Compiling Graph Substructures into Sentential Decision Diagrams

Masaaki Nishino,1 Norihito Yasuda,1 Shin-ichi Minato,2 Masaaki Nagata1
1NTT Communication Science Laboratories, NTT Corporation
2Graduate 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 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.

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 \( \mathcal{P} \) to represent the family consisting of all subsets.

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

\[
f = [p_1(X) \cup s_1(Y)] \cup \ldots \cup [p_n(X) \cup s_n(Y)],
\]

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

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

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

where \( \emptyset, \{A\}, \{B\}, \{\emptyset\} \) and \( \{\emptyset\} \) are primes, and \( \emptyset, \{\emptyset\} \) are sub substructures such as simple paths tend to be represented as simple path sets, 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 \( \alpha \) respects vnode \( v \) if the order of \((X, Y)\)-partitions used in \( \alpha \) follows the vtree whose root is \( v \). We use \((\alpha)\) to represent the family of sets that ZSDD \( \alpha \) represents.

**Definition 1.** \( \alpha \) is a ZSDD that respects vnode \( v \) iff:

- \( \alpha = \varepsilon \) or \( \alpha = \bot \).
- \( \alpha = X \) or \( \alpha = \pm X \) and \( v \) is a leaf with element \( X \).
- \( \alpha = X \) and \( \langle X \rangle = \{\emptyset\} \).
- \( \alpha = \{(p_1, s_1), \ldots, (p_n, s_n)\}, v \) is internal, \( p_1, \ldots, p_n \) are ZSDDs that respect a vnode that is in a subtree whose root is \( v \), and \( \langle p_i \rangle, \ldots, \langle p_n \rangle \) is a partition.

If ZSDDs are either \( \varepsilon, \bot, X, \) or \( \pm X \), we say that they are terminal. Otherwise, a ZSDD represents a \((X, Y)\)-partition, and we call it a decomposition. Fig. 1 (b) shows an example ZSDD that represents the set family
\{\{A, 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 \((X, Y)\)-partition. The figure in a circle node represents the vnode ID that the decomposition respects. Rectangle node \(p[s]\) represents a prime sub pair contained in an \((X, Y)\)-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 \((\alpha, \varepsilon)\) where \(\alpha\) represents a ZDD that corresponds to set family \(\{\{\emptyset\}\} - \langle\alpha\rangle\). We say ZSDD \(\alpha\) employs implicit partitioning if none of the \((X, Y)\)-partitions contained in \(\alpha\) have an element vnode of the form \((\beta, \emptyset)\). 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 structures that can be represented as families of sets whose universe is \(E = \{e_1, \ldots, e_{|E|}\}\). 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 vnode that respects the root vnode, then it makes child znodes of the root vnode. 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 and represent the same family of sets. Let \(\alpha\) 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 nodes 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 \(f \cup \{S\}\) 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 \(v\).

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 \(E_v = \{H, I, \ldots, L\}\) that forms the set of all matchings when combined with already selected edges from \(\{A, B, \ldots, G\}\). The set of frontier gnodes \(F(v)\) is \(\{u_4, u_5, u_6\}\). Fig. 2 (b) shows two different choices of edges from \(\{A, B, \ldots, G\}\). 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). \(m[i] = C\) means \(u_i\) is a frontier gnode and incidents an edge. \(m[i] = U\) means \(u_i\) incidents no edge. \(m[i] = R\) means \(u_i\) is a frontier gnode and currently incidents no edge, but it must eventually incident an edge. \(m[i] = F\) 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 tree node, $v$
Output: ZSDD representing the set of substructures of $G$
1. $Z[v] \leftarrow \text{rootState}()$
2. $\text{construct}(v, Z)$
3. $Z \leftarrow \text{reduce}(Z)$
4. (Optionally $Z \leftarrow \text{compress}(Z)$)
5. return $Z$

