Efficient Methods for Multigram Compound Discovery

Wu Horng Jyh, Paul
Mustard Technology Pte Ltd
Republic of Singapore
paulwu@mustardtech.com

Ng Hong I
School of Computing, National University of Singapore
Republic of Singapore
nghi@comp.nus.edu.sg

Gong Ruibin
School of Computer Engineering, Nanyang Technological University
Republic of Singapore
gongrb@pmail.ntu.edu.sg

Abstract

Multigram language model has become important in Speech Recognition, Natural Language Processing and Information Retrieval. An essential task in multigram language model is to establish a set of significant multigram compounds. In Yamamotto and Church (2001), an O(*N*log*N*) time complexity method based on Generalised Suffix Array (GSA) has been found, which computes the *tf* (term frequency) and *df* (document frequency) over O(*N*) classes of substrings. The *tf* and *df* form the essential statistics on which the metrics, such as MI (Mutual Information) and RIDF (Residual Inverse Document Frequency)[1], are based for multigram compound discovery. In this paper, it is shown that two related data structures to GSA, Generalised Suffix Tree (GST) and Generalised Directed Acyclic Word Graph (GDAWG) can afford even more efficient methods of multigram compound discovery than GSA. Namely, O(*N*) algorithms for computing *tf* and *df* have been found in GST and GDAWG. These data structures also exhibit a series of related, and desirable properties, including an O(*N*) time complexity algorithm to classify O(N^2) substrings into O(*N*) classes. An experiment based on 6 million bytes of text demonstrates that our theoretical analysis is consistent with the empirical results that can be observed.

1 Introduction

Multigram language model has become important in Speech Recognition (SR), Natural Language Processing (NLP) and Information Retrieval (IR) as demonstrated in Siu and Ostendorf (2000), Peng and Schuurmans (2002), and Chien (1999). It has also been used in evaluating NLP applications such as automatic Machine Translation and Text Summarization (Panineni, etc., 2002; Lin and Hovy, 2003).

For a corpus of length *N*, the computing cost of a naïve algorithm for the frequencies over all substrings is at least $O(N^2)$. In Yamamoto and Church (2001), an efficient method is given for computing the term frequency (*tf*) and document frequency (*df*), as well as the Mutual Information (MI) and Residual Inverse Document Frequency (RIDF), for all substrings based on Generalized Suffix Array (GSA). The method groups all $N(N+1)/2$ substrings into up to $2N-1$ equivalence classes, and in this way, the computation is reduced to a manageable computation over these classes, that is, $O(NlogN)$ time and $O(N)$ space.

It is natural to compare Generalised Suffix Tree (GST) and Generalised DAWG (GDAWG) with GSA since they all can be viewed as compact representations of suffix tries. Moreover, the construction complexities of GST and GDAWG are $O(N)$, while that of GSA is $O(NlogN)$. This raises the question:

[1] MI and RIDF by Yamamoto and Church (2001) are given below:

$$MI(xYz) \equiv \log_2 \frac{P(xYz)}{P(xY) \times P(z|Y)} = \log_2 \frac{\frac{tf(xYz)}{N}}{\frac{tf(xY)}{N} \times \frac{tf(Yz)}{tf(Y)}} = \log_2 \frac{tf(xYz) \times tf(Y)}{tf(xY) \times tf(Yz)} \qquad RIDF(x) \equiv -\log \frac{df}{D} + \log(1 - e^{-\frac{tf}{D}})$$

Where *x* and *z* are tokens, *Y* and *xYz* are ngrams (sequences of tokens).

Are GST and GDAWG the same or more efficient data structures than GSA for multigram compound Discovery?

In Crochemore and Rytter (1994), a set of properties has been identified such that a data structure *D* is said to be *good* if:

(Property A) *D* has linear size.
(Property B) *D* can be constructed in linear time.
(Property C) *D* allows computing *FACTORIN(x, text)* in $O(|x|)$ time.

Although the above properties are desired for multigram compound discovery, additional properties are required to provide a more precise assessment. Two important basic statistics: *tf* and *df* are important. The frequency of a *substring* in a collection of strings is called the *term frequency* (or *tf*), and that of a *substring* occurred *among different strings* in the collection is called the *document frequency* (or *df*).

In this paper, the following properties are identified, in addition to Properties A – C, to assess *D*: let *N* be the size of a set of strings *TEXT*:

(Property D) *D* allows *tf* (term frequency) and *df* (document frequency) to be computed in $O(N)$ time.
(Property E) *D* allows classifying $O(N^2)$ multigrams into $O(N)$ classes with the same tf in $O(N)$ time.
(Property F) *D* allows, Residual Inverse Document Frequency (RIDF) and Mutual Information (MI) to be computed in $O(N)$ time.

It is self-evident that Property D is a desirable property. Property E reduces the lower bound of Mutual Information computation from $O(N^2)$ to $O(N)$. Property F represents the ultimate potential for an efficient multigram term discovery algorithm. It is also noted that Properties D, E and F, represent an increasingly tighter criteria; that is, if the earlier, less stringent property is not satisfied, it is impossible for the latter, more stringent property to be satisfied.

This paper proposes two new multigram term discovery algorithms based on GST and GDAWG and proves that they fulfil Property A-E, while the GSA-based method does not satisfy any of the above desirable properties except Property A.

