Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)

Compiling Graph Substructures into Sentential Decision Diagrams

Masaaki Nishino,[1] **Norihito Yasuda,**[1] **Shin-ichi Minato,**[2] **Masaaki Nagata**[1]
[1]NTT Communication Science Laboratories, NTT Corporation
[2]Graduate School of Information Science and Technology, Hokkaido University
nishino.masaaki@lab.ntt.co.jp

Abstract

The Zero-suppressed Sentential Decision Diagram (ZSDD) is a recently discovered tractable representation of Boolean functions. ZSDD subsumes the Zero-suppressed Binary Decision Diagram (ZDD) as a strict subset, and similar to ZDD, it can perform several useful operations like model counting and Apply operations. We propose a top-down compilation algorithm for ZSDD that represents sets of specific graph substructures, e.g., matchings and simple paths of a graph. We experimentally confirm that the proposed algorithm is faster than other construction methods including bottom-up methods and top-down methods for ZDDs, and the resulting ZSDDs are smaller than ZDDs representing the same graph substructures. We also show that the size constructed ZSDDs can be bounded by the branch-width of the graph. This bound is tighter than that of ZDDs.

Introduction

The Binary Decision Diagram (BDD) (Bryant 1986) is a data structure that represents a Boolean function in a compressed form. Once a Boolean function is compiled into a BDD, it can answer several types of queries in polytime against BDD size. Due to its effectiveness, BDDs have many variants, e.g., Sentential Decision Diagrams (SDD) (Darwiche 2011), Zero-suppressed BDDs (ZDD) (Minato 1993), and Zero-suppressed SDDs (ZSDD) (Nishino et al. 2016). Among them, SDD and ZSDD are prominent in that they support bottom-up construction and are more succinct than either BDD or ZDD (Darwiche 2011; Nishino et al. 2016).

Boolean functions appear in various situations. An important one is to represent specific *graph substructures*. Here we use graph substructures as subsets of graph nodes and edges satisfying specific conditions. A graph has several important substructures such as matchings, cycles, and simple paths. A set of such substructures can be represented as a Boolean function whose input variables correspond to every node or edge of the graph. It is known that decision diagrams can succinctly represent a set of graph substructures. For example, Knuth (Knuth 2011) shows that the set of all connected components of a graph, whose size is more than 10^{10}, can be represented as a BDD that has only several hundred nodes. Due to their succinctness and efficiency with regard to graph manipulation, decision diagrams representing sets of graph substructures appear in several applications including the task of assessing the reliability of networks (Hardy, Lucet, and Limnios 2007), graph coloring (Morrison, Sewell, and Jacobson 2016), and loss minimization over distribution networks (Inoue et al. 2014).

A key to the effective use of decision diagrams in manipulating graph substructures is a fast algorithm for compiling the set of graph substructures into decision diagrams. BDDs and ZDDs support the top-down construction method called SimPath (Knuth 2011). SimPath constructs BDDs and ZDDs representing sets of graph substructures directly from the input graphs, and is known to be much more efficient than the standard bottom-up construction method using the Apply operation since SimPath avoids constructing many intermediate decision diagrams.

We extend SimPath to construct ZSDDs. One of the most important properties of SimPath is that it can give a theoretical upper bound on the sizes of constructed BDDs and ZDDs (Inoue and Minato 2016). This upper bound is derived from the *path-width* of the input graph. Therefore, SimPath may take a long time, or even fail to compile graphs with large path-widths. In contrast, the proposed algorithm also can give an theoretical upper bound on the sizes of constructed ZSDDs, which is derived from the *branch-width* of the graph. Since the branch-width of a graph is equal to or smaller than the path-width, our algorithm can give tighter upper bounds than SimPath. Experiments show that our top-down construction algorithm is more efficient than bottom-up construction methods and SimPath, and can construct ZSDDs that are smaller than ZDDs obtained by SimPath.

Our method can be seen as a variant of a recently proposed top-down compilation algorithm for SDDs (Oztok and Darwiche 2015). The algorithm takes a CNF as its input and returns the corresponding SDD. The main difference from ours is that the algorithm uses CNFs as its input. For some graph substructures including simple paths and connected components, the size of CNFs representing the set of all substructures have exponentially many clauses, and it is impractical to prepare such CNFs. The other approach shown in (Choi, Tavabi, and Darwiche 2016), first exploits SimPath to construct a ZDD representing graph substructures and then converts it into a SDD and reduces the size by applying the dynamic minimization method (Choi and Darwiche 2013).

Since our algorithm is faster than SimPath, it runs faster than this method of converting ZDDs to SDDs.

Technical Preliminaries

Let $G = (V, E)$ be an undirected graph where V is the set of nodes and E is the set of edges. Let $|V|$ be the number of nodes and $|E|$ be the number of edges. Since some graph substructures can be represented as sets of edges, a set of such substructures can be represented as a family of sets whose universe is E. A family of sets whose universe is E can be represented as an $|E|$-ary Boolean function. Therefore, ZSDDs can be seen as representing families of sets. In the following, we treat ZSDDs as representing families of sets since this approach is suitable for representing graph substructures. We use $\mathfrak{P}$ to represent the family consisting of all subsets.

$(\mathbf{X}, \mathbf{Y})$-*decomposition* is the process of decomposing sets families into sub-families. Let f be a family of sets, and $\mathbf{X}$, $\mathbf{Y}$ be subsets of the universe of f; they form a partition of the universe. By using $(\mathbf{X}, \mathbf{Y})$-decomposition, f can be decomposed as

$$f = [p_1(\mathbf{X}) \sqcup s_1(\mathbf{Y})] \cup \cdots \cup [p_n(\mathbf{X}) \sqcup s_n(\mathbf{Y})] \ ,$$

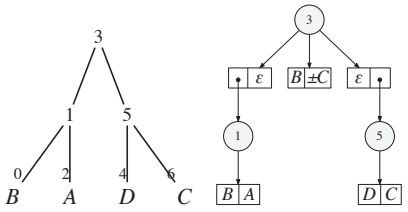
where $p_i(\mathbf{X})$, $s_i(\mathbf{Y})$ are sets families whose universes are $\mathbf{X}$ and $\mathbf{Y}$, respectively. In the following, we write p_i and s_i instead of $p_i(\mathbf{X})$ and $s_i(\mathbf{Y})$. Operations $\cup$ and $\sqcup$ are union and join operations over sets of families defined as $f \cup g = \{a \mid a \in f \text{ or } a \in g\}$, and $f \sqcup g = \{a \cup b \mid a \in f \text{ and } b \in g\}$. We call $p_1, \ldots, p_n$ *primes* and $s_1, \ldots, s_n$ *subs*. If primes are exclusive ($p_i \cap p_j = \emptyset$ for all $i \neq j$), exhaustive ($\bigcup_{i=1}^{n} p_i = \mathfrak{P}$), and consistent ($p_i \neq \emptyset$ for all i), then we say the decomposition is *an* $(\mathbf{X}, \mathbf{Y})$*-partition*, and denote it as $\{(p_1, s_1), \ldots, (p_n, s_n)\}$. Here we define $\cap$ as $f \cap g = \{a \mid a \in f \text{ and } a \in g\}$. Moreover, if $s_i \neq s_j$ for all $i \neq j$ is satisfied, we say the $(\mathbf{X}, \mathbf{Y})$-partition is *compressed*.

