Discovering Patterns from Large and Dynamic Sequential Data

Ke Wang
Department of Information Systems and Computer Science
National University of Singapore
wangk@iscs.nus.sg

Abstract

Most daily and scientific data are sequential in nature. Discovering important patterns from such data can benefit the user and scientist by predicting coming activities, interpreting recurring phenomena, extracting outstanding similarities and differences for close attention, compressing data, and detecting intrusion. We consider the following incremental discovery problem for large and dynamic sequential data. Suppose that patterns were previously discovered and materialized. An update is made to the sequential database. An incremental discovery will take advantage of discovered patterns and compute only the change by accessing the affected part of the database and data structures. In addition to patterns, the statistics and position information of patterns need to be updated to allow further analysis and processing on patterns. We present an efficient algorithm for the incremental discovery problem. The algorithm is applied to sequential data that honors several sequential patterns modeling weather changes in Singapore. The algorithm finds what it is supposed to find. Experiments show that for small updates and large databases, the incremental discovery algorithm runs in time independent of the data size.

Key words. combinatorial pattern matching, data mining, sequential pattern, suffix tree, update

1 Introduction

In the daily and scientific life, sequential data, called strings below, are available and used everywhere. Examples are text, music notes, weather data, satellite data streams, stock prices, experiment runs, DNA sequences, histories of medical records, log files, etc. Given a (potentially large) string S, we are interested in sequential patterns of the form $\alpha \rightarrow \beta$, where $\alpha, \beta, \alpha\beta$ are substrings inside S, such that the frequency of $\alpha\beta$ is not less than some minimum support and the probability that α is immediately followed by β is not less than some minimum confidence. Discovering sequential patterns can benefit the user or scientist by predicting coming activities, interpreting recurring phenomena, extracting outstanding similarities and differences for close attention, compressing data, detecting intrusion. For example, by discovering the common login patterns of the authorized user, a security system may be able to detect an foreign intrusion when login activities in a session are drastically different from what is predicted by the patterns. Since the underlying database is usually large, dealing with changing data and patterns is a challenge for research and application in knowledge discovery and data mining, and incremental methods for updating the patterns are possible solutions [FSS96].

The problem. We consider the following incremental discovery problem. Suppose that sequential patterns were previously discovered and materialized. An update is made to the underlying database. For a simple update, it is expected that most patterns and data structures remain unchanged and recomputing all patterns is unnecessarily expensive, especially for large and dynamic databases. The *incremental discovery* will take advantage of discovered patterns and compute only the change by accessing the affected part of the database and data structures. In addition to patterns, the incremental discovery also maintains the statistics and position information of patterns to allow further analysis and processing on patterns. Maintaining the position information is also necessary to locate occurrences of a substring for performing updates on them. In the discovery process, the user may dynamically refine his/her interestingness level by trying several minimum support and confidence. A viable incremental solution should allow tuning of patterns according to different minimum support and confidence, without recomputing patterns.

The contribution. This paper presents an efficient algorithm for the incremental discovery problem. We allow general updates at any specified position of the string database, not necessarily constrained to the two ends of the string. The suffix tree indexing widely used in the area of combinatorial pattern matching [B92, GY91, S94] does not work for such dynamic strings. The difficulty lies in maintaining the position information that are sensitive to general updates. We propose a new representation of the suffix tree, called the *dynamic suffix tree*, to index dynamic strings. The string database is stored on disks using a structure that supports efficient updates and mappings between positions and disk addresses. The dynamic suffix tree maintains disk addresses rather than positions of substrings, thus eliminating the sensitivity to updates. We present an algorithm for the incremental discovery problem using the dynamic suffix tree. The discovery framework and incremental discovery algorithm are tested on the sequential data that honors several sequential patterns modeling weather changes in Singapore. The algorithm has found what it is supposed to find. Experiments show that for small updates and large databases, the incremental discovery algorithm runs in time independent of the data size.

Discovering sequential patterns is related to finding all occurrences of a substring in a string database, called the *string searching problem* in the area of combinatorial pattern matching (see [B92, GY91, S94], e.g., for a survey). The main difference is that the string searching problem finds occurrences of a particular substring, whereas the discovery problem finds all substrings of some significant statistics. In similarity, both problems require some form of indexing on substrings. Two famous algorithms for the string searching problem are the Boyer-Moore algorithm [BM77] and the Knuth Morris Pratt algorithm [KMP77]. Extensions for this problem have dealt with approximate matchings, a pattern with “wild cards”or a regular expression, multiple patterns searching, and multi-dimensional searching. See [B92, BG92, WM92] for a partial list. For most work in this area, indexing or preprocessing was primarily used to speed up subsequent searches. The suffix tree indexing was used in [M76] to speed up updates for dynamic strings. Since [M76] adopted a position numbering scheme in which a position number never changes once assigned, like the Dewey-Decimal library access code, generated position numbers do not correspond to the logical ordering of characters and are practically useless. That position numbering scheme also suffered from running out of position numbers after some number of updates. More seriously, [M76] did not address the problem of dangling references to deleted characters in the suffix tree. See Section 3 for elaboration. The problem of discovering sequential patterns was an active research area in AI (see, for example, [MD85]) and was recently studied in the database area (see, for example, [ALSS95, AS95, W*94]). However, the discovery algorithms in these works have to be rerun if the data is updated.

Section 2 defines the discovery problem of sequential patterns and the incremental version of the problem. The suffix tree is introduced as a representation of sequential patterns. Section 3 briefly describes the construction and update algorithms of the suffix tree in the literature, highlighting the inability of the suffix tree for handling dynamic strings. Section 4 proposes the *dynamic suffix tree* for dynamic strings. Section 5 presents an incremental discovery algorithm based on the dynamic suffix tree. Section 6 extends the discovery framework and incremental discovery algorithm to multiple strings. Section 7 evaluates the proposed framework and algorithms. Section 8 remarks on future work and concludes the paper.

