Compact Complete Inverted Files for Texts and Directed Acyclic Graphs Based on Sequence Binary Decision Diagrams

Shuhei Denzumi[1], Koji Tsuda[2,3], Hiroki Arimura[1], and Shin-ichi Minato[1,3]

[1] Graduate School of IST, Hokkaido University, Sapporo, Japan
[2] AIST Computational Biology Research Center, Tokyo, Japan
[3] JST ERATO MINATO Discrete Structure Manipulation System Project, Sapporo, Japan
{denzumi, arim, minato}@ist.hokudai.ac.jp, koji.tsuda@aist.go.jp

Abstract. A complete inverted file is an abstract data type that provides functions for text retrieval. Using it, we can retrieve frequencies and occurrences of strings for given texts. There have been various complete inverted files for texts. However, complete inverted files for graphs have not been studied. In this paper, we define complete inverted files based on sequence binary decision diagrams (SDD) for directed acyclic graphs (DAG). Directed acyclic graphs are given as sequence binary decision diagrams. We propose new complete inverted files called PosFSDD and PosFSDDdag for a text and a DAG, respectively. We also present algorithms to construct them and to retrieve occurrence information from them. Computational experiments are executed to show the efficiency of PosFSDDs.

1 Introduction

Recent emergence of massive text and sequence data has increased the importance of string processing technologies. In particular, complete inverted files for efficient text retrieval and analysis has attracted much attention in many applications such as bioinformatics, natural language processing, and sequence mining. A complete inverted file for a text w is a data structure that stores all factors of w allowing three functions; *find*, *freq*, and *locations*. In many real applications, indices that store occurrence information are highly required. Sequence binary decision diagrams (sequence BDDs or SDDs, for short) are compact representation for manipulating sets of strings, proposed by Loekito, *et al.* [7]. In this paper, we consider the problem of constructing a complete inverted file on SDD framework. We define complete inverted files on SDDs, named PosFSDD (See Fig. 2), and propose an algorithm to construct a PosFSDD from an input text. We also define a complete inverted file for a directed acyclic graph (DAG) and present an efficient construction algorithm to construct a PosFSDDdag from an input DAG, which is given as an SDD. There is research on construction factor automata from automata [10]. On the other hand, complete inverted files for graphs have not been studied. We can construct complete inverted files for multiple texts by concatenating them on existing data structures. However, those methods cannot deal with very large number of strings such that DAGs can represent by sharing its subgraphs. For example, regular expressions without infinite loop and human genomes with many replacements can be represented much more compactly by DAGs than by explicit representations. We also show some experimental results for real data. Our method will be useful for wide variety of pattern matching applications and sequence mining.

Shuhei Denzumi, Koji Tsuda, Hiroki Arimura, Shin-ichi Minato: Compact Complete Inverted Files for Texts and Directed Acyclic Graphs Based on Sequence Binary Decision Diagrams, pp. 157–167.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0 © Czech Technical University in Prague, Czech Republic

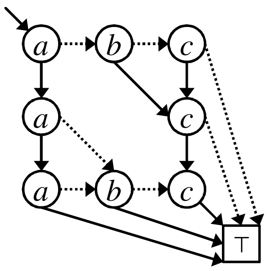
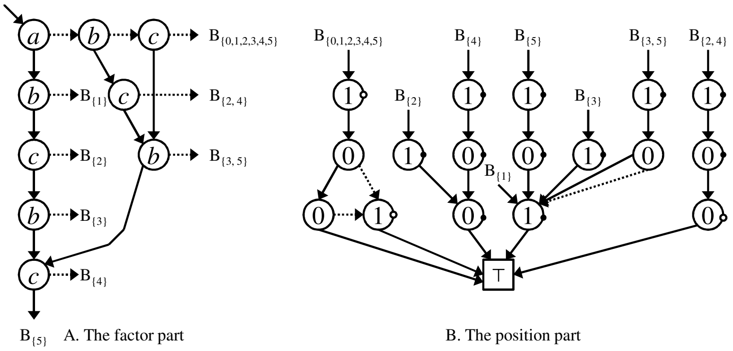


Figure 1. An SDD for the language $\{\epsilon$, *aaa*, *aab*, *aac*, *ab*, *ac*, *b*, *bcc*, *c*, *ccc*$\}$. Circles denote non-terminals. Squares denote terminals. The 0-terminal $\perp$ and 0-edges coming to $\perp$ are omitted.



A. The factor part

B. The position part

Figure 2. An example of a complete inverted file based on SDD, PosFSDD, for $w = abcbc$. The 0-terminal $\perp$ is omitted. All 0-edges coming to $\perp$ and $\top$ are indicated by a small black dot and white dot on the right side, respectively.