Example 1. Given $\mathbf{X} = \{A, B\}$ and $\mathbf{Y} = \{C, D\}$, a compressed $(\mathbf{X}, \mathbf{Y})$-partition of $\{\{A, B\}, \{B\}, \{B, C\}, \{C, D\}\}$ is

$$[\{\{A, B\}\} \sqcup \{\emptyset\}] \cup [\{\{B\}\} \sqcup \{\emptyset, \{C\}\}] \\ \cup [\{\emptyset\} \sqcup \{\{C, D\}\}] \cup [\{\{A\}\} \sqcup \emptyset] \ ,$$

where $\{\{A, B\}\}$, $\{\{B\}\}$, $\{\emptyset\}$, and $\{\{A\}\}$ are primes, and $\{\emptyset\}$, $\{\emptyset, \{C\}\}$, $\{\{C, D\}\}$, and $\emptyset$ are subs.

A ZSDD represents a family of sets by recursively applying $(\mathbf{X}, \mathbf{Y})$-partitions to decompose the family into sub-families, where the order of partitions is determined by *a vtree*. A vtree is a binary tree whose leaves correspond to elements of the universe. We show a vtree example in Fig. 1 (a). Symbols appearing at leaves represent corresponding elements, and numbers appearing in each node represent vtree node IDs. Every internal node represents a partition of a universe into two groups: elements appearing in the left and the right subtrees. In this figure, root vtree node whose ID is 3 (denoted here as v_3) represents the $(\mathbf{X}, \mathbf{Y})$-partition of universe $\{A, B, C, D\}$ where $\mathbf{X} = \{A, B\}$ and $\mathbf{Y} = \{C, D\}$. Similarly, node v_1 represents a partition of universe $\{A, B\}$ where $\mathbf{X} = \{B\}$ and $\mathbf{Y} = \{A\}$. We use v^l, v^r to represent



(a) A vtree (b) A ZSDD

Figure 1: A vtree and a ZSDD that respects the vtree and represents $\{\{A, B\}, \{B\}, \{B, C\}, \{C, D\}\}$.

the left and the right child vtree nodes of v, respectively. We say that vtree node v is a *Shannon vtree node* if v^l is a leaf node, otherwise we say v is a *decomposition vtree node*. In Fig. 1 (a), v_3 is a decomposition vtree node, and its child nodes v_1 and v_5 are Shannon vtree nodes. To avoid confusion we call vtree nodes *vnodes* and represent them as v_i, v^l, v^r. We call ZSDD nodes *znodes* and represent them as z. We call graph nodes *gnodes* and represent them as u_i.

Zero-suppressed Sentential Decision Diagrams

The *Zero-suppressed Sentential Decision Diagram (ZSDD)* is a variant of the Sentential Decision Diagram (SDD). It subsumes the Zero-suppressed Binary Decision Diagram (ZDD) as a strict subset. ZSDDs are more succinct than ZDDs and support many of the polytime queries and transformations that ZDD supports. By comparison, ZSDDs support almost all operations supported by SDDs and tend to be smaller than SDDs when representing sparse families of sets. We say a family of sets is sparse if it consists of a small number of subsets, each of which is also small. Graph substructures such as simple paths tend to be represented as sparse set families, and are suitable for representation by ZSDDs. We select ZSDDs as the target of compilation algorithm, but it can be applied to SDDs with some small modification.

A ZSDD is recursively defined as follows. We say ZSDD α *respects* vnode v if the order of $(\mathbf{X}, \mathbf{Y})$-partitions used in α follows the vtree whose root is v. We use $\langle \alpha \rangle$ to represent the family of sets that ZSDD α represents.

Definition 1. *α is a ZSDD that respects vnode v iff:*

- $\alpha = \varepsilon$ *or* $\alpha = \bot$.
  Semantics: $\langle \varepsilon \rangle = \{\emptyset\}$ *and* $\langle \bot \rangle = \emptyset$
- $\alpha = X$ *or* $\alpha = \pm X$ *and v is a leaf with element X.*
  Semantics: $\langle X \rangle = \{\{X\}\}$ *and* $\langle \pm X \rangle = \{\{X\}, \emptyset\}$.
- $\alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}$*, v is internal, $p_1, \ldots, p_n$ are ZSDDs that respect a vnode that is in a subtree whose root is v^l, $s_1, \ldots, s_n$ are ZSDDs that respect a vnode that is in a subtree whose root is v^l, and $\langle p_1 \rangle, \ldots, \langle p_n \rangle$ is a partition.*
  Semantics: $\langle \alpha \rangle = \bigcup_{i=1}^{n} \langle p_i \rangle \sqcup \langle s_i \rangle$.

If ZSDDs are either ε, $\bot$, X, or $\pm X$, we say that they are *terminal*. Otherwise, a ZSDD represents a $(\mathbf{X}, \mathbf{Y})$-partition, and we call it a *decomposition*. Fig. 1 (b) shows an example ZSDD that represents the set family

$\{\{A,B\},\{B\},\{B,C\},\{C,D\}\}$ and respects the root vnode of the tree shown in Fig. 1 (a). A circle node and its child rectangle nodes represent a decomposition, which corresponds to an $(\mathbf{X},\mathbf{Y})$-partition. The figure in a circle node represents the vnode ID that the decomposition respects. Rectangle node $\boxed{p \mid s}$ represents a prime sub pair contained in an $(\mathbf{X},\mathbf{Y})$-partition where p is a prime and s is a sub. Every p, s are terminal ZSDDs or pointers to decomposition ZSDDs. We call circle nodes decision znodes and rectangle nodes element znodes. We define the size of a ZSDD as the sum of the sizes of $(\mathbf{X},\mathbf{Y})$-partitions appearing in the ZSDD. The size of the ZSDD shown in Fig. 1 (b) is 5.

We say a ZSDD is *trimmed* if it does not have $(\mathbf{X},\mathbf{Y})$-partitions of the form $\{(\varepsilon,\alpha),(\bar{\varepsilon},\bot)\}$, $\{(\alpha,\varepsilon),(\bar{\alpha},\bot)\}$, or $\{(\mathfrak{P},\bot)\}$, where $\bar{\alpha}$ represents a ZDD that corresponds to set family $\mathfrak{P} - \langle\alpha\rangle$. We say ZSDD α employs *implicit partitioning* if none of the $(\mathbf{X},\mathbf{Y})$-partitions contained in α have an element znode of the form $(\beta,\bot)$. The ZSDD shown in Fig. 1 (b) is a compressed and trimmed ZSDD that employs implicit partitioning. Our top-down algorithm constructs trimmed ZSDDs that employ implicit partitioning.