Algorithm 2: $\text{construct}(v, Z)$

1. if $v$ is a Shannon vnode then
   2. for $z \in Z[v]$ do
      3. $\text{elems} \leftarrow \emptyset$
      4. $m_f \leftarrow \text{shannonChild}(v, z, \text{false})$
      5. if $m_f \neq \perp$ then
         6. $\text{elems} \leftarrow \text{elems} \cup \{(\varepsilon, \text{unique}(m_f))\}$
      7. $m_t \leftarrow \text{shannonChild}(v, z, \text{true})$
      8. if $m_t \neq \perp$ then
         9. $\text{elems} \leftarrow \text{elems} \cup \{(X, \text{unique}(m_t))\}$
   10. Set $\text{elems}$ as the children of $z$
11. if $v'$ is not a leaf vnode then $\text{construct}(v', Z)$
12. else
   13. for $z \in Z[v']$ do
      14. $\text{elems} \leftarrow \emptyset$
      15. for $(m_p, m_s) \in \text{decompChild}(v, z)$ do
      16. $\text{elems} \leftarrow \text{elems} \cup \{(y_p, y_s)\}$
      17. Set $\text{elems}$ as the child nodes of $z$
      18. $\text{construct}(v', Z)$.
      19. $\text{construct}(v', Z)$

Algorithm 3: Subroutines used for Matchings

function $\text{shannonChild}(v, z, t)$:
1. $m \leftarrow$ copy of label of $z$
2. $X \leftarrow$ element corresponds to vnode $v'$
3. $(u_a, u_b) \leftarrow$ vertices incident with edge $e_X$
4. if $t = \text{true}$ then
   5. if $m[a] = C$ or $m[b] = C$ then return $\perp$
   6. $m[a] \leftarrow C$, $m[b] \leftarrow C$
   7. for $u_i \in F(v) \setminus F(v')$ do
      8. if $m[i] = R$ then return $\perp$
      9. else $m[i] \leftarrow F$
   10. if $v' \neq \text{leaf node}$ then return $m$
   11. else
      12. $Y \leftarrow$ the element corresponds to $v'$
      13. $(u_i, u_k) \leftarrow$ vertices incident with edge $e_Y$
      14. if $m[a] = C$ or $m[b] = C$ then return $\perp$
      15. else $m[a] \leftarrow R$ or $m[b] = R$ then return $Y$
      16. else return $\perp$

function $\text{decompChild}(v, z)$:
19. $\text{elems} \leftarrow \emptyset$
20. $\text{common} \leftarrow F(v') \cap F(v)$
21. $m_p, m_s \leftarrow$ copies of the label of $z$
22. for $u_i \in F(v') \setminus F(v)$ do $m_p[i] \leftarrow F$
23. for $u_i \in F(v) \setminus F(v')$ do $m_s[i] \leftarrow F$
24. if $\text{common} = \emptyset$ then return $\{(m_p, m_s)\}$
25. for $u_i \in \text{common}$ do
26. if $m_p[i] \neq C$ then $\text{combs}[i] \leftarrow \{(C, C)\}$
27. else if $m_p[i] = U$ then
      28. $\text{combs}[i] \leftarrow \{(R, C), (C, U)\}$
29. else if $m_p[i] = R$ then $\text{combs}[i] \leftarrow \{(R, C), (C, R)\}$
30. for $vals \in \text{enumerateCombination}($combs$)$ do
31. $m_p'[i] \leftarrow$ copy of $m_p$, $m_s'[i] \leftarrow$ copy of $m_s$
32. for $u_i \in \text{common}$ do
33. $(m_p'[i], m_s'[i]) \leftarrow$ vals[i]
34. $\text{elems} \leftarrow \text{elems} \cup \{(m_p'[i], m_s'[i])\}$
35. return $\text{elems}$