2 Multigram Compound Discovery Methods Based on Generalized Suffix Tree

The fact that Suffix Tree has linear size and can be constructed in linear time is well documented in the literature (Ukkonen, 1995). It is also known that *FACTORIN(x, text)* can be computed in $O(|x|)$ time. The algorithm is simply to traverse the Suffix Tree from the root by consuming the string *x* character by character. If the traversal can be completed for the entire string, then the answer to *FACTORIN(x, text)* is yes; otherwise, the answer is no. The time taken to decide *FACTORIN(x, text)* is thus, $O(|x|)$. A

A GST is an extension to Suffix Tree over the a set of strings, $text_i$, $i = 1, n$. For the convenience of the discussion, it is assumed that these $text_i$ are sorted in alphabetic order. In the following algorithm, we adopt the notion of (Ukonnen, 1995) and describe the algorithm to construct the Generalised Stuffix Tree (GST).

```
GST Construction ← procedure($text_1$, ..., $text_n$) {
    Construct the Generalised Suffix Tree of $text_1$, GST ($text_1$);
    For $i \leftarrow 2 \ldots n$ do
        insert ($text_i$, GST ($text_1$, ..., $text_{i-1}$)); }

insert ← function ($text_i$, GST ($text_1$, ..., $text_{i-1}$)) {
    given $text_i = t_{i1}\, t_{i2} \ldots$ #
    $(s_1, k_1) \leftarrow$ findPrefix ($text_i$, GST ($text_1$, ..., $text_{i-1}$));
    $s \leftarrow s_1$; $k \leftarrow k_1+1$; $i \leftarrow k_1$;
    While $t_i$ != # do
      $i \leftarrow i + 1$;
      $(s, k) \leftarrow$ update $(s, (k, i))$;
      $(s, k) \leftarrow$ canonize $((s, (k, i))$; }
```

Where **findPrefix** will traverse the longest possible prefix of ***text$_i$*** contained in GST(***text$_1$***, ..., ***text$_{i-1}$***) and return the canonical reference pair (*s_1*, *k_1*) for that prefix; procedures **update** and **canonize** are the same as those defined in (Ukonnen, 1995).

The Generalised Suffix Tree as constructed above retains all of the above properties, namely properties A, B, and C. This can be observed quite clearly by the fact that GST is but a union of all the automatas that individually satisfy properties A, B and C.

The GST for two alphabetically sorted strings (1) "cacacao" and (2) "cacao" is demonstrated in the following Figure 2.1.

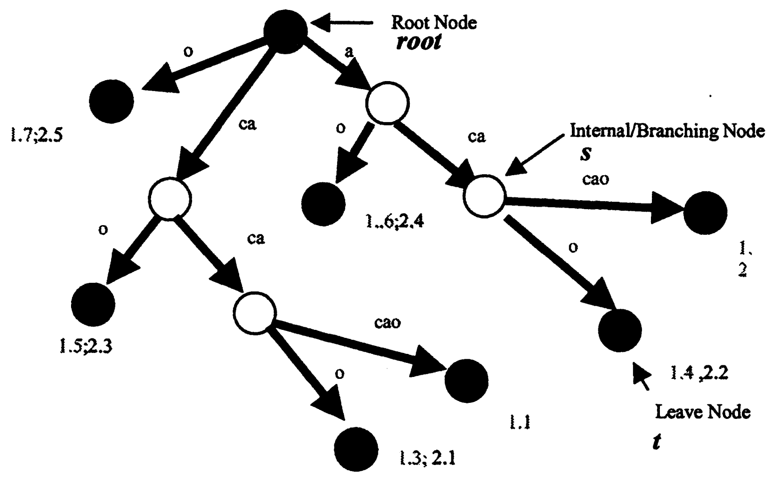


Figure 2.1: A GST for strings: (1) "cacacao" (2) "cacao," GST("cacacao", "cacao"). Each suffix is associated with a *occurrence pair x.y*. Fore example, the occurrence pair of "cacacao" is **1.1**.

Assuming background understanding of a tree data structure, a few relevant concepts of Generalized Suffix Tree (GST) are recalled in Figure 2.1: the "root node," denoted as ***root***, is colored in gray. The white nodes are the "internal," or "branching," nodes; ***s*** is one such node in Figure 2.1. The leaf nodes are demonstrated as the black nodes, of which ***t*** is an instance.

A GST tree can also be viewed as an automata where the nodes are the states and the "labelled" edges the acceptable input strings. In the following, when the properties are discussed in the , the duality is assumed between a node/state, ***n***, of a tree and a string/prefix ***l*** that satisfy ***n*** = (***root***, ***l***). For example, instead of saying ***t*** is reachable from ***s***, one may say "acao" is reachable from ***s*** since ***t*** = (***root***, "acao").

As demonstrated in Figure 2.1, each suffix ***suffix$_i$*** is associated with an *occurrence pair x.y*, where ***x***, called the ***x*** dimension of the occurrence pair, which is the alphabetic order of the string ***text$_i$*** of which ***suffix*** is a suffix; and ***y***, called the ***y*** dimension of the occurrence pair, is the starting position of ***suffix$_i$*** in ***text$_i$***. In Figure 2.1, the suffix "acao" that terminates at a leaf node ***t*** has two occurrence pairs **1.4** and **2.2**, which specify that "acao" starts at the 4th and 2nd positions of the 1st and 2nd strings of the GST("cacacao, "cacao"), respectively.