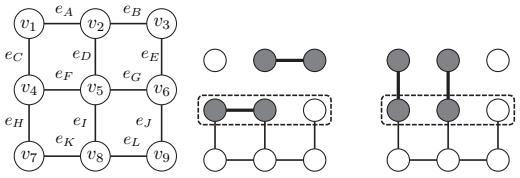
Top-down Compilation Algorithm

SimPath constructs ZDDs representing all graph substructures by creating ZDD nodes in order from the root to the terminals; it first makes a ZDD node that respects the first element, and then recursively makes child nodes of the created nodes to finally construct a ZDD. Our top-down algorithm is partially identical to SimPath, but it employs additional procedures for constructing ZSDDs. Similar to SimPath, our top-down construction algorithm can be used for constructing several different graph substructures by changing a few details of the algorithm. Due to the space limitation, we select matchings and simple paths as examples and describe algorithms for constructing them. We use the algorithm for constructing all matchings as the running example, since matchings are easier to construct than simple paths. We treat substructures that can be represented as families of sets whose universe is $E = \{e_1,\dots,e_{|E|}\}$. In the following, we use edge-IDs instead of edges, i.e., we represent $\{\{e_A,e_B\},\{e_C\}\}$ as $\{\{A,B\},\{C\}\}$.

Frontier Nodes

We first introduce an important mechanism for checking the equivalency of znodes. The proposed algorithm takes a vtree and graph G as its input, and generates znodes in order from the root to leaves; it first generates a znode that respects the root vnode, then it makes child znodes of the root znode. By recursively repeating this procedure for all child znodes, we can obtain the ZSDD representing all substructures. However, if we naively construct znodes in a top-down manner, the number of child znodes will grow exponentially. We therefore merge *equivalent* znodes when constructing them to avoid this.

Two znodes are equivalent if they respect the same vnode v and represent the same family of sets. Let α be the ZSDD respecting vnode v and representing family of sets f, and $E_v \subseteq E$ be the set of graph edges that correspond to leaf vnodes of the vtree whose root is v. E_v is the universe of f. Since f appears as a subfamily of the family of sets representing all graph substructures, f has some $S \subseteq E \setminus E_v$ for which $f \sqcup \{S\}$ is the set of specific graph substructures. If S changes, the corresponding f also changes, but for some $S, S' \subseteq E \setminus E_v$, the corresponding family f is equivalent. *Frontier gnodes* or *frontiers* can be used to judge this equivalency of S and S'. Let G_1 be the subgraph induced by E_v and G_2 be the subgraph induced by $E \setminus E_v$. We call the gnodes appearing in both G_1 and G_2 frontier gnodes. For some substructures, the possible family of sets, f, is determined by how edges in G_2 are connected to frontier gnodes, and hence we can check the equivalency of S and S' by checking the equivalency of edge connections to frontier gnodes. In the following, we use $F(v)$ to represent the set of frontier gnodes corresponding to vnode v.



(a) A graph (b) Equivalent connection patterns

Figure 2: An example graph and assignments of elements $\{A,B,\dots,G\}$ that give the same connection pattern.

Matching is an example of the substructure on which the above frontier-based equivalency check can be applied. Fig. 2 (a) is an example graph, and we want to find the family of sets whose universe is $E_v = \{H,I,\dots,L\}$ that forms the set of all matchings when combined with already selected edges from $\{A,B,\dots,G\}$. The set of frontier nodes $F(v)$ is $\{u_4,u_5,u_6\}$. Fig. 2 (b) shows two different choices of edges from $\{A,B,\dots,G\}$. Both choices make u_4, u_5 incident an edge, and u_6 incident no edge. It means both choices have the same connection patterns on frontier gnodes. Then the set of possible choices from E_v is $\{\emptyset,\{J\},\{J,K\},\{K\},\{L\}\}$, the same for both examples. This example shows the equivalency of znodes can be judged from how frontier nodes incident edges.

The top-down construction algorithm we will show below uses states of frontier gnodes as the label of generated ZSDD nodes, and we judge two znodes as equivalent if they respect the same vnode and have the same label. Labels are represented as $|V|$ element array m, where the state of frontier node $u_i \in V$ is stored in $m[i]$. In the case of matching, state $m[i]$ is represented by any of the following four symbols U (unconnected), C (connected), R (reserved), or F (finished). $m[i] = \mathsf{C}$ means u_i is a frontier gnode and incidents an edge. $m[i] = \mathsf{U}$ means u_i incidents no edge. $m[i] = \mathsf{R}$ means u_i is a frontier gnode and currently incidents no edge, but it must eventually incident an edge. $m[i] = \mathsf{F}$ means u_i is currently not a frontier node.

Algorithm

We show the scheme of the general top-down construction algorithm for ZSDDs in Alg. 1. The algorithm takes graph

Algorithm 1: A top-down construction algorithm

Input: $G = (V, E)$, a the root vtree node, v
Output: ZSDD representing the set of substructures of G

```
1  Z[v] ← rootState()
2  construct(v, Z)
3  Z ← reduce(Z)
4  (Optionally Z ← compress(Z))
5  return Z
```

Algorithm 2: construct(v, Z)

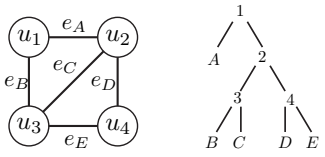
```
1  if v is a Shannon vnode then
2      for z ∈ Z[v] do
3          elems ← ∅
4          m_f ← shannonChild(v, z, false)
5          if m_f ≠ ⊥ then
6              elems ← elems ∪{(ε, unique(m_f))}
7          m_t ← shannonChild(v, z, true)
8          if m_t ≠ ⊥ then
9              elems ← elems ∪ {(X, unique(m_t))}
10         if m_f ≠ ⊥ and m_f = m_t then
11             elems ← {(±X, unique(m_f))}
12         Set elems as the child nodes of z
13     if v^r is not a leaf vnode then construct(v^r, Z)
14 else                    // v is a decomposition vnode
15     for z ∈ Z[v] do
16         elems ← ∅
17         for (m_p, m_s) ∈ decompChild(v, z) do
18             y_p ← unique(m_p),   y_s ← unique(m_s)
19             elems ← elems ∪ {(y_p, y_s)}
20         Set elems as the child nodes of z
21     construct(v^l, Z),  construct(v^r, Z)
```