$G = (V, E)$ and a vnode as its input, and returns a ZSDD representing the set of all substructures. $Z[v]$ is a table storing decision nodes that respect vnode $v$. Since a ZSDD is represented as a set of decision nodes, 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 node with its label. The procedure differs when we construct different substructures. Next the algorithm calls construct($v, Z$), which recursively constructs nodes that respect vnode $v$ and its descendant vnodes. Procedure reduce($Z$) recursively deletes and merges nodes 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 $t \in \{\text{true, false}\}$. This procedure creates $m$ that is either the label of a child of $z$ that respects $v'$ or a terminal vnode. unique($m$) takes $m$ as the input, and returns $m$ if $m$ is a terminal vnode. Otherwise it checks whether there already exists a znode respecting $v'$ and has the same label in $Z[v']$. If such a znode exists, the procedure returns the address of the znode. Otherwise it creates a new decision znode that respects $v'$ and has label $m$, stores it in $Z[v']$ and returns the address of the znode. If neither $m_f$ nor $m_t$ are $\perp$ or $m_f \neq m_t$, we set $(X, Y)$-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 nodes that form primes and subs. Finally, we set elements as children of znode $z$, and recursively call construct($v, Z$) for $v = v'$ and $v = v'$ (line 21). If a right-linear tree 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
SimPath by introducing \texttt{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 \texttt{rootNode()}, \texttt{shannonChild}(v, z, t), and \texttt{decompChild}(v, z) appropriately for the target substructure. We first show concrete procedures used for constructing the set of all matchings. Procedure \texttt{rootState()} returns \( m \) where \( m[i] = U \) for every \( u_i \in V \), since no gnodes incident edges in the initial state. Alg. 3 shows \texttt{shannonChild}(v, z, t) and \texttt{decompChild}(v, z). Procedure \texttt{shannonChild}(v, z, t) updates the label of znode \( z \) to make labels of its child nodes. If \( t = \text{true} \), then the procedure updates labels by adding edge \( X \) corresponding to leaf vnode \( v^t \). 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 \( \bot \) (line 6). Otherwise, we update \( m[a] \) and \( m[b] \). We then set \( m[i] = F \) for every \( u_i \in F(v^t) \) that will not appear in \( F(v^r) \) (line 8-10). If \( m[i] = R \) for some \( u_i \in F(v^t) \setminus F(v^r) \), it returns \( \bot \) since the condition that \( u_i \) must incident an edge will not be satisfied (line 9). If \( v^t \) is not a leaf node, we finish the procedure and return \( m \) (line 11). If \( v^t \) is a leaf, the procedure returns the terminal ZSDD node according to the states of frontier nodes (line 12-18).

Procedure \texttt{decompChild}(v, z, t) is simple if \( v^t \) 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 \( m_p[i] \leftarrow F \) and \( m_s[i] \leftarrow F \) for all gnodes \( u_i \) that do not appear in \( F(v^t) \) and \( F(v^r) \), respectively (line 24, 25). Finally, it returns the pair \((m_p, m_s)\) (line 26). If \( v^t \) 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 \( u_i \in F(v^t) \cap F(v^r) \), and \( m[i] = U \). Then \( u_i \) can incident at most one edge that is either in sub-vtree \( v^t \) or \( v^r \). If such edge is in \( v^t \), then \( m_p[i] = C \) since no edge in \( v^t \) incidents \( u_i \), otherwise \( m_s[i] = C \). 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 \( m_p[i] = U \), it contains cases in which no edge in \( v^t \) incidents \( u_i \). Such cases may also occur when \( m_p[i] = C \). In this way, we store all possible assignments on \((m_p[i], m_s[i])\) for all \( u_i \in F(v^t) \cap F(v^r) \) in comb (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 \texttt{enumerateCombination} enumerates all possible combinations of pairs of gnode states stored in comb, and every vals contains pairs of \((m_p[i], m_s[i])\) for all \( u_i \in \text{common} \).

**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 \((U, U, U, U)\) by using \texttt{rootState()}. The procedure then calls \texttt{construct}(v1, z1). Since \( v_1 \) is a Shannon vnode, the procedure calls \texttt{shannonChild}(v1, z1, t) for different values of \( t \in \{\text{true}, \text{false}\} \). Since \( u_1, u_2 \) are connected with \( e_A, m_f = \left( (U, U, U, U) \right) \text{ and } m_t = (C, C, C, C) \). Thus, the root znode has two child elements \((\varepsilon, m_f)\) 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 \( m_f = m_t \) and \((m_f, z_1)\) and \( (m_f, z_2) \).