2 Preliminaries

2.1 Strings and string sets

Let $\Sigma = \{a, b, \ldots\}$ be a countable alphabet of symbols, for which the equality $=_\Sigma$ and a strict total order $\prec_\Sigma$, such that $a \prec_\Sigma b \prec_\Sigma \cdots$, are defined on Σ. We often omit the subscript Σ if no confusion arises. A *string* on Σ of length $n \geq 0$ is a sequence $s = a_1 \cdots a_n$ of symbols, where $|s| = n$ is called the *length* of s and for every $i = 1, \ldots, n$, $s[i] = a_i \in \Sigma$ is called the i-th symbol of s for $1 \leq i \leq |s|$.

Let ϵ be the *empty string* of length zero, and Σ^* be the set of *all possibly empty finite strings*. For strings $x = a_1 \cdots a_m$ and $y = b_1 \cdots b_n$, we define the *concatenation* of x and y by $x \cdot y = xy = a_1 \cdots a_m b_1 \cdots b_n$. For any symbol $\alpha \in \Sigma$, let $\alpha \cdot L = \{\alpha\} \cdot L = \{\, \alpha x \mid x \in L \,\}$. We denote the reversed string of x by $x^R = x[|x|] \cdots x[1]$. For a string s, if $s = xyz$ for $x, y, z \in \Sigma^*$, then we call x, y, and z a *prefix*, a *factor*, and a *suffix* of s, respectively. The sets of prefixes, factors, and suffixes of a string s are denoted by $\mathit{Prefix}(s)$, $\mathit{Factor}(s)$, and $\mathit{Suffix}(s)$, respectively. Given a set S of strings, let the sets of prefixes, factors, and suffixes of the strings in S be denoted by $\mathit{PREFIX}(S)$, $\mathit{FACTOR}(S)$, and $\mathit{SUFFIX}(S)$, respectively.

For any $x \in \mathit{Factor}(w)$, $\mathit{epos}_w(x)$ denotes the set of all positions in w immediately following the occurrences of x and $\mathit{bpos}_w(x)$ denotes the set of all positions immediately preceding occurrences of x. We denote binary representation of an integer i by $\mathit{binstr}(i) \in \{\mathit{0}, \mathit{1}\}^*$ where the leading *0*s are omitted. Therefore, $\mathit{binstr}(0) = \epsilon$. If $\mathit{0}, \mathit{1} \in \Sigma$, $a \prec \mathit{0} \prec \mathit{1}$ for any symbol $a \in \Sigma$.

2.2 Finite Automata

We presume a basic knowledge of the automata theory. For comprehensive introduction to it, see [5,11] for example. A (partial) *deterministic finite automaton DFA* is represented by a quintuple $A = \langle \Sigma, \Gamma, \delta, q_0, F \rangle$, where Σ is the input alphabet, Γ is the *state set*, δ is the *partial transition function* from $\Gamma \times \Sigma$ to Γ, $q_0 \in \Gamma$ is the *initial state* and $F \subseteq \Gamma$ is the set of *acceptance states*. The partial function δ can be regarded as a subset $\delta \subseteq \Gamma \times \Sigma \times \Gamma$. We define the *size* of a DFA A, denoted by $|A|$, as the number of labeled edges in A, i.e., $|A| = |\delta|$.

The set of strings that lead the automaton A from a state q to an acceptance state is denoted by $L_A(q)$. The *language* $L(A)$ accepted by A is $L_A(q_0)$. We say that ADFAs A and A' are *equivalent* if $L(A) = L(A')$. A *minimal DFA* has no state q such that $L_A(q) = \emptyset$ and no distinct states q' and q'' such that $L_A(q') = L_A(q'')$. Since we are concerned with finite languages, all DFAs discussed in this section are *acyclic* DFAs (ADFA, for short).

2.3 Sequence binary decision diagrams

In this subsection, we briefly give a formalization of sequence BDDs, introduced by Loekito, Bailey, and Pei [7], and related concepts for further discussion. Let $\mathsf{dom} = \{u, v, v_1, v_2, \ldots\}$ be a countable set, where each element is called a *node*, and let Σ be a countable alphabet with which a strict total order $\prec_\Sigma$ is associated. A *labeled binary directed acyclic graph* (*labeled binary DAG*) is a directed acyclic graph (DAG) in which every node has out-degree either zero (terminal) or two (non-terminal), where each non-terminal node has a pair of distinguished edges called the 0-edge and the 1-edge . We call the nodes pointed to by the 0- and 1-edges the 0-child and the 1-child, respectively. We define the *subgraph of S rooted at node v* by the connected subgraph of S reachable from v and denote it by $S(v)$.

Roughly speaking, a *sequence binary decision diagram* [7] on Σ is a node-labeled binary DAG that encodes an acyclic DFA on Σ in the *leftmost child and right-sibling* (*LCRS*, for short) *representation* (see, e.g., [1,6]), where the 0-child and 1-child of a non-terminal node correspond to its leftmost child and the right sibling, respectively. Formally, sequence binary decision diagram is defined as follows.