2 The Discovery Problem

Sequential data. A *string* S is a sequence of *characters* written $abcd\ldots$, where letters $a, b, c, \ldots$ represent characters. The first character is at position 1, the second at position 2, and so on. The characters in a string may be English characters in a text file, DNA base pairs, lines or source code, angles between edges in polygons, machines or machine parts in a production schedule, music notes and tempo in a musical score, and so forth. For example, a daily weather report can be represented by a string S in which characters are daily weather of type $(sky, temporature)$. A *substring* of S is a sequence of characters in S that are consecutive in position. The *length* of substring α, denoted $|\alpha|$, is the number of characters in α.

Discovery of sequential patterns. The *support* of substring α is the ratio of the number of positions in S at which α starts over $|S|$. $sup(\alpha)$ denotes the support of α. Let α and β be non-empty substrings of S such that the concatenation $\alpha\beta$ is a substring of S. A *sequential pattern* or simply *pattern* in S has the form $\alpha \rightarrow \beta$. The *support* of $\alpha \rightarrow \beta$ is $sup(\alpha\beta)$ and the *confidence* of $\alpha \rightarrow \beta$ is $sup(\alpha\beta)/sup(\alpha)$. For the user-specified minimum support $minisup$ and minimum confidence $miniconf$, a pattern is *interesting* if its support is at least $minisup$ and its confidence is at least $miniconf$. The *discovery problem* for string S finds all interesting patterns in S. Suppose that all interesting patterns in S were previously discovered and stored. Assume that $S = \alpha\beta\gamma$ is updated to $\alpha\delta\gamma$. With the update and old patterns as the input, the *incremental discovery* finds all interesting patterns in the updated S. We will extend these definitions to multiple strings in Section 6.

Example 2.1 *Consider string $S = abcebcdbc$, where each letter represents a character. $sup(bc) = 3/|S|$ and $sup(bcdbc) = 1/|S|$, so the support of $bc \rightarrow dbc$ is $1/|S|$ and the confidence of $bc \rightarrow dbc$ is $1/3$. The support of $b \rightarrow c$ is $3/|S|$ and the confidence of $b \rightarrow c$ is 1. If $minisup = 2/|S|$ and $miniconf = 1/3$, $b \rightarrow c$ is interesting, but $bc \rightarrow dbc$ is not. In fact, $b \rightarrow c$ is the only interesting pattern in S for such $minisup$ and $miniconf$.*

The discovery problem can be decomposed into two subproblems:

1. Find all substrings α of S such that $sup(\alpha)$ is at least $minisup$. Such substrings are called *frequent* substrings.

2. Use the frequent substrings to generate patterns. Here is a straightforward algorithm for this task. For every frequent substring β and every non-empty proper prefix α of β, output

pattern $\alpha \rightarrow \beta - \alpha$ if $sup(\beta)/sup(\alpha)$ is at least $miniconf$, where $\beta - \alpha$ is obtained by deleting the prefix α from β.

Essentially, the discovery problem requires to index substrings together with their support and position information. Let us review several existing techniques for serving this purpose.

The R-tree [G84] and R+-tree [SRF87] support operations on multi-dimensional data. They can be used for strings by treating the position as one dimension and substrings as the second dimension. However, since the number of possible substrings in a string of length n can be $O(n^2)$ and a substring can be scattered all over the string, the performance can be very bad. The technique of inverted lists [F85, TMS94, ZMD93] has been used to index a fixed set of "words" in a text database. For general-purpose strings for which the data processing unit is substrings, there is no clear cut of "words". Taking all possible substrings as "words" leads to the blowup of $O(n^2)$ words for a string of length n. The suffix tree indexing [C95, M76, LV89, U92, U93, W73] was previously used to speed up subsequent search of substrings. The update of the suffix tree was considered in [M76] for dynamic strings, by assuming that a position number never changes once assigned. Such position numbers do not correspond to the logical ordering of characters and are practically useless. However, several properties of the suffix tree is appealing for the discovery problem: the construction of the suffix tree takes linear time and linear space; the frequency and position information of substrings are readily available in the suffix tree; the suffix tree serves a natural and compact representation of sequential patterns. In this paper, we adopt the suffix tree for indexing substrings, with the focus on updating the suffix tree for solving the incremental discovery problem. The following briefly introduces the suffix tree.

The suffix tree [LV89, M76, S94, U92]. Assume that no suffix of S is a prefix of a different suffix of S. This can be satisfied by appending the unique delimiter \$ at the right extreme of S. S can be mapped to a tree T in which root-to-leaf paths are suffixes of S and terminal nodes represent uniquely starting positions of suffixes. Formally, the *suffix tree* T for S satisfies the following properties:

T1 each arc of T represents a non-empty substring of S,

T2 each non-terminal node of T, except the root, must have at least two offspring arcs,

T3 substrings represented by offspring arcs of the same node must begin with different characters.

T is a multiway Patrica tree and contains at most $|S|$ non-terminal nodes.

