructed DD for computing the polynomial can be seen as the representation of all the spanning tree. Sekine and Imai [17] counts the number of *s*-*t* paths in a given graph by using the DD representing all the spanning tree and modifying it. The efficiency of the algorithm is not superior to that of Simpath. Motter and Markov [13] develop an algorithm called a compressed breadth-first search algorithm for CNF-SAT. The DD which appears in their paper is similar to ours constructed in this paper.

In fact, we can regard the DD constructed by frontier-based search as BDDs or Zero-suppressed BDDs (ZDDs) [11]. In [2], a general branch-and-bound algorithm for discrete optimization is proposed. In this algorithm, the search space is represented by BDDs that the algorithm constructs. Hooker [5] considers a relation

Table 7: Experiment for Steiner tree.

Graph	# of Nodes FBS	FBS Time	# of subgraphs
G_6	94571	0.04	1.93747×10^{15}
G_7	561319	1.06	3.51350×10^{21}
G_8	3213376	10.02	7.72562×10^{28}
G_9	17952618	68.03	2.05461×10^{37}
M_{300}	4672	0.01	1.38721×10^{8}
M_{400}	37217	0.02	4.93532×10^{9}
M_{500}	343769770	1130.37	2.43278×10^{18}
W_{20}	44190	0.02	6.49571×10^{9}
W_{30}	1195399	2.06	8.66131×10^{14}
W_{40}	763758988	2168.72	8.31267×10^{19}

Table 8: Experiment for matching.

Graph	# of Nodes FBS	FBS Time	# of subgraphs
G_5	1105	0.01	2.81069×10^{6}
G_{10}	178937	0.07	2.17214×10^{27}
G_{15}	14000121	82.08	4.13975×10^{62}
G_{20}	830734329	9106.00	1.94786×10^{112}
M_{500}	64903	0.03	2.39934×10^{14}
M_{600}	465162	1.05	1.19158×10^{20}
M_{700}	10593099	39.07	7.95097×10^{25}
M_{800}	565761693	110.36	8.47895×10^{31}
W_{50}	662114	1.06	1.46547×10^{14}
W_{60}	1615444	4.02	9.90674×10^{16}
W_{70}	4299561	12.06	5.72795×10^{19}
W_{80}	20706108	67.06	3.70771×10^{22}

between dynamic programming and the BDDs constructed by an algorithm similar to frontier-based search.

6 Conclusion

We have seen that frontier-based search constructs DDs representing various subgraphs by specifying the degrees of some vertices, components which some vertices belong to, the number of components, and whether having a cycle or not. We also have carried out some experiments to show the effectiveness of our algorithm.

There are at least two another implementation on frontier-based search. One is a Python/C++ library provided by Inoue [6], which can allow us to treat DDs without knowing behavior of the algorithm. The other is a C++ implementation by Iwashita et al. [8], using more advanced techniques called subsetting.

Since a DD constructed by frontier-based search satisfies requirements of ZDD, many properties of ZDD are useful when we exploit a constructed DD. One of the most important property is that we can compute a new set from two sets by set operations such as union and intersection using ZDDs. Using this property, it is possible to eliminate subgraphs which do not satisfy another condition from the DD constructed by frontier-based search (see [7] for example). We believe that our frontier-based search can be used to solve various graph problems.

7 Acknowledgment

T. Inoue and H. Iwashita's work was performed while they were affiliated with JST ERATO, Japan.

References

[1] David Avis and Komei Fukuda. Reverse search for enumeration. *Discrete Applied Mathematics*, 65:21–46, 1993.

[2] David Bergman, Andre A. Cire, Willem-Jan van Hoeve, and John N. Hooker. Discrete optimization with decision diagrams. *submitted manuscript*, 2013.