Algorithm 3: Subroutines used for Matchings

```
1  function shannonChild(v, z, t):
2      m ← copy of label of z
3      X ← element corresponds to vnode v^l
4      (u_a, u_b) ← vertices incident with edge e_X
5      if t = true then
6          if m[a] = C or m[b] = C then return ⊥
7          m[a] ← C,   m[b] ← C
8      for u_i ∈ F(v) \ F(v^r) do
9          if m[i] = R then return ⊥
10         else m[i] ← F
11     if v^r is not a leaf node then return m
12     else
13         Y ← the element corresponds to v^r
14         (u_a, u_b) ← gnodes incident with edge e_Y
15         if (m[a], m[b]) = (C, R) or (R, C) then return ⊥
16         else if m[a] = C or m[b] = C then return ε
17         else if m[a] = R or m[b] = R then return Y
18         else return ±Y
19 function decompChild(v, z):
20     elems ← ∅
21     common ← F(v^l) ∩ F(v^r)
22     m_p, m_s ← copies of the label of z
23     for u_i ∈ F(v^l) \ F(v) do m_p[i] ← F
24     for u_i ∈ F(v^r) \ F(v) do m_s[i] ← F
25     if common = ∅ then return {(m_p, m_s)}
26     for u_i ∈ common do
27         if m_p[i] = C then combs[i] ← {(C, C)}
28         else if m_p[i] = U then
29             combs[i] ← {(R, C), (C, U)}
30         else if m_p[i] = R then
31             combs[i] ← {(R, C), (C, R)}
32     for vals ∈ enumerateCombination(combs) do
33         m'_p ← copy of m_p,   m'_s ← copy of m_s
34         for u_i ∈ common do (m'_p[i], m'_s[i]) ← vals[i]
35         elems ← elems ∪ {(m'_p, m'_s)}
36     return elems
```

$G = (V, E)$ and a vnode as its input, and returns a ZSDD representing the set of all substructures. $Z[v]$ is a table storing decision znodes that respect vnode v. Since a ZSDD is represented as a set of decision znodes, the set of $Z[v]$ for all internal vnodes v can be seen as representing a ZSDD. The algorithm first calls rootState(), which returns the root znode with its label. The procedure differs when we construct different substructures. Next the algorithm calls construct(v, Z), which recursively construct znodes that respect vnode v and its descendant vnodes. Procedure reduce(Z) recursively deletes and merges znodes to make a trimmed and implicitly partitioned ZSDD. We omit details of reduce(Z). The obtained ZSDD is trimmed and implicitly partitioned, but not compressed. Procedure compress uses Apply operations to make a compressed ZSDD. Compressed ZSDDs are canonical, but compression may increase ZSDD size (Van den Broeck and Darwiche 2015).

Alg. 2 shows the procedure construct(v, Z). It uses a different procedure depending on whether v is a Shannon vnode or not. If v is a Shannon vnode (lines 1-13), it calls shannonChild(v, z, t) with different $t \in \{\text{true}, \text{false}\}$. This procedure creates m that is either the label of a child of z that respects v^r or a terminal znode. unique(m) takes m as the input, and returns m if m is a terminal znode. Otherwise it checks whether there already exists a znode respecting v^r and has the same label in $Z[v^r]$. If such a znode exists, the procedure returns the address of the znode. Otherwise it creates a new decision znode that respects v^r and has label m, stores it in $Z[v^r]$ and returns the address of the znode. If neither m_f nor m_t are $\bot$ or $m_f \neq m_t$, we set $(\mathbf{X}, \mathbf{Y})$-partition $\{(\varepsilon, \text{unique}(m_f)), (X, \text{unique}(m_t))\}$ as the child of z (line 12). If $m_f = m_t$, we compress the child nodes to make element $\{(\pm X, \text{unique}(m_f))\}$ (line 11). If v is a decomposition vnode (line 14-21), it calls decompChild(v, z) for every node $z \in Z[v]$. decompChild(v, z) returns a set of pairs of labels or terminal znodes that form primes and subs. Finally, we set elements as children of znode z, and recursively call construct(v, Z) for $v = v^l$ and $v = v^r$ (line 21). If a right-linear vtree is given as the input, construct(v, Z) always calls shannonChild(v, z, t). Then the algorithm is almost identical to SimPath. Our top-down algorithm extends



(a) A graph (b) A vtree

Figure 3: An example graph and a vtree used in Example 2.

SimPath by introducing decompChild(v, z) to make child znodes of a parent that respect a decomposition vnode.

The proposed method can be applied to several graph substructures by designing rootNode(), shannonChild(v, z, t), and decompChild(v, z) appropriately for the target substructure. We first show concrete procedures used for constructing the set of all matchings. Procedure rootState() returns table m where $m[i] = \mathsf{U}$ for every $u_i \in V$, since no gnodes incident edges in the initial state. Alg. 3 shows shannonChild(v, z, t) and decompChild(v, z). Procedure shannonChild(v, z, t) updates the label of znode z to make labels of its child nodes. If $t = \mathsf{true}$, then the procedure updates labels by adding edge X corresponding to leaf vnode v^l. Let $u_a, u_b \in V$ be the gnodes that incident X. If $m[a]$ or $m[b]$ is C, then adding X makes more than two edges incident u_a or u_b, which violates the definition of matching, thus it returns $\perp$ (line 6). Otherwise, we update $m[a]$ and $m[b]$. We then set $m[i] \leftarrow \mathsf{F}$ for every $u_i \in F(v)$ that will not appear in $F(v^r)$ (line 8-10). If $m[i] = \mathsf{R}$ for some $u_i \in F(v) \setminus F(v^r)$, it returns $\perp$ since the condition that u_i must incident an edge will not be satisfied (line 9). If v^r is not a leaf node, we finish the procedure and return m (line 11). If v^r is a leaf, the procedure returns the terminal ZSDD node according to the states of frontier nodes (line 12-18).

Procedure decompChild(v, z) is simple if v^l and v^r have no common frontier nodes; in such case, it first copies label m of a parent node to m_p and m_s (line 22, 23), and then sets $m_p[i] \leftarrow \mathsf{F}$ and $m_s[i] \leftarrow \mathsf{F}$ for all gnodes u_i that do not appear in $F(v^l)$ and $F(v^r)$, respectively (line 24, 25). Finally, it returns the pair (m_p, m_s) (line 26). If v^l and v^r have common frontier gnodes, we make child gnodes for all possible pairs of prime and sub labels. Suppose that there is a common frontier gnode $u_i \in F(v^l) \cap F(v^r)$, and $m[i] = \mathsf{U}$. Then u_i can incident at most one edge that is either in sub-vtree v^l or v^r. If such edge is in v^r, then $m_p[i] = \mathsf{C}$ since no edge in v^l incidents u_i, otherwise $m_s[i] = \mathsf{C}$. Therefore, possible assignments on $(m_p[i], m_s[i])$ are either (C, U) or (R, C), they correspond to the two cases above. Here, the latter is (R, C) instead of (U, C) because primes must be exclusive; if $m_p[i] = \mathsf{U}$, it contains cases in which no edge in v^l incidents u_i. Such cases may also occur when $m_p[i] = \mathsf{C}$. In this way, we store all possible assignments on $(m_p[i], m_s[i])$ for all $u_i \in F(v^l) \cap F(v^r)$ in combs (line 27-33). Then we enumerate all combinations of possible assignments of states over common frontier nodes, and make pairs of primes and subs for every possible assignment (line 34-37). Procedure enumerateCombination enumerates all possible combinations of pairs of gnode states stored in combs, and every vals contains pairs of $(m_p[i], m_s[i])$ for all $u_i \in$ common.