Definition 1. *Let Σ be an alphabet. A sequence binary decision diagram (a* sequence BDD*, for short) is a DAG $S = \langle \Sigma, V, \tau, \bot, \top, \boldsymbol{r} \rangle$ satisfying the following conditions:*

- *$V = V(S) \subseteq \mathsf{dom}$ is a finite set of nodes and every node has unique ID,*
- *$\boldsymbol{r} \in V$ is a distinguished node called the* root *of S.*
- *$\bot$ and $\top \in V$ are distinguished nodes called the* 0- *and* 1-terminal*, respectively. The nodes in $V_N = V \backslash \{\bot, \top\}$ are called* non-terminals.
- *$\tau : V_N \to \Sigma \times V^2$ is the function that assigns to each $v \in V_N$ the triple $\tau(v) = \langle v.lab, v.0, v.1 \rangle$, called the* node triple *for v. Then, the triple indicates that (i) $v.lab \in \Sigma$ is the label of v, (ii) $v.0 \in V$ is the child, called the* 0-child*, that is pointed to by a* 0-edge *from v, and (iii) $v.1 \in V$ is the child, called the* 1-child*, that is pointed to by a* 1-edge *from v.*
- *S must be* acyclic *in its* 0- *and* 1*-edges, that is, there exists some strict partial order $\succ_V$ on V such that for any $v \in V_{\mathrm{N}}$, both of $v \succ_V v.0$ and $v \succ_V v.1$ hold.*
- *S must be* 0-ordered*, that is, for every non-terminal node v, if $v.0$ is a non-terminal node then $v.lab \prec_\Sigma (v.0).lab$ must hold. This means that siblings are deterministically ordered from left to right by $\prec_\Sigma$ on their labels when S is interpreted as an acyclic DFA in the LCRS representation.*

In the figures of this paper, the terminals/nonterminals are denoted by squares/circles, and the 0/1-edges are denoted by dotted/solid lines.

In the above definition, S is said to be *well-defined* if it is both acyclic and 0-ordered. We define the *size* $|S|$ of S by the number of non-terminals in S, i.e., $|S| = |V_{\mathrm{N}}| = |V| - 2$. In the rest of this paper, we often abbreviate a sequence BDD as *SDD* if no confusion arises.

An important class of sequence BDDs is that of reduced sequence BDDs [7], which is a syntactic normal form of SDDs defined as follows.

Definition 2 (reduced SDD [7]). *An sequence BDD is said to be* reduced *if it satisfies the following two conditions:*

- Node-sharing rule*: For any non-terminal nodes $u, v \in V_N$, $\tau(u) = \tau(v)$ implies $u = v$, i.e., no distinct non-terminal nodes have the same triple.*
- Zero-suppress rule*: For any non-terminal nodes $v \in V_N$, $v.1 \neq \bot$ holds, i.e., no non-terminal node has the* 0*-terminal as its* 1*-child.*

The above two rules were originally introduced by Minato [8] for ZDDs [6]. A sequence BDD S defines its language $L(S)$ in the following way. The language of a sequence BDD S is the language assigned to its root $\mathbf{r}$.

Definition 3 (language). *To each node $v \in V$, we inductively assign a language $L_S(v)$ w.r.t. $\succ_V$ as follows: (i) $L_S(\bot) = \emptyset$; (ii) $L_S(\top) = \{\varepsilon\}$; (iii) $L_S(v) = L_S(v.0) \cup (v.lab) \cdot L_S(v.1)$.*

In Fig. 1, we show an example of SDD for the language $\{\epsilon, aaa, aab, aac, ab, ac, b, bcc, c, ccc\}$.

In sequence BDD environment, we can create a new subgraph by combining one or more existing subgraphs in an arbitrary way. As an invariant, all subgraphs are maintained as minimal. In the environment, We use two hash tables *uniqtable* and *cache*, explained below. The first table *uniqtable*, called the unique node table, assigns a nonterminal node $v = \mathit{uniqtable}(c, v_0, v_1)$ to a given triple $t = \langle c, v_0, v_1 \rangle$ of a symbol c and a pair of nodes v_0 and v_1. This table is maintained such that it is a function from all triples $t \in \Sigma \times V^2$ to the nonterminal node v in V such that $\tau(v) = t$. If such a node does not exist, *uniqtable* returns *null*. We define a procedure Getnode(c, v_0, v_1) that returns a node with the triple $\langle c, v_0, v_1 \rangle$. If there is such a node in V, Getnode returns it. Otherwise, it creates such a node and returns it. The Getnode checks the two reduction rules by using the *uniqtable* to avoid creating duplicated nodes. The second table *cache*, called the operation cache, is used for a user to memorize the invocation pattern "$op(v_1, \ldots, v_k)$" of a user-defined operation *op* and the associated return value $u = op(v_1, \ldots, v_k)$, where each v_i, $i = 1, \ldots, k$ is an existing node in V.