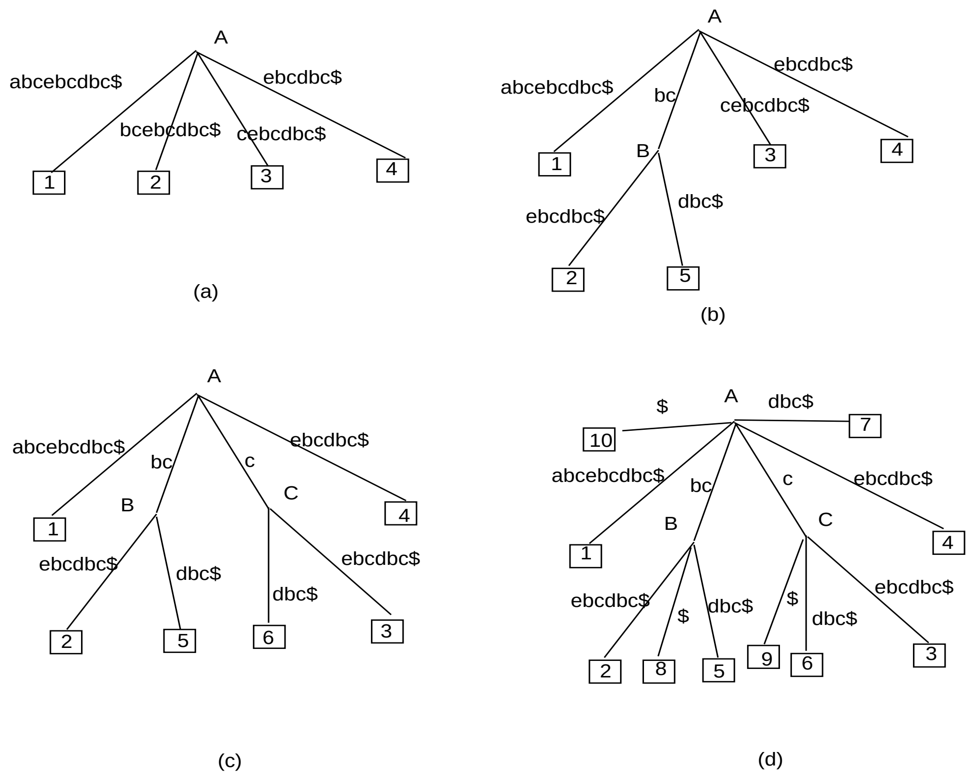


Figure 1: Construction of suffix trees

Example 2.2 *Consider the string $S = abcebcdbc\$$ in Example 2.1. We can build the suffix tree T of S by inserting suffixes into T one at a time, starting from the longest suffix abcebcdbc\$. In Figure 1, (a) shows the tree after the first four suffixes are inserted. Since none of these suffixes shares a prefix, they all go to different branches. A square box represents a terminal node and contains the starting position of the corresponding suffix. (b) shows the tree after the suffix bcdbc\$ is inserted. Since bcdbc\$ shares prefix bc with bcebcdbc\$, the arc for bcebcdbc\$ is split into two, one for bc and one for ebcdbc\$, so that T3 is satisfied. (c) is the tree after suffix cdbc\$ is inserted, in which the arc for cebcdbc\$ is split. (d) is the suffix tree after all suffixes of S are inserted.*

In Figure 1(d), for any substring α of S, by following the path from the root that spells out α we can find the subtree containing all starting positions of α in terminal nodes. For example, by following the path that spells out bc, i.e., arc (A, B) in this case, we find the root B of the subtree containing starting positions 2,5,8 of bc in its terminal nodes. The number of such terminal nodes can be stored at B to facilitate counting the occurrence of bc. Because of property T3, the path that spells out a substring is unique. Note that the implementation of the suffix tree stores the starting position and length of the substring associated with an arc, not the substring itself.

Let v be a non-terminal node in the suffix tree T. $subtree(v)$ denotes the subtree rooted at v. $string(v)$ denotes the substring spelled out by the path from the root to v. $count(v)$ denotes the number of terminal nodes within $subtree(v)$. The *locus* of substring α, denoted $locus(\alpha)$, is the first node in T encountered after α is spelled out by following a path from the root. All starting positions of α are found in the terminal nodes in $subtree(locus(\alpha))$. The number of starting positions of α in S is given by $count(locus(\alpha))$. Therefore, $sup(\alpha) = count(locus(\alpha))/|S|$. For example, in Figure 1(d), $locus(bc) = B$ and $locus(cd)$ is terminal node 6. The starting positions of bc are found in the terminal nodes within $subtree(B)$, i.e., terminal nodes 2,5,8. The starting positions of cd are found in the terminal nodes within $subtree(6)$, that is, node 6.

From the suffix tree T for S with $count(v)$ stored at every non-terminal node v, we can extract all interesting sequential patterns from T. For every path from the root that spells out substring β and for every non-empty proper prefix α of β, if

$$count(locus(\beta))/|S| \geq minisup \text{ and}$$
$$count(locus(\beta))/count(locus(\alpha)) \geq miniconf,$$

we output pattern $\alpha \rightarrow \beta - \alpha$ together with the support

$$count(locus(\alpha))/|S|$$

and the confidence

$$count(locus(\beta))/count(locus(\alpha)),$$

where $\beta - \alpha$ is obtained by deleting the prefix α from β. For a fixed $minisup$, only the upper part of T consisting of all nodes v such that $count(v)/|S| \geq minisup$ is of interest. However, there are good reasons for materializing the whole suffix tree. As the user tunes $minisup$, the part of T of interest will shrink or grow, and by materializing the suffix tree the user can tune the interestingness level of patterns without suffering from the delay caused by reconstructing the suffix tree. Most importantly, the suffix tree is materialized for the incremental discovery of patterns.

3 Existing Suffix Tree Algorithms

We review existing suffix tree algorithms and point out their pitfalls for dynamic strings. This motivates a modification of the suffix tree for solving the incremental discovery problem in the next section.

3.1 Construction

The naive construction of the suffix tree by descending a path from the root for each suffix, as in Example 2.2, immediately leads to the quadratic time complexity. By exploring the relationship between every two consecutive suffixes, the construction can be reduced to a linear time [M76, W73, U92]. The following is the construction in [M76]. ([M76] is chosen because it considered the update problem of the suffix tree and others didn't.) Consider two non-terminal nodes u and v. There is a *suffix-link* from u to u if u is the root; otherwise, there is a *suffix-link* from u to v if $string(u) = x\alpha$ and $string(v) = \alpha$, respectively, where x is a character and α is a substring. Let suf_i denote the suffix of S beginning at position i. If $x\alpha$ is a prefix of suf_i, then α is a prefix of suf_{i+1}. Therefore, after inserting suf_i into the suffix tree, the suffix-link at node u points to a place to insert the rest of suf_{i+1}, i.e., the part of suf_{i+1} without the prefix α.