Next, \texttt{construct}(v_2, v_3) is called. Since \( v_2 \) is a decomposition vnode, procedure \texttt{decompChild}(v_2, v_3) is called for znodes \( z_1, z_2 \). Here \( v_1 = v_3 \) and \( v_1 = v_4 \), and common frontier gnodes of left and right child vnodes are \( F(v_1) \cap F(v_4) = \{u_1, u_2, u_3\} \). \( z_1 \) has four possible labels of prime child nodes \((U, C, C, C), (U, C, R, F), (U, R, R, F), \) and \((U, R, C, F)\), and these prime labels form pairs with subs \((F, U, U, U),(F, U, C, U), (F, C, C, U), (F, C, U, U)\). 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 \( v_2 \) has two possible labels of prime child znodes \((C, C, R, F)\) and \((C, C, C, F)\). Corresponding sub child znodes are \((F, C, C, U)\) and \((F, C, C, U)\). 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 \((U, C, C, F)\), and \( F(v_3) = \{u_1, u_2, u_3\} \). Since \( v_3 \) is a Shannon vnode, procedure \texttt{construct}(v_3, Z) calls \texttt{shannonChild}(v_3, z_3, t) with \( t \in \{\text{true}, \text{false}\} \). If \( t = \text{false} \), the procedure returns element \((\varepsilon, \varepsilon)\) since \( v^t \) is a leaf vnode and the corresponding edge \( e_C \) cannot be taken since both \( m[2], m[3] = C \). If \( t = \text{true} \), the procedure returns \( \bot \) since \( m[3] = C \) 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 \( \alpha \) is the ZSDD representing the set of all matchings obtained by our top-down construction algorithm, the size of \( \alpha \) is \( O(|E|^2 W^2) \), where \( W \) is the width of vtree and is defined as \( W = \max_v |F(v^t)| \).

**Proof.** The number of decision znodes that respect vtree node \( v \) is bounded by the number of possible frontier pat-
lem, 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, select a frontier node, and select a frontier node, then the set of possible values of m[2] is either \{R, C\} or \{U, C\}. Hence the number of decision znodes are bounded by \(|E|^2\). Since every decision znode has at most 2\(^W\) child element nodes, the ZSDD size is \(O(|E|2^{bw})\).

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 \(O(|E|2^{bw})\). 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 ZDDs with a limitation that vtrees must be right-linear. Since a ZDD respecting a right-linear vtree is equivalent to a ZDD, the algorithm is equivalent to SimPath for ZDDs. For ZSDDs, 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 graphilion (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-

---

1https://github.com/nsnmsak/zsdd
2http://www.graphdrawing.org/download/rome-graphml.tgz
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.

**Conclusion**

We showed 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 nodes as labels. Labels represent the connection relations between frontier nodes. Fig. 5 shows two equivalent frontier patterns with frontier nodes $u, v, z, t$. These two examples use different paths between $u_4$, $u_5$, and $u_6$. In this example, $u_4$ is connected to $u_1 = s$, and $u_5$ and $u_6$ are terminal nodes 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 $\text{shannonChild}(v, z, t)$ and $\text{decompChild}(v, z)$ for constructing ZSDDs representing the sets of all simple paths. We first overview these subroutines, then show the concrete design of labels and algorithms. $\text{shannonChild}(v, z, t)$ updates the label of znode $z$ by selecting or not selecting the edge corresponding to vnode $v'$. In the same way as for matchings, $\text{decompChild}(v, z)$ 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 nodes $(F(v') \cap F(v'))$, and then (b) enumerates all
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 vnode \(u_i \in F(v^i) \cap F(v^s)\).