For two given SDDs P and Q, we can compute a SDD R such that R is the language obtained from primitive set operations, union, intersection and difference, on the languages $L(P)$ and $L(Q)$ by recursive algorithms [4]. In addition, concatenation of languages can be computed by Concat in Fig. 3. Using these algorithms, we can construct SDDs for sets of strings of exponential size such as regular expressions without infinite repeats.

2.4 Complete Inverted File

The notion of an inverted file for a textual database is common in the literature on information retrieval, but precise definitions of this concept vary. We use the following definition. Given a finite alphabet Σ, and a text word $w \in \Sigma^*$, a complete inverted file for (Σ, w) is an abstract data type that implements the following functions:

- (1) *find*: $\Sigma^* \to \mathit{Factor}(w)$, where *find*$(x)$ is the longest prefix y of x such that $x \in \mathit{Factor}(w)$ and y occurs in w, that is, $x = yz$, $x, y, z \in \Sigma^*$, and y is a factor of a text w.

```
Global variable: $uniqtable, cache$: hash tables.

Proc Concat($P, Q$: SBDD):
Return: $R$: SDD;
if ($P = \bot$ or $Q = \bot$) return $\bot$;
else if ($P = \top$) return $Q$;
else if ($Q = \top$) return $P$;
else if (($R \leftarrow cache[$“$Concat(P, Q)$”$]$) exists) return $R$;
else
    $\langle x, P_0, P_1 \rangle \leftarrow \tau(P)$;
    $R \leftarrow$ Getnode($P.lab, \bot,$ Concat($P_1, Q$));
    $R \leftarrow$ Union($R,$ Concat($P_0, Q$));
    $cache[$“$Concat(P, Q)$”$] \leftarrow R$;
    return $R$;
```

Figure 3. An algorithm Concat that constructs the SDD for the language $L(P) \cdot L(Q)$, for given SDDs P and Q.

- (2) *freq*: $Factor(w) \to \mathbb{N}$, where $freq(x)$ is the number of times x occurs as a factor of the text w.
- (3) *locations*: $Factor(w) \to \mathbb{N}^*$, where $locations(x)$ is the set of end positions within the text in which x occurs.

In this paper, we consider the problem of constructing a complete inverted file for a text w. The function $locations(x)$ returns the SDD that represents set of integers $epos_w(x)$ as set of binary strings in our method. We describe SDDs can implement complete inverted files compactly.

Example 4. Let $w = abaababa$ be a given text. Then, $find(baabbaab) = baab$, $freq(ba) = 3$, and $locations(ba) = \{3, 6, 8\}$.

3 Position Factor SDD

We begin with a brief look at some aspects of the factor structure of a fixed, arbitrary word w. In particular, for each factor x of w we will be interested in the set of positions in w at the ends of occurrences of x. We describe the basic data structure used to implement a complete inverted file for a text w based on an SDD.

In our method, occurrence positions are represented as a set of binary strings instead of simple a list of integers. If a factor x occurs at position i, our inverted file stores $x \cdot binstr(i)$. That is, a factor x of w is followed by its occurrence positions in the complete inverted file. Then, we can know the occurrences of x after traversing the path corresponding to x. All equivalent subgraphs are online minimized automatically by always using Getnode when a node with some triple is needed. Therefore, the subgraphs which represent binary strings also share their equivalent subgraphs and become compact.

Definition 5. *Let w be any string. Then, we define two languages.*

- $\mathcal{L}_{epos}(w) = \{x \cdot binstr(k) : x \in Factor(w), k \in epos_w(x)\}$,
- $\mathcal{L}_{bpos}(w) = \{x^R \cdot binstr(k) : x \in Factor(w), k \in bpos_w(x)\}$.

Definition 6. *The Position Factor SDD (PosFSDD) of $w \in \Sigma^*$ is the SDD $F = \langle \Sigma \cup \{0, 1\}, V, \tau, \bot, \top, \boldsymbol{r} \rangle$ such that $L(\boldsymbol{r}) = \mathcal{L}_{epos}$.*

The PosFSDD for $w = abcbc$ is given in Figure 2. Note that the SDDs that represent binary strings play a role analogous to the identification pointers in the compact DAWG [3].