Let T_i denote the suffix tree after $suf_1, \ldots, suf_i$ are inserted. The suffix tree for S is constructed in the order $T_1, T_2, \ldots$, where T_{i+1} is obtained by inserting suf_{i+1} into T_i. Initially, T_0 contains only the root. Consider inserting suf_{i+1} into T_i. On the path for suf_i, let u denote the lowest non-terminal node containing a suffix-link and let w denote the lowest non-terminal node. u and w were visited when inserting suf_i. Assume that $string(u) = x\alpha$ and $string(w) = x\alpha\beta$, respectively, where x is a character and α and β are substrings of S. Suppose that every non-terminal node in T_i, except possibly w, contains a suffix-link. This property initially holds for T_0 and will be inductively established for T_{i+1}.

To insert suf_{i+1}, we follow the short-cut provided by the suffix-link at u. From the definition, for the node v pointed by the suffix-link, $string(v)$ is a prefix of suf_{i+1}. Starting from v, we descend the tree along a sequence of arcs that spells out β. If β does not end exactly at a node, the last arc descended is split and a new node d is inserted at the end of β. The suffix-link of w is assigned to point to d if it is not assigned yet. This step is called *rescanning* in [M76], which establishes the suffix-link at w. The search for the rest of suf_{i+1} (i.e., the part without the prefix $\alpha\beta$) continues from d deeper into the tree. The search eventually "falls out of the tree" because suf_{i+1} is not a prefix of any suffix in the tree. If the search does not fall out exactly at a node, the last searched arc is split and a new non-terminal node is inserted, which, by the assumption on T_i, is the only non-terminal node containing no suffix-link. Finally, a new terminal node is inserted for the rest of suf_{i+1} that falls out of the tree. This step is called *scanning* in [M76].

Example 3.1 *Consider Figure 1(d), where the suffix-link at the root points to itself and the suffix-*

link at B points to C. After inserting suffix suf_8, the suffix-link at B provides a short-cut to insert the part of suf_9 without prefix c, i.e., \$. In this case, $u = w = B$, $x = b$, $\alpha = c$, and $\beta = \emptyset$. Since $\beta = \emptyset$, no new non-terminal node is created during rescanning. Also, since suf_9 falls out of the tree exactly at node C, no new non-terminal node is created during scanning. For a suffix tree, the short-cut provided by suffix links will avoid a long traversing from the root.

The detail of the above construction can be found in [M76].

3.2 Update

The update of the suffix tree for dynamic strings was considered in [M76]. Assume that the suffix tree T for S was materialized and that $S = \alpha\beta\gamma$ is updated to $\alpha\delta\gamma$, where $\alpha, \beta, \delta, \gamma$ are substrings. To reflect the update in T, [M76] determines which suffixes (i.e., paths) in T are affected and updates such suffixes. On one hand, if a suffix is too short to contain any part of β, the suffix is not affected. On the other hand, if a suffix is so long that β is entirely buried in the terminal arc of the suffix, the change of the suffix is reflected by the update in string S. Therefore, for a suffix to be affected, it must contain part of β but does not properly contain $\beta\gamma$ in its terminal arc. Such suffixes were called β-splitters in [M76] and are defined formally below.

Let α^* be the longest suffix of α that occurs in at least two different positions in $S = \alpha\beta\gamma$. With respect to the update from $\alpha\beta\gamma$ to $\alpha\delta\gamma$, *β-splitters* are suffixes of the form $\epsilon\gamma$, where ϵ is a non-empty suffix of $\alpha^*\beta$. See Figure 2 for an illustration. Consider a suffix suf_i. If suf_i is longer than the longest β-splitter $\alpha^*\beta\gamma$, suf_i will properly contain $\beta\gamma$ in its terminal arc because its prefix proceeding $\beta\gamma$ does not repeat in S. From the above discussion, suf_i is not affected by the update. If suf_i is shorter than the shortest β-splitter $x\gamma$, where x is a single character, suf_i does not contain any part of β, thus, is not affected either. Therefore, β-splitters are the only suffixes affected by the update. In testing, suf_i is a β-splitter if and only if suf_i starts on the left of γ, but the terminal arc of suf_i does not properly contain $\beta\gamma$. The test requires to compare the starting position of suf_i with that of γ and compare the position stored on the terminal arc for suf_i with the starting position of $\beta\gamma$. The starting positions of γ and $\beta\gamma$ are known from the update specification. The other position information needed for the test can be found in the terminal node and terminal arc for suf_i. Therefore, by visiting the terminal node and terminal arc for suffix suf_i, we can determine whether suf_i is a β-splitter.

The strategy of replacing β with δ suggested by [M76] is to delete all β-splitters from T and