Example 2. Let us construct the ZSDD representing all matchings of the graph shown in Fig. 3 (a), where the constructed ZSDD follows the vtree shown in Fig. 3 (b). The graph has four gnodes $u_1, \ldots, u_4$, so the labels of znodes are represented by arrays with 4 elements. In the following ,we use a tuple of four elements $(m[1], m[2], m[3], m[4])$ to represent the value of label m. We run the top-down construction algorithm shown in Alg. 1. First, it creates a root znode whose label is $(\mathsf{U}, \mathsf{U}, \mathsf{U}, \mathsf{U})$ by using rootState(). The procedure then calls construct(v_1, z). Since v_1 is a Shannon vnode, the procedure calls shannonChild(v_1, z, t) for different values of $t \in \{\mathsf{true}, \mathsf{false}\}$. Since u_1, u_2 are connected with e_A, $m_f = (\mathsf{U}, \mathsf{U}, \mathsf{U}, \mathsf{U})$ and $m_t = (\mathsf{C}, \mathsf{C}, \mathsf{U}, \mathsf{U})$. Thus, the root znode has two child elements (ε, m_f) and (A, m_t). Fig. 4 (a) shows the state after two element nodes are generated, where two decision znodes respecting v_2 are associated with labels m_f and m_t. We call znode with label m_f z_1, and that with m_t z_2.

Next, construct(v_2, Z) is called. Since v_2 is a decomposition vnode, procedure decompChild(v_2, z) is called for znodes z_1, z_2. Here $v^l = v_3$ and $v^r = v_4$, and common frontier gnodes of left and right child vnodes are $F(v_3) \cap F(v_4) = \{u_2, u_3\}$. z_1 has four possible labels of prime child nodes $(\mathsf{U}, \mathsf{C}, \mathsf{C}, \mathsf{F})$, $(\mathsf{U}, \mathsf{C}, \mathsf{R}, \mathsf{F})$, $(\mathsf{U}, \mathsf{R}, \mathsf{R}, \mathsf{F})$, and $(\mathsf{U}, \mathsf{R}, \mathsf{C}, \mathsf{F})$, and these prime labels form pairs with subs $(\mathsf{F}, \mathsf{U}, \mathsf{U}, \mathsf{U})$,$(\mathsf{F}, \mathsf{U}, \mathsf{C}, \mathsf{U})$, $(\mathsf{F}, \mathsf{C}, \mathsf{C}, \mathsf{U})$, and $(\mathsf{F}, \mathsf{C}, \mathsf{U}, \mathsf{U})$. Since these labels are distinct, we make 4 znodes respecting v_3, and 4 znodes respecting v_4. These znodes appear in Fig. 4 (b) in left-to-right order.

Similarly, znode z_2 has two possible labels of prime child znodes $(\mathsf{C}, \mathsf{C}, \mathsf{R}, \mathsf{F})$ and $(\mathsf{C}, \mathsf{C}, \mathsf{C}, \mathsf{F})$. Corresponding sub child znodes are $(\mathsf{F}, \mathsf{C}, \mathsf{C}, \mathsf{U})$ and $(\mathsf{F}, \mathsf{C}, \mathsf{U}, \mathsf{U})$. Since there are subs with the same labels, we do not make additional sub child znodes and instead point to znodes with the same labels (Fig. 4 (b)). After that, every znode respecting v_3 or v_4 are processed and finally the ZSDD in Fig. 4 (c) is obtained. We show how the leftmost znode respecting v_3, say z_3, is processed. It has label $(\mathsf{U}, \mathsf{C}, \mathsf{C}, \mathsf{F})$, and $F(v_3) = \{u_1, u_2, u_3\}$. Since v_3 is a Shannon vnode, procedure construct(v_3, Z) calls shannonChild(v_3, z_3, t) with $t = \{\mathsf{true}, \mathsf{false}\}$. If $t = \mathsf{false}$, the procedure returns element $(\varepsilon, \varepsilon)$ since v^r is a leaf vnode and the corresponding edge e_C cannot be taken since both $m[2], m[3] = \mathsf{C}$. If $t = \mathsf{true}$, the procedure returns $\perp$ since $m[3] = \mathsf{C}$ and taking e_B violates the condition for matching. As a result, the child element gnodes of z_3 become $\{(\varepsilon, \varepsilon)\}$. The obtained ZSDD is later reduced and compressed by applying reduce and compress.

Relation to Branch Decomposition

We can give upper bounds on the sizes of ZSDDs.

Theorem 1. *If α is the ZSDD representing the set of all matchings obtained by our top-down construction algorithm, the size of α is $O(|E|2^{2W})$, where W is the* width *of vtree and is defined as $W = \max_v |F(v)|$.*

Proof. The number of decision znodes that respect vtree node v is bounded by the number of possible frontier pat-

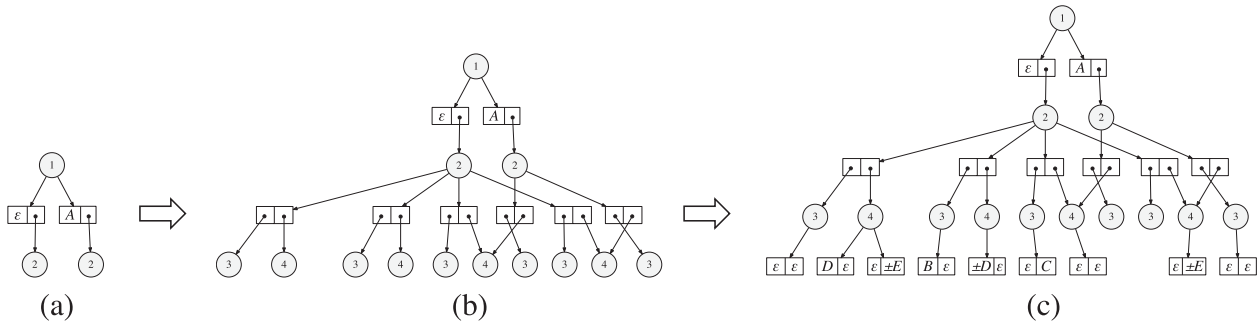


Figure 4: Illustrating construct procedure of Example 2.

terns, since every ZSDD node respecting the same vnode must have a distinct label. In our top-down construction algorithm, every frontier node can have three different values: C, U, or R. Since non-frontier nodes all have the same values in label m (either of U or F), the number of distinct labels are upper bounded by $3^{|F(v)|}$ for znodes that respect vnode v. This bound can be further tightened to $2^{|F(v)|}$, since if we select a vnode, v, and select a frontier gnode, $u_i \in F(v)$, then the set of possible values of $m[i]$ is either $\{\mathsf{R}, \mathsf{C}\}$ or $\{\mathsf{U}, \mathsf{C}\}$. Hence the number of decision znodes are bounded by $|E|2^W$. Since every decision znode has at most 2^W child element nodes, the ZSDD size is $O(|E|2^{2W})$. □

We can also show that the time and space complexity of the top-down algorithm for constructing all matchings is $O(|E|W2^{2W})$, since the procedure construct requires $O(W)$ time and space for every constructed ZSDD node.