Theorem 7. *Using PosFSDD* $F = \langle \Sigma \cup \{0,1\}, V, \tau, \bot, \top, \boldsymbol{r} \rangle$ *for a word* $w \in \Sigma^*$, *for any word* $x \in \Sigma^*$, $y = \mathit{find}(x)$ *can be determined in time* $O(|\Sigma||x|)$. *For any* $x \in \mathit{Factor}(w)$, $\mathit{freq}(x)$ *can be determined in time* $O(|\Sigma||x|)$ *if* $\mathsf{Card}(\boldsymbol{r})$ *is already executed at least once.*

Proof. To implement *find*, we begin at the root **r** and trace a path corresponding to the letters of x as long as possible. This "search path" is determined and continues until the longest prefix y of x in $\mathit{Factor}(w)$ has been found. To implement *freq*, we note that $\mathit{freq}(x) = |\{z : xz \in \mathcal{L}_{epos}(w)\}| = |\mathit{epos}_w(x)|$ for any $x \in \mathit{Factor}(w)$. The algorithm Card computes the cardinality of the language that each SDD node represents and stores each result in *cache* [6]. So, $\mathit{freq}(x)$ can be obtained by following the procedure of *find* and then returning the result of Card of the node stored in the *cache*. $\mathsf{Card}(\boldsymbol{r})$ is executed in linear time to the input SDD size. Since this node represents the language $M = \{z : xz \in \mathcal{L}_{epos}(w)\}$, we can obtain the node that represents $\{b : b \in M, b \in \{\mathit{0}, \mathit{1}\}^*\}$ by traversing 0-edges until getting a node labeled by *0* or *1*. Clearly all queries are $O(|\Sigma||x|)$. □

Our algorithm to construct PosFSDD is described in Fig. 4. The union operation is computed in $O(|P||Q|)$ time for two SDDs P and Q [4]. In fig. 5, we shows the algorithm BinSDD(k) that constructs an SDD that represents a binary representation of a natural number k. That is, $L(\mathsf{BinSDD}(k))$ is $\{\mathit{binstr}(k)\}$. We can also construct an SDD for $\mathcal{L}_{bpos}$ with some modification of BuildPosFSDD. That is swapping $|w|$ with 0 in line 1 and line 5, and changing the for loop in line 2 from descending order $|w|, \ldots, 1$ to ascending order $1, \ldots, |w|$.

For a given text w and its factor x, it takes $O(\mathit{freq}(x) \log w)$ time to compute occurrence list of x after obtaining the SDD for $\mathit{locations}(x)$, because occurrences are represented as binary strings and every node has just one label. On the other hand, there are advantages due to SDD representation, especially when $\mathit{freq}(x)$ is large. A list of integers in ordinary representation requires $O(freq(x))$ space and time to examine all positions. By sharing structures, these positions can be represented compactly in our method. As a result, execution times for various operations are improved. For example, for given two factors x and y, finding the positions that both occur within l symbols is computed with some modifications. At first, we construct SDD for $\mathcal{L}'_{epos}(w) = \{x \cdot \mathit{binstr}(k+j) : x \in \mathit{Factor}(w), k \in \mathit{epos}_w(x), 0 \leq j \leq l\}$. Next, obtain the SDDs for $\mathit{locations}(x)$ and $\mathit{locations}(y)$. Then, the positions we want are computed by the intersection operation of these two SDDs.

4 Position FSDD for SDD

We now show our algorithm that constructs a complete inverted file for a directed acyclic graph given as an SDD. First we note that the complete inverted file for an SDD S is defined as follows. In our method, we use node identifiers (IDs) instead of positions for ordinary texts, and factors correspond to paths in the input SDD.

Given an SDD S, a complete inverted file for S for it is an abstract data type that implements the following functions:

```
Global variable: $uniqtable, cache$: hash tables.
Proc BuildPosFSDD($w$: string):
Return: $F$: PosFSDD;
$P_{|w|} \leftarrow$ BinSDD($|w|$), $F_{|w|} \leftarrow P_{|w|}$;
for $i = |w|, \dots, 1$
    $P_{i-1} \leftarrow$ Getnode($w[i]$, BinSDD($i-1$), $P_i$);
    $F_{i-1} \leftarrow$ Union($F_i, P_i$);
return $F_0$;
```

Figure 4. An algorithm BuildPosFSDD for constructing the PosFSDD of an input string w.

```
Global variable: $uniqtable, cache$: hash tables.
Proc BinSDD($k$: natural number):
Return: $B$: SDD such that $L(B) = \{binstr(k)\}$;
return BinSDD0($k, \lfloor \log_2(k+1) \rfloor$);
Proc BinSDD0($k, l$: natural number):
Return: $B$: SDD that $L(B) = \{l$ length binary string of $k\}$;
if ($l = 0$) return $\top$;
else if ($B \leftarrow cache[$"$BinSDD(k,l)$"$]$ exists) return $B$;
else
    if ($k\&(1 << l) \neq 0$) $B \leftarrow$ Getnode($1, \bot$, BinSDD0($k\&((1 << l) - 1), l - 1$));
    else $B \leftarrow$ Getnode($0, \bot$, BinSDD0($k\&((1 << l) - 1), l - 1$));
    $cache[$"$BinSDD(k,l)$"$] \leftarrow B$;
return $B$;
```