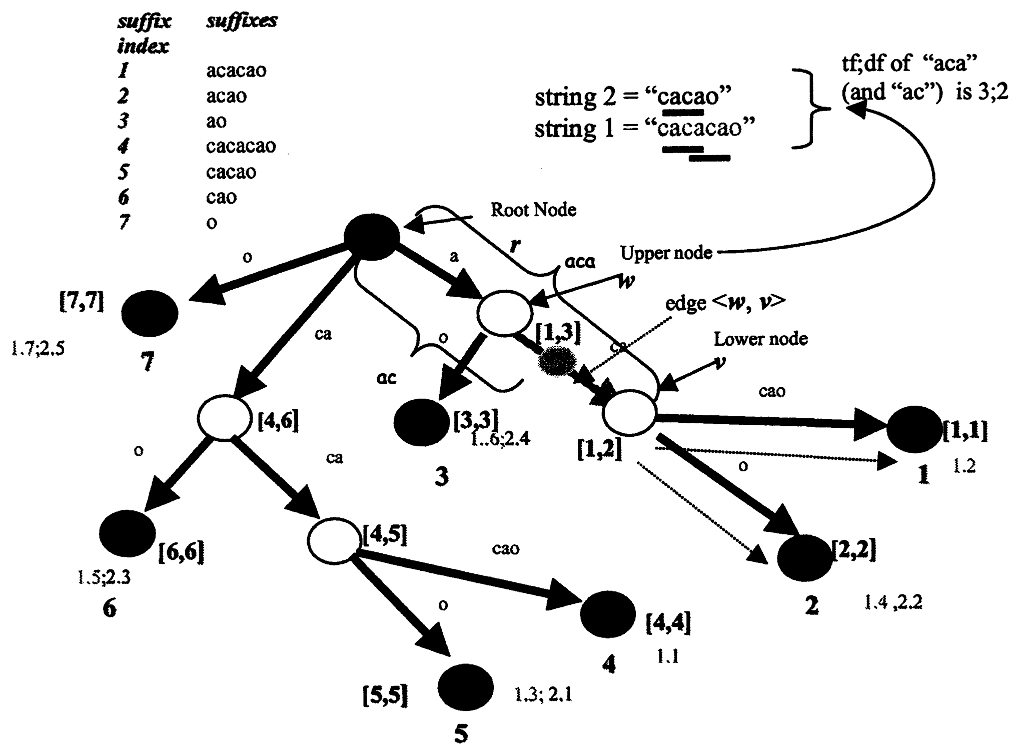
In some cases, a branching node can also be a leaf node. For example, in a GST that contains one string "caca", the node ***v*** = (***root***, "ca") is both an internal node as well as a leaf node, indexed at **1.3**.

2.1 Term frequency (tf) and document frequency (df) of a domination range class

We further define *suffix index* of a suffix as its order in a alphabetically sorted sequence of the suffixes of all the strings in the GST. As demonstrated in Figure 2.2, the suffix "acacao" has the order index of 1 as it is the 1st suffix among all suffixes in the GST("cacacao", "cacao").

Lemma 1a: The *tf* and *df* of a suffix, which terminates at a leaf node, of a GST are equal to the numbers of the suffix's occurrence pairs and the distinct x-dimension integers of its occurrence pairs, respectively.

The proof of lemma is self-evident. For example, the suffix "acao" has a *tf* of 2 and *df* of 2, since the suffix has 2 occurrence pairs – 1.4 and 2.2, and 2 distinct x-dimension integers of the set of occurrence pairs, namely 1 and 2, respectively.



suffix index	*suffixes*
1	acacao
2	acao
3	ao
4	cacacao
5	cacao
6	cao
7	o

Figure 2.2: The seven distinct suffixes of GST("cacacao", "cacao") form a *suffix index*. A left-open edge <*w*, *v*] of the GST has a upper node *w* and lower node *v*. The substring "aca" that terminates in <*w*, *v*] has a *tf;df* counts of 3;2, whose occurrences are underlined in each of the strings in the GST. The domination range of each of the node is demonstrated in the pair [x,y]. Particularly, *v* has a domination range of [1,2] and *w*, [1,3]

Lemma 1b: Given the set of occurrence pairs associated with the suffixes that are reachable from an (internal) node *v*, the *tf* and *df* of the substring that terminates at *v* are equal to the rank of the set of occurrence pairs and the distinct x-dimension integers of the set.

Proof: The substring ***l*** that terminates at an internal node is a *prefix* of all the suffixes m_i that are reachable from the node; that is, each $m_i = l.n_i$ Recall that *tf* of m_i is equal to the frequency of m_i in the GST. Since each occurrence of m_i will imply an occurrence of ***l***, *tf*(***l***), is equal to summation of *tf*(m_i) for all ***i***, that is equal to the rank of the set of occurrence pairs of all m_i. Similarly, one can arrive that for *df*(***l***).

We define an *left-open* edge, <*w*,*v*] of a GST as the edge that contains nodes between the upper node (*w*) and lower node (*v*) of an edge, it is left-open because it does not include the upper node *w*, while it does contain the lower node *v*.

Lemma 1c: Each substring that terminates in a node between a left-open edge <*w*, *v*] has the same *tf* and *df*.

Proof: Recall the property of a suffix tree: all substrings that terminate in an left-open edge of a GST, except the lower node, terminate at an *implicit* node, which does not branch. Thus, the suffixes reachable from these nodes are the same as the lower node. By Lemma 1b, it can be concluded that the *tf* and *df* of the substrings, which terminate in the implicit nodes, are the same as those of the substring that terminates at the lower node.

As demonstrated in Figure 2.2, the substring "ac" terminates at an internal node (coloured grey). Given that the *tf;df* of "aca," the lower node of the edge <*w*, *v*>, is 3;2, the *tf;df* of "ac," is 3;2 as well. In fact, since "ac" is a prefix of "aca," it can be shown in each of the underlined occurrences of "aca," there is an occurrence of "ac," which in consistent with Lemma 1c.

domination range	***longest substring***	***frequency count (tf;df)***	***other prefixes in class***
<1,1>	acacao	1;1	acaca,acac
<2,2>	acao	2;2	-
<1,2>	aca	3;2	ac
<3.3>	ao	2;2	-
<1,3>	a	5;2	-
<4.4>	cacaocao	1;1	cacaca,cacac
<5.5>	cacao	2;2	-
<4,5>	caca	3;2	cac
<6,6>	cao	2;2	-
<4,6>	ca	5;2	c
<7,7>	o	2;2	-

Figure 2.3. The 11 domination ranges, of left-open edges of GST("cacacao", "cacao"). The highlighted <1,2> domination range has the *tf;df* counts of 3;2.