Since W determines ZSDD size, finding a vtree with small width is important. We can make a vtree whose width equals the maximum width of a *branch decomposition* of the input graph. Branch decomposition (Robertson and Seymour 1991) of a graph is an unrooted binary tree, T, where each leaf corresponds to a distinct edge in E, and each non-leaf node of T has degree of exactly three. We define the width for every edge e in T as follows: if an edge is removed from T, then T is decomposed into exactly two connected components. Since leaf nodes of T correspond to graph edges, these two connected components can be seen as a partition. Let two subgraphs of G induced by the set of graph edges contained in each subtree be G_1 and G_2. We define the width of tree edge e as the number of graph nodes appearing in both G_1 and G_2. Given branch decomposition T, whose maximum width is W, we can easily construct a vtree whose maximum frontier size is bounded by W. The relation between branch decomposition and vtree indicates that the problem of finding a good vtree corresponds to the problem of finding a good branch decomposition. Although finding a branch decomposition with minimum width is generally a difficult problem, there are practical algorithms that can find good branch decompositions (e.g., (Cook and Seymour 2003)).

The *branch-width* of a graph is the smallest width of all possible branch decompositions. Let bw be the branch-width of an input graph, then the size of ZSDD representing all matchings is $O(|E|2^{2bw})$. When we use SimPath to construct a ZDD representing a set of matchings, then its size is $O(|E|2^{pw})$, where pw is the *path-width* of the input graph (Inoue and Minato 2016). Since pw and bw satisfy $pw = O(bw \log |V|)$ (e.g., (Bodlaender 1998), (Robertson and Seymour 1991)), our algorithm can give a tighter upper bound than SimPath.

Experiments

We conduct experiments to evaluate the performance of the proposed top-down construction algorithms for constructing ZSDDs representing all matchings and simple paths. As benchmarks, we use a bottom-up algorithm for ZSDDs and the top-down algorithm for ZDDs. We used the bottom-up construction algorithm[1] that converts a CNF into a ZSDD. Since CNFs representing the set of simple paths contains exponentially many clauses, we apply the bottom-up algorithm only to constructing the set of matchings. Since we apply the compress operation to ZSDDs constructed by the top-down algorithm, both the top-down and the bottom-up methods construct the same ZSDD. To implement the top-down algorithm for ZDDs, we use the top-down algorithm for ZSDDs with a limitation that vtrees must be right-linear. Since a ZSDD respecting a right-linear vtree is equivalent to a ZDD, the algorithm is equivalent to SimPath for ZDDs. The vtrees for ZSDDs are obtained by applying a heuristic algorithm for branch decomposition (Cook and Seymour 2003). We use two element orders for ZDDs. The first one uses the order obtained by a breadth-first traversal of input graphs, as is used in graphillion (Inoue et al. 2016), a library that implements a top-down construction algorithm for ZDDs. The other one uses the order induced from the vtrees used in the proposed method. Here we say an order is induced if a left-right traversal of a vtree gives the visiting order of variables (Xue, Choi, and Darwiche 2012).

We use benchmark graphs used in (Cook and Seymour 2003), which were obtained by applying Delaunay triangulation to the geometric instances in TSPLIB. We also used instances from the RomeGraph dataset [2]. We select the first 10 instances that have 100 nodes. We omit instances for which no method could finish within 600 seconds. All ex-

[1] https://github.com/nsnmsak/zsdd

[2] http://www.graphdrawing.org/download/rome-graphml.tgz

Instance	$\|V\|$	$\|E\|$	Compilation time (ms) BU	TD	Z (b)	Z (v)	Size TD	Z (b)	Z (v)
att48	48	130	95	11	132	35	7,420	40,370	20,438
berlin52	52	145	542	23	7,676	158	16,043	520,466	79,726
eil51	51	142	443	22	1,371	217	16,303	366,273	101,306
eil76	76	215	1,888	143	72,474	11,019	103,317	8,253,872	1,152,860
eil101	101	290	17,095	310	–	66,436	177,932	–	10,027,975
pr226	226	660	19,637	83	–	155,318	26,832	–	16,027,488
rat99	99	280	2,147	72	–	18,920	37,421	–	2,157,450
st70	70	197	1,418	62	62,559	70,336	46,288	3,861,677	7,244,529
grafo10106.100	100	119	94	1	371	4	885	141,278	2,690
grafo10116.100	100	149	481	244	–	5,590	108,637	–	853,204
grafo10124.100	100	139	469	118	59,927	385	71,210	5,664,264	187,536
grafo10153.100	100	136	210	36	29,943	119	21,164	4,462,425	55,647
grafo10183.100	100	132	108	10	29,508	658	6,094	3,513,816	213,194

Table 1: Results of constructing ZSDDs and ZDDs representing the set of all matchings.

Instance	$\|V\|$	$\|E\|$	Construction time (ms) TD	Z (b)	Z (v)	Size TD	Z (b)	Z (v)
att48	48	130	597	20,679	1,149	100,131	991,456	240,068
berlin52	52	145	2,059	–	16,225	313,513	–	1,305,617
eil51	51	142	1,705	–	27,873	225,314	–	2,742,091
ulysses22	22	57	2	183	11	1,106	15,649	5,377
grafo10106.100	100	119	2	1,914	4	462	7,861	1,056
grafo10124.100	100	139	36,291	–	60,099	2,043,633	–	4,694,606
grafo10153.100	100	136	14,809	–	3,124	324,406	–	432,263
grafo10183.100	100	132	252	–	140,348	35,318	–	364,206
grafo10184.100	100	140	14,848	–	113,433	616,622	–	2,205,368

Table 2: Results of constructing ZSDDs and ZDDs representing the set of all simple paths.

periments were conducted on a Linux machine with a Xeon E5-2687W 3.10 GHz CPU and 128 GB RAM.

Experimental results are shown in Tab. 1, 2. Here BU is the bottom-up method, TD is the top-down method (proposed), Z (b) and Z(v) are top-down methods for ZDDs that employ breadth first ordering and vtree traversing ordering. Since TD and BU return the same ZSDD, we only show the size of ZSDDs constructed by TD. The empty fields show experiments terminated by out of memory. We can see that the ZSDDs are always smaller than ZDDs. In the case of matchings, ZSDDs are up to 600 times smaller than ZDDs. For compilation time, TD is the fastest for almost all instances. One exception is grafo10124.000 of SimPath, where the compilation time of TD is longer than that of Z(v). This result is due to the fact that compilation time depends on the size of intermediate ZSDDs. If intermediate ZSDDs made by procedure construct are large, then compilation takes a long time even when the finally obtained ZSDDs are small.