[3] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[4] Gary Hardy, Corinne Lucet, and Nikolaos Limnios. A BDD-based heuristic algorithm for design of reliable networks with minimal cost. In Jiannong Cao, Ivan Stojmenovic, Xiaohua Jia, and Sajal K. Das, editors, *Mobile Ad-hoc and Sensor Networks*, volume 4325 of *Lecture Notes in Computer Science*, pages 244–255. Springer Berlin Heidelberg, 2006.

[5] John N. Hooker. Decision diagrams and dynamic programming. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 7874 of *Lecture Notes in Computer Science*, pages 94–110. Springer Berlin Heidelberg, 2013.

[6] Takeru Inoue, Hiroaki Iwashita, Jun Kawahara, and Shin-ichi Minato. Graphillion: Software library designed for very large sets of graphs in Python. *Hokkaido University, Division of Computer Science, TCS Technical Reports*, TCS-TR-A-13-65, 2013.

[7] Takeru Inoue, Keiji Takano, Takayuki Watanabe, Jun Kawahara, Ryo Yoshinaka, Akihiro Kishimoto, Koji Tsuda, Shin-ichi Minato, and Yasuhiro Hayashi. Distribution loss minimization with guaranteed error bound. *IEEE Transactions on Smart Grid*, 5(1):102–111, Jan 2014.

[8] Hiroaki Iwashita and Shin-ichi Minato. Efficient top-down ZDD construction techniques using recursive specifications. *Hokkaido University, Division of Computer Science, TCS Technical Reports*, TCS-TR-A-13-69, 2013.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 4A, Combinatorial Algorithms, Part 1.* Addison-Wesley Professional, 1st edition, March 2011.

[10] Christoph P. Mayer. osm2wkt - openstreetmap to wkt conversion. http://www.tm.kit.edu/ mayer/osm2wkt, 2010.

[11] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th ACM/IEEE Design Automation Conference*, pages 272–277, 1993.

[12] Shin-ichi Minato, Takeaki Uno, and Hiroki Arimura. LCM over ZBDDs: Fast generation of very large-scale frequent itemsets using a compact graph-based representation. In Takashi Washio, Einoshin Suzuki, Kai Ming Ting, and Akihiro Inokuchi, editors, *Advances in Knowledge Discovery and Data Mining*, volume 5012 of *Lecture Notes in Computer Science*, pages 234–246. Springer Berlin Heidelberg, 2008.

[13] DoRon B. Motter and Igor L. Markov. A compressed breadth-first search for satisfiability. In David M. Mount and Clifford Stein, editors, *Algorithm Engineering and Experiments*, volume 2409 of *Lecture Notes in Computer Science*, pages 29–42. Springer Berlin Heidelberg, 2002.

[14] NetworkX developer team. NetworkX. https://networkx.github.io/ documentation/latest/index.html.

[15] OpenStreetMap contributors. OpenStreetMap. http://www.openstreetmap.org.

[16] Robert. C. Read and Robert. E. Tarjan. Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, 5:237–252, 1975.

[17] Kyoko Sekine and Hiroshi Imai. Counting the number of paths in a graph via BDDs (special section on discrete mathematics and its applications). *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 80(4):682–688, 1997-04-25.

[18] Kyoko Sekine, Hiroshi Imai, and Seiichiro Tani. Computing the Tutte polynomial of a graph of moderate size. In *Proceedings of the 6th International Symposium on Algorithms and Computation (ISAAC)*, pages 224–233, 1995.

[19] Shuji Tsukiyama, Isao Shirakawa, Hiroshi Ozaki, and Hiromu Ariyoshi. An algorithm to enumerate all cutsets of a graph in linear time per cutset. *J. ACM*, 27(4):619–632, October 1980.

[20] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[21] Ryo Yoshinaka, Toshiki Saitoh, Jun Kawahara, Koji Tsuruma, Hiroaki Iwashita, and Shin-ichi Minato. Finding all solutions and instances of numberlink and slitherlink by ZDDs. *Algorithms*, 5(2):176–213, 2012.