The *domination range* of a left-open edge in a GST is defined as a pair of suffix indices, [*x*, *y*], where *x* is the *minimum* suffix index of those suffixes that the lower node of the left-open edge dominates, while *y* is the *maximum*. For example, in Figure 2.3, it is demonstrated the domination range of the node *v* is **[1,2]**, this is because the subtree dominated by *v* has two leaf nodes whose suffix indices are **1** and **2**, respectively. It is noted that left-open edges associated with all leaf nodes has a *trivial domination range* where the two suffix indices in the domination range are the same, such as **[1,1]** , **[2,2]**, ..., and **[7,7]**. Each domination range also has a representative, which is the longest substring that terminates at the lower node of the edge. These are demonstrated in Figure 2.3.

Theorem 1: The classes of distinct domination ranges of a GST form a partition of all substrings of the GST, where each substring in a domination range class has the same *tf* and *df*.

Proof: Proof of the latter part of **Theorem 1** follows from **Lemma 1c**; the former part follows from the fact that all substrings terminate in one and only one left-open edge that defines one domination range.

Corollary 1: There are *O(N)* of distinct domination ranges of a GST and it takes *O(N)* time to classify all of the substrings according to its domination range.

Proof: The fact that there are *O(N)* number of left-open edges in a GST proves the first part of **Corollary 1**. The second part follows by the fact that there exists *O(N)* algorithms to construct the GST and once the construction of a GST is finished the edge and the partition based on domination range is completed at the same with the edges constructed.

The discussion in section 2.1 explain Property E can be achieved by defining domination range which classify the substrings in a GST into $O(N)$ classes in $O(N)$ time.

2.2 Algorithm for counting *tf* and *df*

The counting of *tf* and *df* is performed at the end of each insertion step of a string ***text$_k$*** in constructing GST(***text$_1$***, ***text$_2$*** ..., ***text$_m$***). It performs a *bottom-up* traversal of the boundary path, the path followed by the suffix links starting at the longest suffix of ***text$_k$***. It also keeps a stack, storing the parents of leaf nodes in the boundary path and for checking which category the nodes of concern belong to: among pure leaf nodes, pure internal nodes or leaf-cum-internal nodes.

```
update_tf_df ← procedure (GST($text_k$)) {
For $node_i \leftarrow node_1, node_2, \ldots, node_{n-1}$ along the $suffix_j$ of $text_k$, do {
    while $node_j \leftarrow$ pop_stack ($j$), do {
        if $node_j$ is a leaf_cum_internal node, do
            $tf(node_j) \leftarrow tf(node_j) + 1 + delta\_tf(node_j)$; $df(node_j)$ ++;
            $delta\_tf(\text{parent}(node_j)) \leftarrow delta\_tf(\text{parent}(node_j)) + 1$;
            $df(\text{parent}(node_j))$ ++;
            push_stack(parent($node_j$), depth(parent($node_j$)));
        if $node_j$ is an pure_internal, do
            $tf(node_j) \leftarrow tf(node_j) + delta\_tf(node_j)$; $df(node_j)$ ++;
            $delta\_tf(\text{parent}(node_j)) \leftarrow delta\_tf(\text{parent}(node_j))$;
            $df(\text{parent}(node_j))$ ++;
            push_stack(parent($node_j$), $j$);
    $node_i$ is a pure_leaf node, do
        $tf(node_i) \leftarrow tf(node_i) + 1$; $df(node_i) \leftarrow df(node_i) + 1$;
        $delta\_tf(\text{parent}(node_i))$ ++; $df(\text{parent}(node_i))$ ++;
        push_stack(parent($node_i$), depth(parent($node_i$))); }

internal_or_leaf ← function($node_i$, $node_j$) {
    if $node_i == node_j$, do
        return pure_internal;
    else
        return leaf_cum_internal; }
```

The above algorithm can be completed in $O(N)$ + O(sizeof(***stack***)) time. Since the sizeof(***stack***) is proportional to the number of internal nodes of a Suffix Tree, it is known to be $O(N)$. Thus the above algorithm to update *tf* and *df* will take $O(N)$ time altogether.

2.3 Computation of MI and RIDF for multigram compound discovery

The proof of Property F for GST is achieved by considering the following formulae: given a substring $w = xyz$, where x, xy, xyz are the longest substrings in their respective classes:

$$MI(w) = \log_2 \frac{tf(w) \times tf(y)}{tf(xy) \times tf(yz)} \qquad RIDF(w) \equiv -\log \frac{df(w)}{D} + \log(1 - e^{-\frac{tf(w)}{D}})$$

The above formulae can be computed in constant time, since each of the *tf*'s involved in the formula can be accessed from *root* by traversing one of the substrings: w, y, xy and yz. Since there are $O(N)$ classes of a substring like w, the computation of MI and RIDF is achieved in $O(N)$ time. This concludes the description of a proof to Property E.

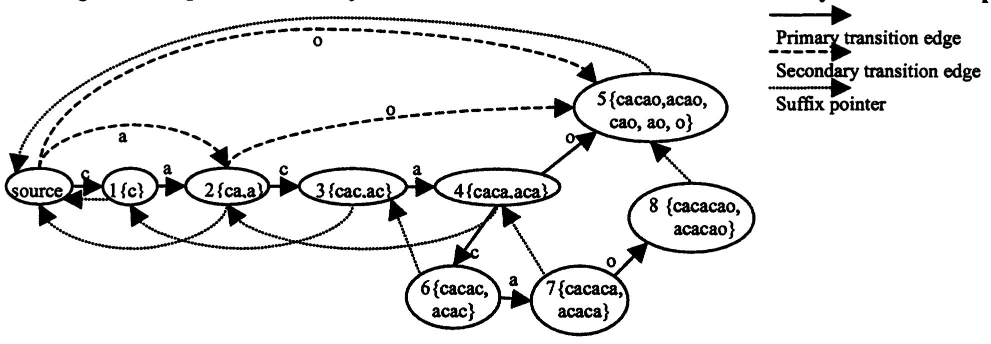


Figure 3.1: GDAWG for strings "*cacao*" and "*cacacao*".