Conclusion

We proposed top-down knowledge compilation algorithms for constructing ZSDDs that represent sets of substructures of input graphs. We showed a general top-down compilation algorithm and two concrete examples of compiling sets of matchings and sets of simple paths. Comparing with the SimPath algorithm, our method can give a better theoretical upper bounds on the sizes of ZSDDs. We experimentally confirmed that the proposed method runs fast and can construct more succinct ZSDDs than SimPath.

Appendix: Construction of Simple Paths

We show a top-down algorithm for constructing a ZSDD representing the set of all simple paths between two nodes $s, t \in G$. Here we say a path is simple if it does not contain nodes that appear more than twice. As in the case of matchings, the top-down construction algorithm for simple paths also exploits states of frontier gnodes as labels. Labels represent the connection relations between frontier nodes. Fig. 5 shows two equivalent frontier patterns with frontier u_4, u_5, u_6. In this example, u_4 is connected to $u_1 = s$, and u_5 and u_6 are terminal gnodes of a path. These connection relations between frontier nodes determine the possible choice of remaining edges. In this example, the only possible choice that forms a simple path in combination with selected edges is $\{H, I, J, K\}$. These two examples use different paths between u_5 and u_6, but they have the same connection pattern over frontier nodes, thus they are equivalent.

Next we show procedures shannonChild(v, z, t) and decompChild(v, z) for constructing ZSDDs representing the sets of all simple paths. We first overview these subroutines, then show the concrete design of labels and algorithms. shannonChild(v, z, t) updates the label of znode z by selecting or not selecting the edge corresponding to vnode v^l. In the same way as for matchings, decompChild(v, z) enumerates all possible combinations of primes and subs and then makes child znodes for every enumerated combination. This enumeration of possible combinations proceeds by (a) generating all combinations of possible states over common frontier gnodes ($F(v^l) \cap F(v^r)$), and then (b) enumerates all

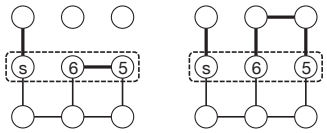


Figure 5: Equivalent frontiers (simple path).

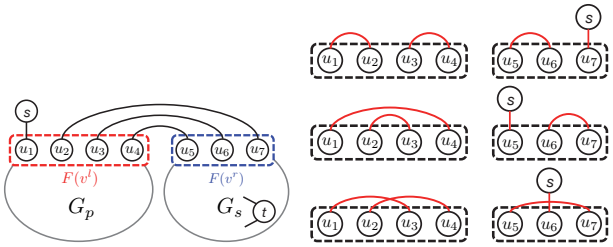


Figure 6: An example of enumeration of possible labels of child nodes in decompChild(v, z). (a) an example label of a parent znode (b) possible combinations of labels of primes and subs.

possible connections between paths that have terminals both in prime and sub frontier nodes. Procedure (a) is the same as that of matchings, and it generates possible assignments of $m_p[i], m_s[i]$ depending on the current value $m[i]$ for every common frontier gnode $u_i \in F(v^l) \cap F(v^r)$.

We elucidate procedure (b) by using the example shown in Fig. 6. Suppose that $F(v)$ for vnode v is $\{u_1, \ldots, u_7\}$, $F(v^l) = \{u_1, \ldots, u_4\}$ and $F(v^r) = \{u_5, u_6, u_7\}$. If the label of a znode, z, that respects v has connections shown in Fig. 6 (a), then the new labels of primes and subs are obtained by copying the corresponding values of the label of z. In this case, there remains three paths that connect prime and sub frontier nodes, (u_2, u_7), (u_3, u_6), and (u_4, u_5). These connections between prime and sub frontiers prevent top-down construction at prime and sub ZSDDs from running independently. For example, source gnode s is currently connected to $u_1 \in F(v^l)$, but target gnode t is contained only in the subgraph G_s that is induced by edges in vtree v^r. Thus we have to connect u_1 with either u_2, u_3, or u_4 to form a path. Since this decision impacts the connection patterns of sub, we cannot process primes and subs independently. We therefore enumerate all possible connections between the terminals of these connected paths that appear in $F(v^l)$. If connections between terminals of primes are once determined, then the connection patterns of subs can be processed independently. Fig. 6 (b) shows all possible connection patterns of prime frontier gnodes and corresponding sub frontier gnodes, given the connection pattern of Fig. 6 (a). There are three possible choices of connecting connecting prime frontier vnodes, and these connections result in different frontier states in subs. Once these connection patterns are enumerated, the top-down construction for v^r proceeds independently. The top-down construction procedure for v^l also proceeds so as to find all possible families of sets that accomplish the desired connections between terminals. We call these desired connections *reserved connections*. This procedure also can be performed independently.

Algorithm 4: shannonChild(v, z, t)

```
1  m ← copy of label of z
2  X ← element corresponds to vtree node v^l
3  (u_a, u_b) ← gnodes incident with edge e_X
4  if t = true then
5      if m[a] = 0 or m[b] = 0 then return ⊥
6      if m[a] = b and m[b] = a then return ⊥
7      if m[a] < 0 and m[b] < 0 then
8          if m[a] = −b and m[b] = −a then
9              m[a] ← 0   m[b] ← 0
10             if finished(m) then return ε
11         else return ⊥
12     else
13         if m[a] = −a then m[a] ← a
14         if m[b] = −b then m[b] ← b
15         t_a ← m[a],   t_b ← m[b]
16         m[a] ← 0,   m[b] ← 0
17         m[σ(t_a) · t_a] ← σ(t_a) · t_b
18         m[σ(t_b) · t_b] ← σ(t_b) · t_a
19 for u_i ∈ F(v^r) \ F(v) do
20     if m[i] ≠ 0 and m[i] ≠ i then return ⊥  else m[i] ← 0
21 if v^r is not a leaf vnode then return m
22 else
23     if finished(m) then return ε
24     Y ← element corresponds to vtree node v^r
25     (u_a, u_b) ← gnodes incident with edge e_Y
26     if m[a] = −b and m[b] = −a then
27         m[a] ← 0,   m[b] ← 0
28         if finished(m) then return Y
29     return ⊥
```

Next we show the concrete procedures. We first show how to represent connection patterns between frontier gnodes. In the following, we assume $s = u_1$ and $t = u_{|V|}$ without loss of generality. We represent connection patterns by using size $|V|$ array m whose values are integers ranging from $-|V|$ to $|V|$. Suppose every gnode $u_i \in V$ is represented by integer $1 \leq i \leq |V|$. If $m[i] = 0$, it means u_i incidents two edges and appears as an internal point of a simple path. If $m[i] = i$, the gnode incidents no edges. If $m[i] = j$ where $1 \leq j \leq |V|$ and $j \neq i$, i is a terminal gnode of a path that is not connected to both s and t, and another terminal of the path is u_j, i.e., $m[i] = j$ means $m[j] = i$. If $m[i] = j$ where $-|V| \leq j \leq -1$ and $j \neq -i$, then i and j has a reserved connection, i.e., i and j must be connected by a simple path. Since $u_1 = s$ must be connected to $u_{|V|} = t$, s and t have a reserved connection. It is represented as $m[1] = -|V|$ and $m[|V|] = -1$ in the initial state. The reserved connections appearing in the upper prime label in Fig. 6 (b) is represented by $(m[1], m[2], m[3], m[4]) = (-2, -1, -4, -3)$. The corresponding sub label is represented by $(m[5], m[6], m[7]) = (6, 5, -s)$. If $m[i] = -i$, then u_i currently incidents no edges but it must be incident two edges.