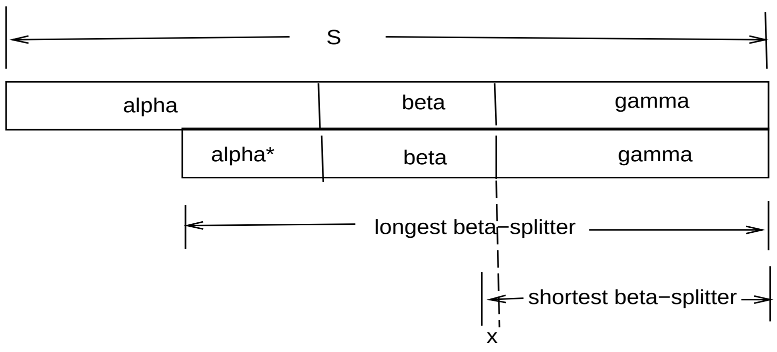


Figure 2: β-splitters

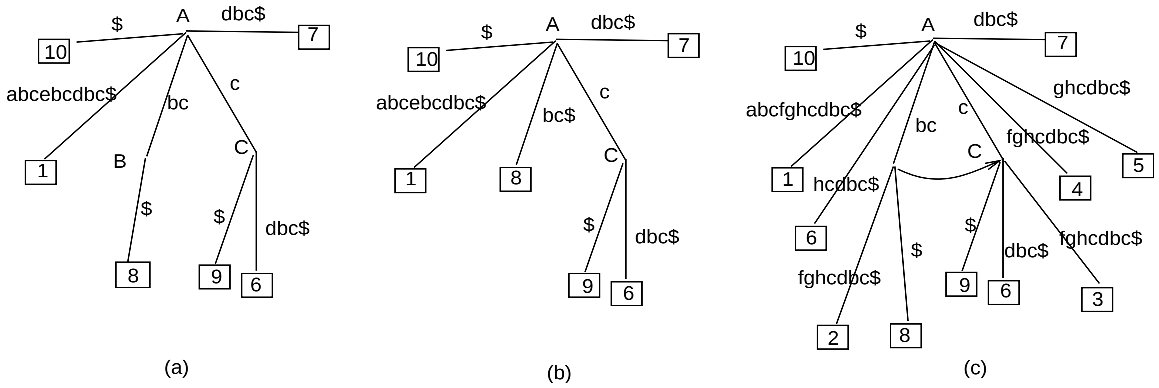


Figure 3: Example 3.2

insert into T all *δ-splitters* of the form $\omega\gamma$, where ω is a non-empty suffix of $\alpha^*\delta$. Intuitively, the δ-splitters are in lieu of the β-splitters accounting for the replacement of β with δ. To find all β-splitters, it is assumed that terminal nodes are double chained up in the order of the position. Let us call this chain the *position chain.* The deletion proceeds from the shortest β-splitter to the longest β-splitter. The terminal node for the shortest β-splitter is the node containing position number $k-1$, where k is the starting position of γ. Deleting a β-splitter corresponds to deleting its terminal arc and possibly merging two arcs. See Example 3.2 below. Inserting δ-splitters is the same as inserting suffixes in the construction algorithm, proceeding from the longest δ-splitter to the shortest. The longest δ-splitter $\alpha^*\delta\gamma$ is obtained from the longest β-splitter $\alpha^*\beta\gamma$ by replacing β with δ.

Example 3.2 *Suppose we replace $\beta = eb$ with $\delta = fgh$ in $S = abcebcdbc\$$. Then $\alpha = abc$ and $\gamma = cdbc\$$. Initially, the suffix tree T for S is given in Figure 1(d). The implementation of the suffix tree stores only the starting position and length of the substring associated with an arc, not* *the substring itself. The position chain is omitted here because it can be derived from the position in terminal nodes. The longest repeating suffix of* α *is* $\alpha^* = bc$*. There are four* β*-splitters:* $bceb\gamma$*,* $ceb\gamma$*,* $eb\gamma$*,* $b\gamma$*. These* β*-splitters are found and deleted as follows. First, we find the terminal node for the shortest* β*-splitter at the node containing position number 5, i.e., one position to the left of the starting position of* γ*. We follow the position chain towards lower position numbers. Since the terminal arcs for positions 5, 4, 3, 2 do not properly contain* $\beta\gamma$*, these terminal nodes represent* β*-splitters and are deleted. By the time* β*-splitter* $bceb\gamma$ *is deleted, as in Figure 3(a), node B is left with only one offspring arc, violating property T2. So the two arcs on the path are merged into one arc, as in Figure 3(b).*

The longest δ*-splitter is obtained from the longest* β*-splitter by replacing* β *with* δ*, that is,* $bcfgh\gamma$*. The five* δ*-splitters* $bcfgh\gamma$*,* $cfgh\gamma$*,* $fgh\gamma$*,* $gh\gamma$*,* $h\gamma$ *are inserted into the tree in Figure 3(b), proceeding from the longest to the shortest. The tree after inserting all* δ*-splitters is given in Figure 3(c). The substring associated with terminal arc* $(A, 1)$ *gets updated by default when* eb *is replaced with* fgh *in the string* S*. The reader may note that there are two terminal nodes in Figure 3(c) containing position number 6. We will discuss this problem shortly.*

The suffix tree was well studied in the literature. We omit the formal description of its algorithms, which can be found in [B92, M76, S94, U92, W73]. Our concern is whether the suffix tree can be applied to solve the incremental discovery problem and what modifications and extensions are needed.

3.3 The pitfalls to handle updates

Unfortunately, the above update algorithm does not work appropriately. First, [M76] adopted a position numbering scheme in which a position number never changes once assigned. That positioning does not correspond to the logical ordering of characters as perceived by the user, therefore, are practically useless. Second, [M76] failed to deal with dangling references to deleted positions in the suffix tree. For example, in Figure 1(d) after deleting character c at position 3, the positions on arcs (A, B) and (A, C) will refer to unexpected places, which clearly posts a big problem for later insertion of suffixes. On the other hand, maintaining the logical position in the suffix tree is a very expensive operation. In Example 3.2, replacing eb with fgh requires to change positions 6,7,8,9 to positions 7,8,9,10. Making such changes in the suffix tree requires to access many nodes and arcs. A similar problem exists if substrings instead of positions are stored in the suffix tree. Finally, the above update algorithm does not address the change of support of substrings, as required by the discovery problem. Therefore, the suffix tree in the literature is not suitable for indexing dynamic strings. Despite these problems, however, the idea of updating the suffix tree by deleting β-splitters and inserting δ-splitters is appealing because it avoids to rebuild the whole tree. The dynamic suffix tree to be proposed below will borrow this idea and take care of the above mentioned problems.