Figure 3.1 shows the GDAWG constructed using the strings "*cacao*" and "*cacacao*". In this Section 3.1, we provide a detailed analysis of the algorithm that constructs a GDAWG using a set of input strings *S*. In Section 3.2, we describe how we calculate term (*tf*) and document frequencies (*df*) in linear time. Term frequency is the number of times where a substring occurs in a corpus. Document frequency is the number of unique strings where a substring occurs. They are required in multigram compound discovery. However, in order to obtain a frequency counting algorithm that runs in linear time, we update the *tf* and *df* in each state as the GDAWG is being constructed. This update is based on recurring prefixes of substrings in *S*. In addition, we store the last string identity (SID) in each state to denote the last string with which the state's *df* is updated. This is to aid the computation of *df*'s. We show the steps for these updating in Section 3.1, together with the algorithm for constructing GDAWG.

3.1 Algorithm for Constructing GDAWG

We describe the differences between our algorithm and Algorithm A of Blumer et al. (1985) in the following sub-sections.

3.1.1 Resetting Current Sink to Source

The *current sink* is reset to be the *source* of the GDAWG when a new string in *S* is about to be processed. The new `builddawg` algorithm that takes $S = \{s_0, s_1, s_2,..., s_{n-1}\}$ as input is presented below.

```
builddawg ← functions(S){
    Create a state named source and let currentsink be source.
    for $s_j \leftarrow s_0, s_1, s_2,..., s_{n-1}$ do
        Let currentsink be source;
        For each symbol a of $s_j$ do currentsink ← update(currentsink, a);
    Return source.}
```

Figure 3.2 (a) to (b) gives a snapshot of resetting the *currentsink* to *source*.

3.1.2 Check for Existing Outgoing Edge and Update Frequencies

Assume the symbol currently being scanned is *a*. We need to check the *current sink* whether there is already an outgoing edge labelled *a* before we create a state named *new-sink*. If an outgoing edge labelled *a* has been created previously in the *current sink*, further processing would depend on whether the edge is a secondary edge. In this case, the next state where this edge leads to must be split using the same splitting function presented in Algorithm A of Blumer et al. (1985).

If an outgoing edge labelled *a* has been created previously in the current sink and it is a primary edge, assume *s'* is the next state of the primary edge, we increase *tf* of *s'* by 1. This implies that the current symbol has contributed one more *tf* count to the strings represented by *s'*. For the *df* count in *s'*, we increase it by 1 and update the SID of *s'* to *j*. However, we do this only if SID of *s'* is less than *j*. This is because that the current string contributes to one more *df* count to those strings represented by *s'*.

Recall that in the `builddawg` function, j is the index of the current string being processed. After this, assume w is the prefix of s_j processed thus far, we loop through the states containing the successively shorter suffixes of w to increase the *df*'s by 1 and update the SID's to j. This loop terminated when we find a state with SID equals to j. The above processes can be seen in Figure 3.2 (b) to (c). Figure 3.2 (b) shows the GDAWG after "*cacao*" has been scanned. Figure 3.2 (c) shows the GDAWG after "*caca*" has been added. Notice that *tf*'s of states 1 to 4 has been increased by 1 in Figure 3.2 (c). Their *df*'s are also increased by 1 since their SID's are all less than j = 1, which represents the second string. After that, their SID's are updated to be the current value j.

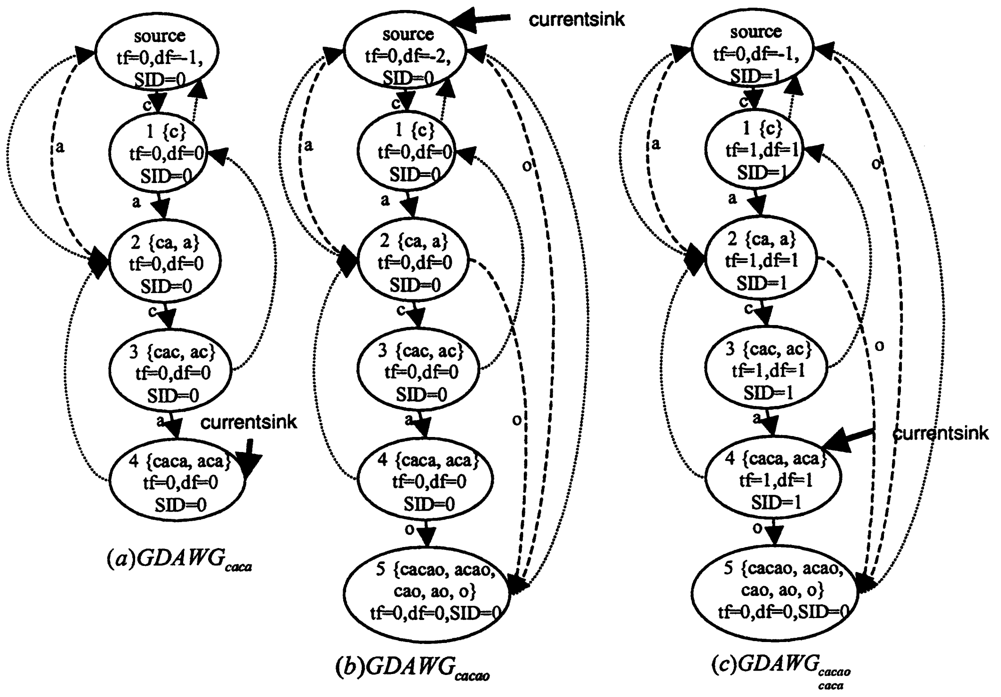
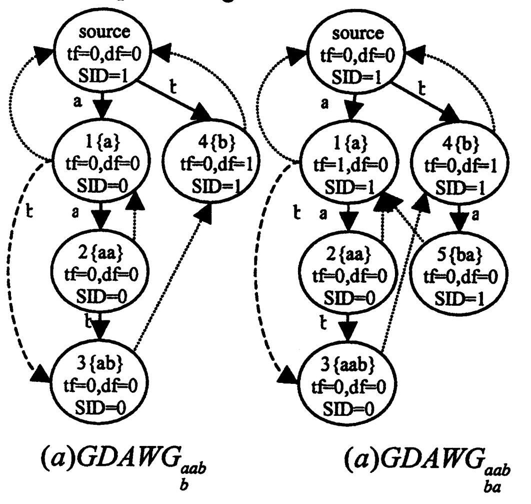