Figure 5. An algorithm BinSDD for constructing the SDD for $\{binstr(k)\}$. Bitwise AND operation and bit left shift operaton are denoted by & and <<, respectively.

```
Global variable: $uniqtable, cache$: hash tables.
Proc AppendID($P$: SDD):
Return: $R$: SDD such that $L(R) = \{x \cdot binstr(P.ID) : x \in FACTOR(L(Q))$,
$P$ is a SDD node reachable from root via the path corresponding to $x$
 and traversing 0-edges$\}$;
if ($P = \bot$) return BinSDD(0);
else if ($P = \top$) return BinSDD(1);
else if ($R \leftarrow cache[$"$AppendID(P)$"$]$ exists) return $R$;
else
    $\langle x, P_0, P_1 \rangle \leftarrow \tau(P)$;
    $R \leftarrow$ Union(Getnode($x$, AppendID($P_0$), AppendID($P_1$)), BinSDD($P.id$));
    $cache[$"$AppendID(P)$"$] \leftarrow R$;
return $R$;
```

Figure 6. An algorithm AppendID for constructing the SDD with node IDs by binary strings.

- (1) *find*: $\Sigma^* \rightarrow FACTOR(L(S))$, where *find*$(x)$ is the longest prefix y of x such that $x \in FACTOR(L(S))$ and y occurs in $L(S)$, that is, y is a factor of a string in $L(S)$.

```
Global variable: $uniqtable, cache$: hash tables.
Proc BuildPosFSDDdag($S$: SDD):
Return: $F$: Position FSDDdag for $S$;
return BuildPosFSDDdag0(AppendID($S$));
Proc BuildPosFSDDdag0($P$: SDD):
Return: $G$: SDD such that $L(G) = \{z : z \in SUFFIX(L(P)), z \in \Sigma^+ \cdot \{0, 1\}^*\}$;
if ($P = \bot$ or $P = \top$) return $P$;
else if ($G \leftarrow cache[$“$BuildPosFSDDdag(P)$”$]$ exists) return $G$;
else
    $\langle x, P_0, P_1 \rangle \leftarrow \tau(P)$;
    if ($x \in \{0, 1\}$) return $P$;
    $G \leftarrow$ BuildPosFSDDdag0($P_0$) $\cup$ BuildPosFSDDdag0($P_1$) $\cup$ Getnode($x, \bot, P_1$) ;
    $cache[$“$BuildPosFSDDdag(P)$”$] \leftarrow G$;
return $G$;
```

Figure 7. An algorithm constructs the PosFSDDdag for the input SDD S. Union operations are denoted by $\cup$.

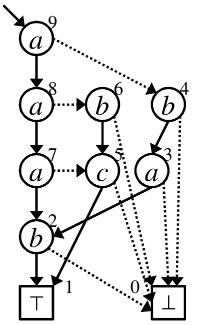


Figure 8. An SDD for $\{aaab, aac, abc, bab\}$. Node IDs are given on the side of each nodes.

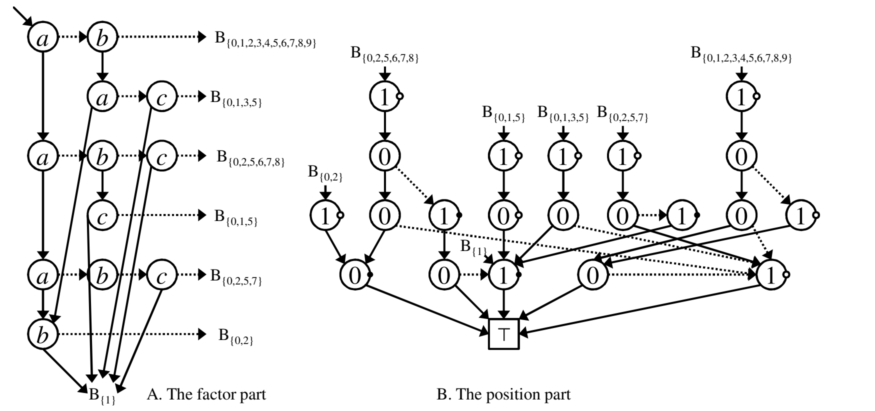


Figure 9. An example of a complete inverted file based on SDD, PosFSDDdag, for the SDD in Fig. 8. The 0-terminal $\bot$ is omitted. All 0-edges incoming to $\bot$ and $\top$ are indicated by a small black dot and white dot on the right side of a node, respectively.