4 The Dynamic Suffix Tree

We assume that large string S is stored on a number of disk pages. In such a disk-based environment, the position does not provide sufficient information to find characters because consecutive characters may not be stored on consecutive disk pages. Instead, characters are accessed through their disk addresses. The disk address of a character consists of a disk page number and an offset of the character within the disk page. Unlike the position, insertion or deletion affects only the address of characters in the disk pages containing inserted or deleted characters; the address of untouched pages remains unchanged. This motivates a new representation of the suffix tree for dynamic strings, called the dynamic suffix tree, in which disk addresses rather than positions are stored. Let us consider the data structures and operations needed for the dynamic suffix tree.

4.1 The B-tree(P)

First, we need a disk-based structure to store the string S. The storage structure must provide efficient mappings between the position and the address. The mapping from addresses to positions is needed to find the position of patterns, and the mapping from positions to addresses is needed to perform updates on string S at a specified position. We propose a variation of the B-tree, called the *B-tree on position* or simply *B-tree(P)*, for storing S.

The B-tree(P). An entry in a non-terminal node of the B-tree(P) has the form $< c, Pr >$, where Pr is the pointer to a child node, and c, called a *c-value*, is the number of characters indexed in the subtree under this entry. In the B-tree(P) of *degree* (m, M), the following invariants hold for a non-terminal node v containing entries $< c_1, Pr_1 >, \ldots, < c_p, Pr_p >$:

P1 for $1 \leq i \leq p$, exactly c_i characters are indexed in the subtree under branch Pr_i,

P2 if v is the root, $p \geq 2$; otherwise, $m \leq p \leq M$.

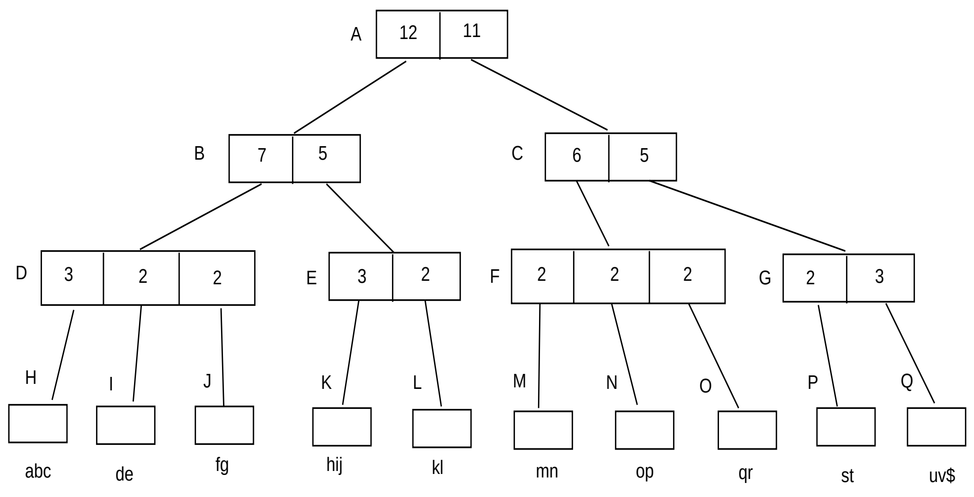


Figure 4: A B-tree(P) for string $S = abcdefghijklmnopqrstuv\$$

P3 all terminal nodes must be at the same level of the tree and characters are contained in terminal nodes only.

A child node also has a pointer to its parent node, which is needed for mapping addresses to positions explained later.

Example 4.1 *Consider string* $S = abcdefghijklmnopqrstuv\$$*, where each letter represents a character. Figure 4 shows a B-tree(P) of degree* $(2, 4)$ *for* S*. The terminal level of the tree contains all characters. The c-value 6 in node* C *indicates that under the first branch of* C *there are exactly 6 characters. It can be verified that every c-value correctly gives the number of characters in the subtree under the entry containing the c-value.*

Mapping positions to addresses. Searching for the address corresponding to position p is done by descending the B-tree(P) along a root-to-leaf path $u_1, u_2, \ldots, u_k$. As a branch is descended, c-values on the left of that branch in the current node are accumulated. The difference between p and the current accumulative sum determines which branch to descend at the next level. On reaching a terminal node u_k, the difference between p and the accumulative sum gives the offset within u_k of the character at position p.

For example, suppose we want to find character i at position 9 in string S in Figure 4. Initially, the accumulative sum $X = 0$. The difference $9 - X$ suggests that the left branch of A be descended, with X unchanged. At node B, the difference $9 - X = 9$ is more than the first c-value, 7, but less than the sum of the first and second c-values, so the second branch is descended and X is changed to 7. At node E, since the difference $9 - X = 2$ is less than the first c-value, the first branch is descended and $X = 7$ remains unchanged. Then the address corresponding to position 9 is given by node K and offset $9 - X = 2$.

Mapping addresses to positions. To find the position corresponding to the address given by terminal node u and offset i, we traverse from node u towards the root and accumulate all c-values on the left of the traversed branch, in a way similar to the descending of the B-tree(P) above. After reaching the root, the accumulative sum gives the position searched.

For example, suppose we want to find the position of the second character in terminal node J in Figure 4, i.e., character g. Initially, the accumulative sum X is set to 2, that is, the given offset within J. We visit the parent D and add to X the c-values in D on the left of the branch just traversed upwards, giving $X = 7$. Next, we visit parent B, and $X = 7$ remains unchanged because the branch traversed is left most. Finally, on reaching the root, we have $X = 7$. Therefore, the position of the second character in terminal node J is 7.