source tf=0,df=-1, SID=0
1 {c} tf=0,df=0 SID=0
2 {ca, a} tf=0,df=0 SID=0
3 {cac, ac} tf=0,df=0 SID=0
4 {caca, aca} tf=0,df=0 SID=0
currentsink
$(a)GDAWG_{caca}$
source tf=0,df=-2, SID=0
1 {c} tf=0,df=0 SID=0
2 {ca, a} tf=0,df=0 SID=0
3 {cac, ac} tf=0,df=0 SID=0
4 {caca, aca} tf=0,df=0 SID=0
5 {cacao, acao, cao, ao, o} tf=0,df=0,SID=0
currentsink
$(b)GDAWG_{cacao}$
source tf=0,df=-1, SID=1
1 {c} tf=1,df=1 SID=1
2 {ca, a} tf=1,df=1 SID=1
3 {cac, ac} tf=1,df=1 SID=1
4 {caca, aca} tf=1,df=1 SID=1
5 {cacao, acao, cao, ao, o} tf=0,df=0,SID=0
currentsink
$(c)GDAWG_{cacao \atop caca}$
c
a
o

Figure 3.2: Process of constructing the GDAWG that represent all substrings in S = {*cacao*, *caca*}.

If an outgoing edge labelled a has not been created previously, we follow the steps in the `update` function of Algorithm A (Blumer et al., 1985). After that, we initialize the *tf* and SID of the newly created *newsink* to 0 and j respectively. In addition, we increase the *tf* of the suffix state by 1 if the suffix state is not the source, and the edge that is followed to reach the suffix state is primary, and there are currently two suffix pointers pointing to it, one of which is added recently. As mentioned in the beginning of this paragraph, the initial *tf* count is implicitly represented in each state during the state creation. This implicit count can be stored in the *tf* of the state either now or during the final update of *tf*'s. We choose to make it explicit now so that for states with more than one child states, we simply sum up the *tf*'s of the children plus any additional *tf* counts contributed by recurring strings during the final update of *tf*'s. This process is shown in Figure 3.3. Note that *tf* of state 1 has been increased by one. Following that, we set the SID of the suffix state to j if the SID of the suffix state is less than j. This implies that the *df* count represented in *newsink* will contribute to one *df* count to the suffix state through the suffix pointer (Figure 3.3 – state 5 will contribute one *df* count to state 1 during our algorithm presented in Section 3.2). If the SID of the suffix state is j, and there are currently more than one suffix pointers pointing to it, we decrease *df* of the suffix state by one. This implies that the *df* contributed by s_j has already been taken cared of by the other suffix pointer. This is shown in **Error! Reference source not found.** (a) to (b). Note that SID of *source* in (a) is j = 0. In (b), *df* of *source* is decreased by one because there are two suffix pointers that contribute to the *df* count of s_j. One from state 1 and the other one from state 2.

The `update` algorithm that takes *currentsink* and *a* as inputs is presented below.



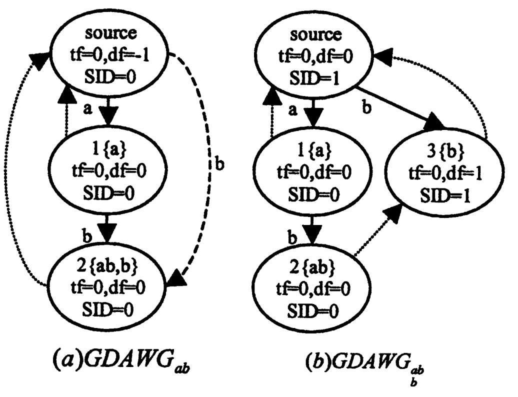


Figure 3.3: Process of scanning '*a*' after {*aab*, *b*}. In state 1, *tf* is increased by 1 and SID is set to 1.

Figure 3.4: Process of adding s_j="*b*" to $GDAWG_{ab}$

```
update ← function(currentsink, a){
  Let newsink be the state pointed to by an existing a-labelled outgoing edge of currentsink.
  if newsink is defined, do
    if the existing a-labelled outgoing edge of currentsink is a secondary edge, do
      newsink ← split(currentsink, newsink);
    else (the existing a-labelled outgoing edge of currentsink is a primary edge)
      $tf_{newsink} \leftarrow tf_{newsink} + 1$;
      if $SID_{newsink} < j$, do $df_{newsink} = df_{newsink} + 1$; $SID_{newsink} \leftarrow j$;
      Let currentstate be the state pointed to by the suffix pointer of newsink.
      while $SID_{currentstate} < j$, do
        $df_{currentstate} = df_{currentstate} + 1$; $SID_{currentstate} = j$;
        Let currentstate be the state pointed to by the suffix pointer of currentstate.
  else
    (As what is done in Algorithm A of Blumer et al. (1985) except the following.)
    $tf_{newsink} \leftarrow 0$; $SID_{newsink} \leftarrow j$ (from $s_j$);
    When the currentstate has a primary outgoing edge labelled a while traversing the
    successively shorter suffixes, set edgetype to true;
    if suffixstate is not source and edgetype is true, and there are currently 2 suffix pointers
    pointing to it, do $tf_{suffixstate} \leftarrow tf_{suffixstate} + 1$;
    if $SID_{suffixstate} < j$, do $SID_{suffixstate} \leftarrow j$;
    else if there are more than 1 suffix pointers pointing at the suffixstate, do
      Reduce $df_{suffixstate}$ by 1;
  Return newsink;}
```