We elucidate procedure (b) by using the example shown in Fig. 6. Suppose that \(F(v)\) for vnode \(v\) is \(\{u_1, \ldots, u_7\}\), \(F(v^1) = \{u_1, \ldots, u_5\}\) and \(F(v^s) = \{u_5, u_6, u_7\}\). 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, \((u_2, u_7), (u_4, u_6),\) and \((u_1, u_5)\). These connections between prime and sub frontiers prevent top-down construction at prime and sub ZSDDs from running independently. For example, source vnode \(s\) is currently connected to \(u_1 \in F(v^1)\), but target vnode \(t\) is contained only in the subgraph \(G_s\) that is induced by edges in vtree \(v^s\). 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 \(F(v^s)\). 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 vnodes and corresponding sub frontier vnodes, given the connection pattern of Fig. 6 (a). There are three possible choices of 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 \(v^s\) proceeds independently. The top-down construction procedure for \(v^s\) 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.

Next we show the concrete procedures. We first show how to represent connection patterns between frontier nodes. In the following, we assume \(s = u_1\) and \(t = m[v]\) 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 vnode \(u_i \in V\) 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 vnode incidents no edges. If \(m[i] = j\) where \(1 \leq j \leq |V|\) and \(j \neq i\), \(i\) is a terminal vnode 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 \(u_1|V| = t\), \(s\) 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-
The algorithm begins by checking if the copy of $m_p$ is empty (line 15). If it is, the algorithm returns the current elements (line 16). Otherwise, it proceeds to enumerate the possible assignments (line 17) and for each assignment (line 18), it checks if the connection is empty (line 19). If the connection is empty, it adds the current elements to a new set (line 20). It then enumerates all possible patterns (line 21) and for each pattern (line 22), it updates the elements (line 23). The algorithm returns the updated elements (line 24).

The algorithm then checks if the $m_p$ label is a terminal node (line 25). If it is, it adds the current elements to the new set (line 26). Otherwise, it repeats the process for the next element (line 27).

The key idea of this algorithm is to enumerate all possible assignments and check if they form a valid connection. This is done by checking if the connection is empty and if so, adding the current elements to a new set. The algorithm then recursively calls itself for the next element until all elements have been processed.

Algorithm 5: decompChild($v, z$)

1. $\text{elems} \leftarrow \emptyset$
2. $\text{common} \leftarrow F(v') \cap F(v'')$
3. $m_p \leftarrow \text{label of } z$, $m_a \leftarrow \text{label of } z$
4. for $u_i \in F(v') \setminus F(v'')$ do $m_p[i] \leftarrow 0$
5. for $u_i \in F(v'') \setminus F(v')$ do $m_a[i] \leftarrow 0$
6. for $u_i \in \text{common}$ do
7. if $m_p[i] = 0$ then
8. $\text{combs}[i] \leftarrow \{(0, 0)\}$
9. else if $m_p[i] = i$ then
10. $\text{combs}[i] \leftarrow \{(-i, 0), (0, i), (\pi, \pi)\}$
11. else if $m_p[i] = -i$ then
12. $\text{combs}[i] \leftarrow \{(-i, 0), (0, -i), (\pi, \pi)\}$
13. else
14. $\text{combs}[i] \leftarrow \{(m[i], 0), (0, m[i])\}$
15. for $v_i \in \text{enumerateCombination(combs)}$ do
16. $m_p' \leftarrow \text{copy of } m_p$, $m_a' \leftarrow \text{copy of } m_a$
17. for $u_i \in \text{common}$ do
18. $(m_p'[i], m_a'[i]) \leftarrow v_i$
19. $\text{connection} \leftarrow \text{connections between } m_p'$ and $m_a'$
20. if $\text{connection}$ is empty then
21. $\text{elems} \leftarrow \text{elems} \cup \{(m_p'[i], m_a'[i])\}$
22. else
23. for $cVals \in \text{enumeratePats(connection)}$ do
24. $m_p'' \leftarrow m_p'$, $m_a'' \leftarrow m_a'$
25. $\text{Update } m_p'', m_a'' \text{ so as to follow } cVals$
26. $\text{elems} \leftarrow \text{elems} \cup \{(m_p'', m_a'')\}$
27. return $\text{elems}$

References