- (2) *freq*: $FACTOR(L(S)) \to \mathbb{N}$, where $freq(x)$ is the number of nodes reachable by paths corresponding to x that begins from any nodes in S.
- (3) *locations*: $FACTOR(L(S)) \to \mathbb{N}^*$, where $locations(x)$ is the set of IDs of nodes in S to which paths lead that corresponding to x.

In our method, the set of node IDs that *locations* returns is represented by an SDD for the set of binary strings of the IDs.

Let S be an SDD. For any $x \in FACTOR(L(S))$, $enode_S(x)$ denotes the set of all IDs of nodes in S following the paths corresponding to x and traversing some 0-edges, $bnode_S(x)$ denotes the set of all IDs of nodes in S which represent a language M such that $x \in PREFIX(M)$.

Definition 8. *We define* $\mathcal{L}_{enode}(S) = \{x \cdot binstr(i) : x \in FACTOR(L(S)), i \in enode_S(x)\}$, *and* $\mathcal{L}_{bnode}(S) = \{x^R \cdot binstr(i) : x \in FACTOR(L(S)), i \in bnode_S(x)\}$. *The PosFSDDdag for S is the SDD G such that* $L(G) = \mathcal{L}_{enode}(S)$.

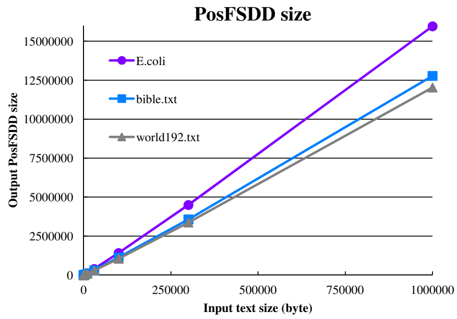


Figure 10. SDD size of PosFSDD with increasing length of input string.

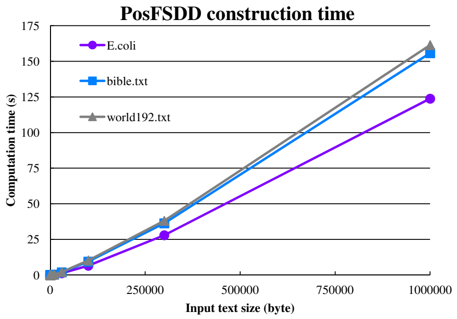


Figure 11. Computation time of BuildPosFSDD with increasing length of input string.

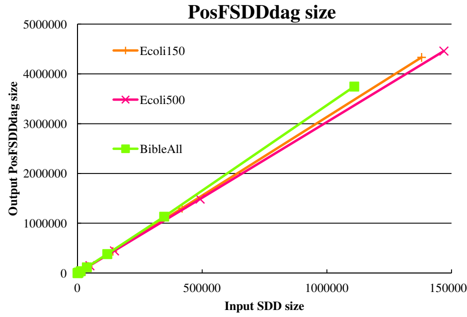


Figure 12. SDD size of PosFSDDdag with increasing input SDD size.

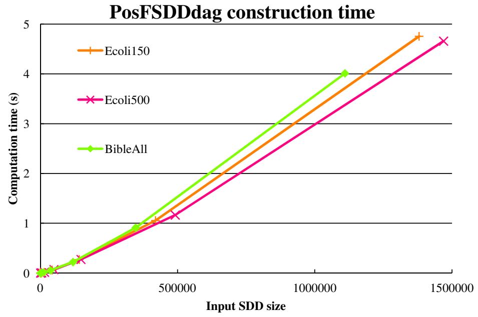


Figure 13. Computation time of BuildPosFSDDdag with increasing input SDD size.

The PosFSDDdag for an SDD S such that $L(S) = \{aaab, aac, abc, bab\}$ is given in Fig. 9, and Fig. 8 shows the input SDD.

Theorem 9. *Using PosFSDDdag G, for any word $x \in \Sigma^*$, $y = find(x)$ can be determined in time $O(|\Sigma||x|)$. For any $x \in FACTOR(L(S))$ can be determined in time $O(|\Sigma||x|)$.*

Proof. We can implement *find*, *freq* and *locations* as in PosFSDD for a text. □

Fig. 7 shows an algorithm to build the PosFSDDdag for an SDD S. The algorithm in Fig. 6 is used for prepocessing of PosFSDDdag. The basic action of the algorithm for an SDD S is to construct the PosFSDDdag for each node recursively, synchronized with the depth-first traversal of S. We can construct reversed version of the PosFSDDdag. It allows for the computation of the exact number of paths corresponding to queries. It also allows for returning the node IDs at which the paths begin. Such an SDD is constructed by executing BuildPosFSDDdag after appliying the algorithm that construct an SDD for reversed $L(S)$, which is proposed by Aoki *et al.* [2].