3.1.3 Update Document Frequencies during a Split

As in `update` function, we need to update the frequencies during a split operation. After the split operation as presented in Algorithm A of Blumer et al. (1985) is performed, we increase *df* of suffix state of new child state by one and set its SID to *j* if the SID is less than *j*, i.e.,

```
Let suffixstate be the suffix state of the newchildstate.
if $SID_{suffixstate} < j$, do
  $df_{suffixstate} = df_{suffixstate} + 1$;
```

$SID_{suffixstate} \leftarrow j$;

During a split operation, the suffix pointer pointing from child state to the suffix state is changed so that it points from new child state to the same suffix state. Thus, the above update signifies that the strings represented by the new child state contribute to one more *df* count to the suffix state. This is shown in Figure 3.4. Note that SID_{source} is 0 in (a). So, df_{source} is increased by 1 and SID_{source} is set to $j = 1$.

In addition, the new child state will be the suffix state of the child state. Thus, we increase the *df* count at the new child state by one if the SID of the child state is less than j. This implies that, in addition to the *df* count contributed by the child state, the current string s_j contributes to one *df* count at the new child state too, i.e.,

if $SID_{childstate} < j$, do $df_{newchildstate} = df_{newchildstate} + 1$;

$SID_{newchildstate} \leftarrow j$;

This is shown in Figure 3.4 (b). Note that state 4 is the new suffix state of state 2 and df_{state4} is increased by one from the initial value of zero.

3.1.4 Combinatorial Analysis of the Algorithm for Constructing GDAWG

The extra `for` loop in our `builddawg` function is simply used to loop through all the strings in *S*. Thus, it does not create more complexity to the original DAWG construction algorithm. The only extra loop is in our `update` function. It is used to update the SID's of the successive suffix states of *newsink* and increase their *df*'s by following the suffix link that begins from the *newsink*.

Our corpus for multigram compound discovery contains 146,844 strings and 5,863,591 symbols. The minimum, average and maximum string lengths are 3, 39.93 and 138 symbols respectively. Due to this, we think the extra loop will not increase the complexity of the algorithm. This is supported by our experiment results where the shortest, average and longest suffix link following during the GDAWG construction are 0, 5.39 and 24 respectively. In addition, the time grows linearly with our corpus size.

Thus, our algorithm to construct a GDAWG based on *S* is online in linear space and time, and the resulting GDAWG allows the computation of *FACTORIN*(*x*, *TEXT*) in $O(|x|)$ time.

3.2 Final Update of Term and Document Frequencies

After the processing described in Section 3.1, *tf*'s in the GDAWG represent the counts contributed by the recurring prefixes in the corpus and non-unique first symbols; and *df*'s represent the offsets that should be added to the number suffix pointers pointing to it in order to compute the correct final *df*'s. To compute the final *tf* and *df*'s, we do a depth-first traversal on the GDAWG. During the traversal, *tf* and *df* at the leave nodes are increased by one in order to count the initial occurrence of the strings implicitly represented by these leave nodes. For non-leave nodes, the final *tf* and *df* are simply the addition of the original counts and the *tf* and *df* contributed by the child states. In addition, for states with only one child, we need to increase its *tf* count by one in order to take care of the initial string occurrence that causes the creation of the state.

The `updateFreq` function that takes in the source of the GDAWG is presented below.

```
updateFreq ← function(source of GDAWG){
    if the state is a leave node, do
        Increase tf of the state by 1;
        Increase df of the state by 1;
        Return tf;
    else (the state is not a leave node)
        for each child of state, do
            tf ← tf + updateFreq(child);
            df ← df + 1;
        if the state has only one child state, do tf ← tf + 1;
        Return tf;}
```

As shown above, the algorithm to perform the final update of *tf* and *df*'s is linear.

3.3 Multigram Compound Discovery Based on GDAWG

As in DAWG, each state in the resulted GDAWG represents a class of substrings that are *end-equivalent.* Since the number of resulted states is linear, i.e., $N = |TEXT|$ (Blumer et al., 1985), there are at most $2N$-1 classes of substrings in GDAWG. Since the GDAWG construction algorithm is linear, the number of classes represented by GDAWG and the time to find the classes is linear. (In the following, we use class and state interchangeably.) Thus, the two parameters used in our multigram compound discovery, i.e., MI and RIDF (as shown in Section 2.3) for the longest substring in each class, can be computed in linear time.

Here, *yz* represents the longest suffix of *w*. It's either in the same class as *w* or in the class pointed to by the suffix pointer of the class containing *w*. "*xy*" can be accessed by keeping a parent pointer during the traversal of GDAWG. Thus, all required parameters for the above formula can be accessed in constant time. We just need to traverse the entire GDAWG to compute the MI and RIDF of the longest substring in each class.

4 Multigram Compound Discovery Methods based on Generalized Suffix Array

Suffix Array (SA) is an array of all N suffixes of a given corpus, sorted alphabetically. It was introduced as a new and conceptually simple data structure for online string searching by Manber and Myers (1990). SA allows computing the membership function, *FACTORIN(x, text)* in $O(|x|+log|text|)$ time. SA can be constructed in $O(NlogN)$ time[2]. These results hold for GSA. The major advantage of GSA over GST is space. The space requirements for GST grow with the alphabet size $|\Sigma|$: $O(N|\Sigma|)$ space. The dependency on alphabet size could be a serious issue for many cases, e.g., some Asia languages, such as Chinese, have a relatively large alphabet of more than 6,000 characters. So the advantages of Suffix Arrays over Suffix Trees becomes much significant for larger alphabets.