We next show the three sub-procedures. rootState() re-

Algorithm 5: decompChild(v, z)

```
1  elems ← ∅
2  common ← F(v^l) ∩ F(v^r)
3  m_p ← label of z,  m_s ← label of z
4  for u_i ∈ F(v^l) \ F(v) do m_p[i] ← 0
5  for u_i ∈ F(v^r) \ F(v) do m_s[i] ← 0
6  for u_i ∈ common do
7      if m_p[i] = 0 then
8          combs[i] ← {(0, 0)}
9      else if m_p[i] = i then
10         combs[i] ← {(−i, 0), (0, i), (π, π)}
11     else if m_p[i] = −i then
12         combs[i] ← {(−i, 0), (0, −i), (π, π)}
13     else
14         combs[i] ← {(m[i], 0), (0, m[i])}
15 for vals ∈ enumerateCombination(combs) do
16     m'_p ← copy of m_p,    m'_s ← copy of m_s
17     for u_i ∈ common do
18         (m'_p[i], m'_s[i]) ← vals[i]
19     connection ← connections between m'_p and m'_s
20     if connection is empty then
21         elems ← elems ∪ {(m'_p, m'_s)}
22     else
23         for cVals ∈ enumeratePats(connection) do
24             m''_p ← m'_p,  m''_s ← m'_s
25             Update m''_p, m''_s so as to follow cVals
26             elems ← elems ∪ {(m''_p, m''_s)}
27 return elems
```

turns initial state m whose values are $m[1] = -|V|$, $m[|V|] = -1$, and $m[i] = i$ for $i = 2, \ldots, |V| - 1$.

We show shannonChild(v, z, t) in Alg. 4. If $t =$ true, the algorithm first tries to add edge e_X that corresponds to leaf vnode v^l (line 4-18). If either u_a or u_b already incidents two edges (line 5), or u_a and u_b are terminal gnodes of a path and connecting them makes an cycle (line 6), we return $\perp$ since the set of selected edges will never form a simple path. If $m[a] < 0$ and $m[b] < 0$, we check if they are reserved to be connected (line 8), and if so, we check whether a simple path has been found or not by calling finished(m). If $m[i] = 0$ or $m[i] = i$ for all $u_i \in V$, then finished(m) returns true. If finished, then terminal znode ε is returned (line 10). If u_a, u_b are not reserved, then $\perp$ is returned. In other cases, we connect u_a, u_b to update values to reflect the new connection made by adding e_X (line 12-18), where $\sigma(t)$ is 1 if $t \geq 0$, otherwise -1. Then for all u_i that will be removed from frontier, we set $u[i] = 0$. If u_i is a terminal of a path or reserved to be connected, we return $\perp$. Finally, if v^r is a leaf vnode, we return a terminal znode depending on the current label m, otherwise we return m.

Procedure decompChild(v, z) starts working in almost the same way as matchings; it enumerates possible assignments over common gnodes in $F(v^l) \cap F(v^r)$, and makes m'_p or m'_s for every assignment (line 1-14). Symbol π, appears in lines 10 and 12, is used to indicate that u_i will incident two edges, one is in v^l and the other is in v^r. It means if $m_p[i] = m_s[i] = \pi$, it can be treated as a path connecting prime and sub frontier gnodes. After enumerating all possible assignments on common frontier nodes, the procedure finds all connection paths between m'_p and m'_s (line 19). If no connection exists, it adds (m'_p, m'_s) to elems (line 21). Otherwise, the procedure also enumerates all possible reserved connections between prime frontier gnodes (line 23). textsfcVals stores reserved connections between prime frontier gnodes and values of subs reflecting the reserved connections. We use cVal to update m''_p and m''_s (line 25), then we add them to elems (line 26).

References

Bodlaender, H. L. 1998. A partial k-arboretum of graphs with bounded treewidth. *Theor. Comput. Sci.* 209(1):1 – 45.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Trans. on* C-35(8):677–691.

Choi, A., and Darwiche, A. 2013. Dynamic minimization of sentential decision diagrams. In *AAAI*, 187–194.

Choi, A.; Tavabi, N.; and Darwiche, A. 2016. Structured features in naive bayes classification. In *AAAI*, 3233–3240.

Cook, W., and Seymour, P. 2003. Tour merging via branch-decomposition. *INFORMS J. on Computing* 15(3):233–248.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, 819–826.

Hardy, G.; Lucet, C.; and Limnios, N. 2007. K-terminal network reliability measures with binary decision diagrams. *Reliability, IEEE Trans. on* 56(3):506–515.

Inoue, Y., and Minato, S. 2016. Acceleration of zdd construction for subgraph enumeration via path-width optimization. Technical Report TCS-TR-A-16-80, Division of Computer Science, Hokkaido University.

Inoue, T.; Takano, K.; Watanabe, T.; Kawahara, J.; Yoshinaka, R.; Kishimoto, A.; Tsuda, K.; Minato, S.; and Hayashi, Y. 2014. Distribution loss minimization with guaranteed error bound. *Smart Grid, IEEE Trans. on* 5(1):102–111.

Inoue, T.; Iwashita, H.; Kawahara, J.; and Minato, S. 2016. Graphillion: software library for very large sets of labeled graphs. *J. on Soft. Tool. for Tech. Trans.* 18(1):57–66.

Knuth, D. E. 2011. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley.

Minato, S. 1993. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *DAC*, 272–277.

Morrison, D. R.; Sewell, E. C.; and Jacobson, S. H. 2016. Solving the pricing problem in a branch-and-price algorithm for graph coloring using zero-suppressed binary decision diagrams. *INFORMS J. on Computing* 28(1):67–82.

Nishino, M.; Yasuda, N.; Minato, S.; and Nagata, M. 2016. Zero-suppressed sentential decision diagrams. In *AAAI*, 1058–1066.

Oztok, U., and Darwiche, A. 2015. A top-down compiler for sentential decision diagrams. In *IJCAI*, 3141–3148.

Robertson, N., and Seymour, P. D. 1991. Graph minors. x. obstructions to tree-decomposition. *J. of Comb. Theor., Series B* 52(2):153–190.

Van den Broeck, G., and Darwiche, A. 2015. On the role of canonicity in knowledge compilation. In *AAAI*, 1641–1648.

Xue, Y.; Choi, A.; and Darwiche, A. 2012. Basing decisions on sentences in decision diagrams. In *AAAI*, 842–849.