Updating the B-tree(P). Inserting and deleting a substring at a specified position is performed in two phases. The search phase finds the terminal nodes to insert or delete the substring, and the propagation phase inserts or deletes entries for nodes inserted or deleted at lower levels. The paths descended in the search phase are saved on the stack and used by the propagation phase. Insertion at one level may cause more than one node to be created, and entries in a node may be split and redistributed to satisfy property P2. For a deletion specified by starting and ending positions, the search phase looks for the left and right limits of the deletion by descending the B-tree(P) along two paths. All characters at the terminal level between the two limits are deleted. Underflow nodes are merged and entires may be redistributed to satisfy property P2. In the propagation phase, the c-value at an entry is computed by summing all c-values in the child node under the entry, if the child is non-terminal, or by the number of characters in the child node under the entry, if the child is terminal. Figure 5 illustrates affected nodes for a general update, where the shaded area denotes the nodes that are inserted or deleted at one level. Importantly, for both insertion and deletion, addresses of characters contained in untouched terminal nodes are not affected, though their positions may have been changed.

We like to mention that the B-tree(P) has all the nice properties of the B-tree, i.e., the balanced height, a large branching factor, localizing the search to a single path, etc. Unlike the B-tree, however, a number of nodes could be inserted or deleted at a level of the B-tree(P), depending on the size of the substring inserted or deleted, but not on the size of the database. We omit

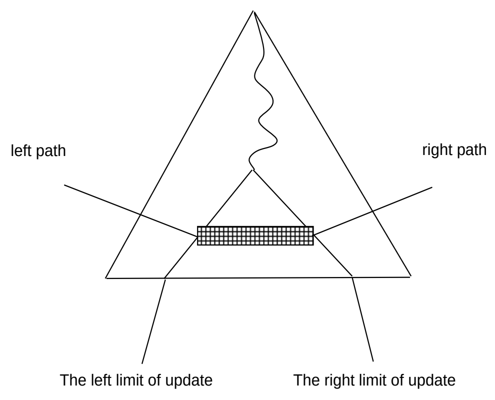


Figure 5: Nodes affected by insertion and deletion

the precise description of these operations on the B-tree(P). We hope that informal discussion and examples can better bring out the working idea.

4.2 The dynamic suffix tree

We consider how to modify the suffix tree for indexing dynamic string S stored in the B-tree(P). Let $p.i$ denote the address of the character at the offset i on disk page p. Instead of storing starting and ending positions of substrings in the suffix tree, we store a *reference pair* of the form $(addr, l)$ on each arc, where $addr$ and l are the starting address and length of the substring associated with the arc. We call such a suffix tree the *dynamic suffix tree*. The dynamic suffix tree for the suffix tree in Figure 1(d) is shown in Figure 6(c), with the B-tree(P) in Figure 6(a). For convenience, the starting positions of suffixes are given next to terminal nodes. The length l on a terminal arc is not used because the substring on a terminal arc always extends to the end of the string. To apply the dynamic suffix tree to solve the incremental discovery problem, the problems mentioned in subsection 3.3 must be addressed. This is the topic of the next section.

5 The Incremental Discovery Algorithm

Assume that string S is updated from $\alpha\beta\gamma$ to $\alpha\delta\gamma$. The update of S is performed in the B-tree(P) by deleting β and inserting δ, as in Section 4. We focus on the update of the dynamic suffix tree T for S. Let $pos(addr)$ denote the position corresponding to the address $addr$.

It is important that the change in the dynamic suffix tree is limited to only affected paths, that is, β-splitters and δ-splitters. The dynamic suffix tree is updated in two phases, corresponding to deleting β-splitters and inserting δ-splitters in Section 3. The first phase deletes all β-splitters and adjusts all reference pairs $(addr, l)$ such that the position range $[pos(addr), pos(addr)+l-1]$ intersets with the position range of β. If there is an intersection, $(addr, l)$ is *referring* to deleted characters and is called *dangling*. A dangling reference pair must be replaced with a non-dangling reference pair that represents the same substring. The second phase inserts δ-splitters and adjusts the support affected by the deletion of β-splitters and insertion of δ-splitters. Since inserting δ-splitters needs to access substrings associated with arcs, references pairs must be replaced by non-dangling reference pairs in the first phase so that the tree is free of dangling references. The big question is where to find all dangling reference pairs in the suffix tree. The following theorem gives the answer.

Theorem 5.1 *Suppose that S is updated from $\alpha\beta\gamma$ to $\alpha\delta\gamma$. If the reference pair on a non-terminal arc (u, v) is dangling, (u, v) is on a β-splitter with respect to this update.*

From Theorem 5.1, we can find all dangling reference pairs on β-splitters, or equivalently, we can adjust dangling reference pairs by accessing only ancestor arcs of deleted terminal nodes. To prove Theorem 5.1, we say that references pairs $(addr_1, l_1), \ldots, (addr_k, l_k)$ along a path in the dynamic suffix tree are *continuous* if the last position referred to by $(addr_i, l_i)$ proceeds immediately the first position referred to by $(addr_{i+1}, l_{i+1})$, that is, $pos(addr_i) + l_i = pos(addr_{i+1})$, for $1 \leq i < k$.

Proof of Theorem 5.1: Consider the suffix tree for $\alpha\beta\gamma$ constructed by inserting all suffixes of $\alpha\beta\gamma$. For any non-terminal arc (x, y), there is at least one arc (y, z) from y such that the reference pairs on (x, y) and (y, z) are continuous. In fact, we can choose the arc (y, z) such that (x, y) and (y, z) are produced by splitting a single arc during the construction of the suffix tree. Thus, there is a path $(x, y), \ldots, (x', y')$ in the suffix tree to a terminal node, on which all reference pairs are continuous. Now we consider the arc (u, v) in the theorem. Let $(u, v), \ldots, (u', v')$ be the path to a terminal node on which all reference pairs are continuous. Since the reference pair on (u, v) refers to a deleted character in β, the terminal arc (u', v') does not properly contain $\beta\gamma$, otherwise, the reference pairs on path $(u, v), \ldots, (u', v')$ are not continuous. In other words, the terminal node v' represents a β-splitter containing arc (u, v). Then the theorem follows because the update algorithm below preserves the continuity of reference pairs required. □

The incremental discovery algorithm for string update from $\alpha\beta\gamma$ to $\alpha\delta\gamma$. It is assumed that the dynamic suffix tree for string $\alpha\beta\gamma$ is stored.