Detailed techniques of using GSA to compute *tf* and *df* for all substrings in a corpus were given in Yamamoto and Church (2001). The main idea is to group all $N(N+1)/2$ substrings into a manageable number, i.e. up to $2N-1$, of equivalence classes, and the substrings in a class all share the same *tf* and *df*. In this way, the computation over substrings is reduced to a manageable computation over classes, that is, $O(NlogN)$ time and $O(N)$ space. This implies Property D and E do not hold for GSA.

In Yamamoto and Church (2001), MI and RIDF were computed for the longest substring in each non-trivial class (up to N-1). The time required is $O(NlogN)$ as each of the terms in the formula will require $O(logN)$ time to access. This means GSA does not have Property F either.

5 Experiment Result

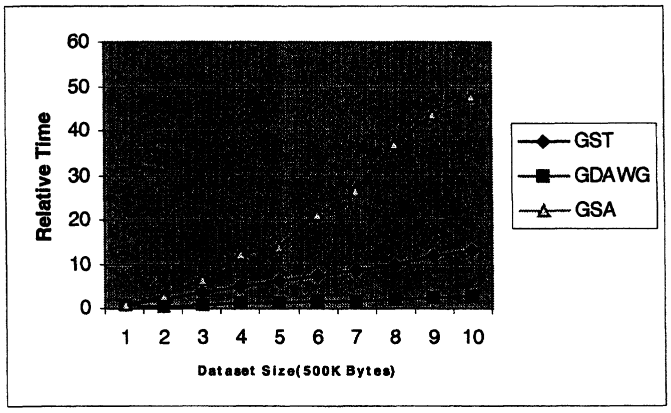


Figure 5.1: GST/GDAWG/GST

[2] Manber and Myers (1990) also gave a augmented algorithm that, regardless of the alphabet size, constructs Suffix Array in *O(N) expected* time, albeit with lesser space efficiency. It also reported that Suffix Arrays use three to five times less space than Suffix Trees even in the case of relatively small alphabet size ($|\Sigma|$=96) in practice.

To evaluate the performance in a real world application, we run our algorithms together with the one based on GSA over our Philippine address dataset, which consists of 6 million bytes data and 146,844 address records and has a small alphabet set (< 128) and record size (< 1K bytes). Figure 5.1 shows the experiment result measured by *Relative Time*, which takes the processing time over unit test data (500K Bytes) as the time unit. Obviously, the time cost of our algorithms grow in linear with the data size, that is, in *O(N)*, which coincides with the theoretical analysis in Section 2 and 3.

6 Conclusion

This paper discusses the multigram compound discovery methods based on GST, GDAWG and GSA. A set of properties (A to F) is defined to access the efficiency of the algorithms.
This paper proposes two new algorithms based on GST and GDAWG, and proves that they are able to fulfil Property A to F (detailed comparisons with GSA are shown in Table 6.1). Thus, they are efficient algorithms for multigram compound discovery.

An experiment based 6 million bytes of text demonstrate that our theoretical analysis is consistent with the empirical results.

	Property A	Property B	Property C	Property D	Property E	Property E
	Size of *D*	Time required to construct *D*	*FACTORIN (x, |TEXT|)*	*tf* and *df*	Classification into *N* classes	*MI & RIDF*
GST	*O(N)*	*O(N)*	*O(|x|)*	*O(N)*	*O(N)*	*O(N)*
GDAWG	*O(N)*	*O(N)*	*O(|x|)*	*O(N)*	*O(N)*	*O(N)*
GSA	*O(N)*	*O(NlogN)*	*O(|x|+logN)*	*O(NlogN)*	*O(NlogN)*	*O(NlogN)*

Table 6.1: GST/GDAWG/GSA(Given the size of the set of strings |*TEXT*| is *N*)

References

Blumer, A,J. Blumer, D. Haussler, A. Ehrenfeucht, M.-T. Chen and J. Seiferas. 1985. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31-55.

Chien. L. F. 1999. PAT-Tree-Based Adaptive Keyphrase Extraction for Intelligent Chinese Information Retrieval, *Information Processing and Management*, 35(4):501-521.

Crochemore, M and Rytter, W. 1994. Text Algorithm. Oxford University Press, New York & Oxford.

Lin, C. Y., Hovy, E. 2003. Automatic Evaluation of Summaries Using N-gram Co-Occurrence Statistics, *Proceedings of the Human Technology Conference 2003 (HLT-NAACL-2003),* Edmonton,

Manber, U. and Myers, G. 1990. Suffix arrays: A new method for on-line string searches. In *the first Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 319-327.

Panineni, K. Roukos, S. Ward, T., and Zhu W.J. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation, *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL),* Philadelphia, pp. 311-318.

Peng. F and Schuurmans, D. 2001. A Hierarchical EM Approach to Word Segmentation, *Proceedings of the Sixth Natural Language Processing Pacific Rim Symposium (NLPRS 2001).* Pp. 475-480

Siu, M. and Ostendorf, M. 2000. Variable N-grams and Extension for Conversational Speech Language Modeling. *IEEE Transactions on Speech and Audio Processing*, 8(1):63-75.

Ukkonen, E. 1995. On-line Construction of Suffix Trees. *Algoritmica* 14(3):249-260

Yamamoto, M. and Church, K. 2001. Using Suffix Arrays to compute Term Frequency and Document Frequency for All Substrings in a Corpus, *Computational Linguistics, vol 27:1*, pp. 1-30, MIT Press.