First, we append SDDs for node IDs to the input by AppendID. Next, we construct reversed SDD of it, but we do not reverse the SDDs that represent node IDs as binary strings. Then, we can construct the SDD for $\mathcal{L}_{bnode}(S)$ by execute BuildPosFSDDdag0 on the obtained SDD.

5 Experimental Results

Setting: In the experiments, we used the following data sets. As real data sets, we used `E.coli`, `bible.txt`, and `world192.txt` obtained from the Canterbury corpus[1].

[1] `http://corpus.canterbury.ac.nz/resources/`

From these data sets, we obtained the following derived data sets: BibleAll is the set of all lines drawn from `bible.txt`. Ecoli150 and Ecoli500 are the set of factors drawn from `E.coli` by cutting the whole sequence at every 150-th or 500-th letter, respectively. We made subsets of these data sets by randomly taking l lines varying $l = 10, 30, 100, \ldots$ for BibleAll, Ecoli150, and Ecoli500.

We implemented our shared and reduced SDD environment on the top of the *SAPPORO BDD package* [9] for BDDs and ZDDs written in C and C++, where each node is encoded in a 64-bit integer and a node triple occupies approximately 50 to 55 bytes on average including hash entries in *uniqtable*. We performed experiments on a machine that consists of eight quad-core 3.1 GHz Intel Xeon CPU E7-8837 SE processors (i.e, 32 CPU cores in total) and 1 TB DDR2 memory shared among cores. For PosFSDD and PosFSDDdag construction, we implemented BinSDD, BuildPosFSDD, AppendID, and BuildPosFSDDdag.

Experiment 1: PosFSDD construction. First, Fig. 10, and Fig. 11 show the results. From Fig. 10, we see that PosFSDDs are almost $O(n \log n)$ size for n length text. The number of nodes are between $12n$ to $15n$. As is illustrated in Fig. 11, the proposed BuildPosFSDD runs in $O(n \log n)$.

Experiment 2: PosFSDDdag construction. Fig. 12 demonstrates that the PosFSDDdags are close to linear in the size of the input SDDs. The number of nodes are almost twice as that of the input SDD. As can be seen from Fig.13, BuildPosFSDDdag runs in almost $O(N \log N)$ time for N sized input SDDs, practically.

6 Conclusions

We proposed PosFSDD that is a complete inverted file for a text based on SDD. We also defined complete inverted files for directed acyclic graphs and implemented it as PosFSDDdag. They allow all queries to be solved in $O(|\Sigma||x|)$ time for n sized input. We gave algorithms that construct PosFSDD and PosFSDDdag. From the experimental results, their sizes are compact and our algorithms BuildPosFSDD and BuildPosFSDDdag run in almost $O(n \log n)$ time. The exact size bound of PosFSDD and the exact time complexity of our algorithms are not obvious. To propose more efficient algorithms is our future work. Position restricted search with PosFSDD is also a challenging problem.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman: *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
2. H. Aoki, S. Yamashita, and S. Minato: *An efficient algorithm for constructing a sequence binary decision diagram representing a set of reversed sequences*, in Proceedings of the 2011 IEEE International Conference on Granular Computing (GrC'2011), IEEE, 2011, pp. 54–59.
3. A. Blumer, J. Blumer, D. Haussler, R. M. McConnell, and A. Ehrenfeucht: *Complete inverted files for efficient text retrieval and analysis.* J. ACM, 34(3) 1987, pp. 578–595.
4. S. Denzumi, R. Yoshinaka, H. Arimura, and S. Minato: *Notes on sequence binary decision diagrams: Relationship to acyclic automata and complexities of binary set operations*, in Proceedings of the Prague Stringology Conference 2011 (PSC'11), J. Holub and J. Žďárek, eds., Czech Technical University in Prague, 2011, pp. 147–161.
5. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 3rd. ed., 2006.
6. D. E. Knuth: *The Art of Computer Programming, volume 4, fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams*, Addison-Wesley, 2009.

7. E. Loekito, J. Bailey, and J. Pei: *A binary decision diagram based approach for mining frequent subsequences*. Knowledge and Information Systems, 24(2) 2010, pp. 235–268.
8. S. Minato: *Zero-suppressed BDDs and their applications*. Software Tools for Technology Transfer, 3(2) 2001, pp. 156–170.
9. S. Minato: *SAPPORO BDD package*. Division of Computer Science, Hokkaido University, 2011, unreleased.
10. M. Mohri, P. Moreno, and E. Weinstein: *Factor automata of automata and applications*, in Proceedings of the 12th International Conference on Implementation and Application of Automata (CIAA'07), LNCS 4783, Springer, 2007, pp. 168–179.
11. D. Perrin: *Finite automata*, in Handbook of Theoretical Computer Science,, J. van Leuwen, ed., vol. B. Formal Models and Semantics, Elsevier, 1990, pp. 1–57.