Phase 1.

Step 1a. Delete all β-splitters, as in Section 3.2. Let Δ^- contain the parents of deleted terminal nodes. (If deleting a β-splitter causes two arcs (u,x) and (x,v) to be merged into one arc (u,v), Δ^- contains u instead of x.

Step 1b. Mark all nodes in Δ^-, their ancestors, and connecting arcs. To avoid repeated markings, we start with nodes in Δ^-, walk up the tree and mark nodes and arcs until encountering either the root or a marked node. From Theorem 5.1, only reference pairs on marked arcs are affected.

Step 1c. Adjust reference pairs on marked arcs. This is done by the postorder traversal of all marked arcs. For each arc (u,v) being traversed, let $(addr1, l)$ be the reference pair on (u,v) and let $(addr2, m)$ be the reference pair on any offspring arc of v. We replace $(addr1, l)$ on (u,v) with $(addr3, l)$, where $addr3$ is the address corresponding to position $pos(addr2) - l$. The address $addr3$ can be found by mapping $addr2$ to $pos(addr2)$ and mapping $pos(addr2) - l$ to the corresponding address using the B-tree(P). We say that $addr3$ is at *l-distance* from $addr2$.

By definition, $(addr3, l)$ and $(addr2, m)$ are continuous. Let us complete the proof of Theorem 5.1. Assume that before updating the dynamic suffix tree, every non-terminal node has a path to a terminal node on which all reference pairs are continuous. An induction on the traversing order can show that any marked node has a path to a terminal node on which all reference pairs are continuous. This completes the proof of Theorem 5.1.

The reason that $(addr1, l)$ on (u,v) can be replaced with $(addr3, l)$ in Step 1c is because they refer to the same substring. From the above discussion, for any offspring arc (v,w) at v there is a path $(v,w), \ldots, (v', w')$ to some terminal node w' on which all reference pairs are continuous, and $addr3$ is at l-distance from the address on (v,w). This implies that replacing $addr1$ with $addr3$ does not change the suffix represented by w'. Thus, $(addr1, l)$ and $(addr3, l)$ represent the same substring.

Phase 2

Step 2a. Insert all δ-splitters, as in Section 3.2. Let Δ^+ contain all parents of inserted terminal nodes.

Step 2b. Mark all nodes in Δ^+, their ancestors and connecting arcs, similar to Step 1b.

Step 2c. Adjust $count(v)$ for marked nodes v. This is done by the postorder traversal of all arcs marked in either Step 1b or 2b. The change of $count(v)$ at a node v in $\Delta^- \cup \Delta^+$ is the signed number of terminal nodes deleted and inserted under v. The change of $count(v)$ at a marked node v not in $\Delta^- \cup \Delta^+$ is the sum of the changes of $count(u)$ for all marked children u of v. The new $count(v)$ at a marked node v is equal to the old $count(v)$ plus the change of $count(v)$.

Example 5.1 *Consider string $S = abcebcdbc\$$ stored in the B-tree(P) in Figure 6(a). Figure 6(c) shows the dynamic suffix tree for S with the suffix-links and position chain omitted. A reference pair $(p.i, l)$ denotes the substring starting at address $p.i$ and having length l. Each terminal node in the dynamic suffix tree contains the starting address of the represented suffix. For convenience, the starting position of suffixes are given next to terminal nodes in Figure 6(c). $count(v)$ for a non-terminal node v is printed inside the node. For example, $suf_2 = bcebcdbc\$$ is represented by path $(F, A), (A, 2)$ and has the starting address $X.2$, $suf_5 = bcdbc\$$ is represented by path $(F, A), (A, 5)$ and has the starting address $Y.1$. The reference pairs on path $(F, A), (A, 2)$ are continuous, but the reference pairs on path $(F, A), (A, 5)$ are not.*

Consider the update U from $abcebcdbc\$$ to $acbdeebcdbc\$$, that is, replace the first bc by $cbde$. Thus, $\alpha = a$, $\beta = bc$, $\delta = cbde$, $\gamma = ebcdbc\$$ in the update from $\alpha\beta\gamma$ to $\alpha\delta\gamma$. The following steps are performed.

First, the B-tree(P) in Figure 6(a) is descended to search for the left and right limits of β. The search leads to terminal nodes X and W and all characters in β are deleted. Then δ is inserted into the B-tree(P). The offset of characters in W is affected by the update. For example, the character e originally having offset 2 in W has offset 3 after update U, as in Figure 6(b). To reflect this change, the actual update U' performed on the dynamic suffix tree T is $\alpha\beta'\gamma'$ to $\alpha\delta'\gamma'$, where $\beta' = bce$, $\delta' = cbdee$, $\gamma' = bcdbc\$$. That is, U' is U extended to all characters in node W.

Then update U' is performed on the dynamic suffix tree in Figure 6(c). Since character a does not repeat, $\alpha^* = \emptyset$. There are three β'-splitters: $bce\gamma'$, $ce\gamma'$, $e\gamma'$, represented by terminal nodes containing addresses $X.2, W.1, W.2$, corresponding to positions 2,3,4. There are five δ'-splitters: $cbdee\gamma'$, $bdee\gamma'$, $dee\gamma'$, $ee\gamma'$, $e\gamma'$. Figure 6(d) shows the suffix tree after Step 1a in which the β'-splitters are deleted . Δ^- contains nodes A, B, F, the parents of deleted terminal nodes. Marked arcs are in bold face. Figure 6(e) shows the tree after adjusting reference pairs in Step 1c. For example, $(X.2, 2)$ in Figure 6(d) is adjusted to $(Y.1, 2)$ in Figure 6(e), where address $Y.1$ is at

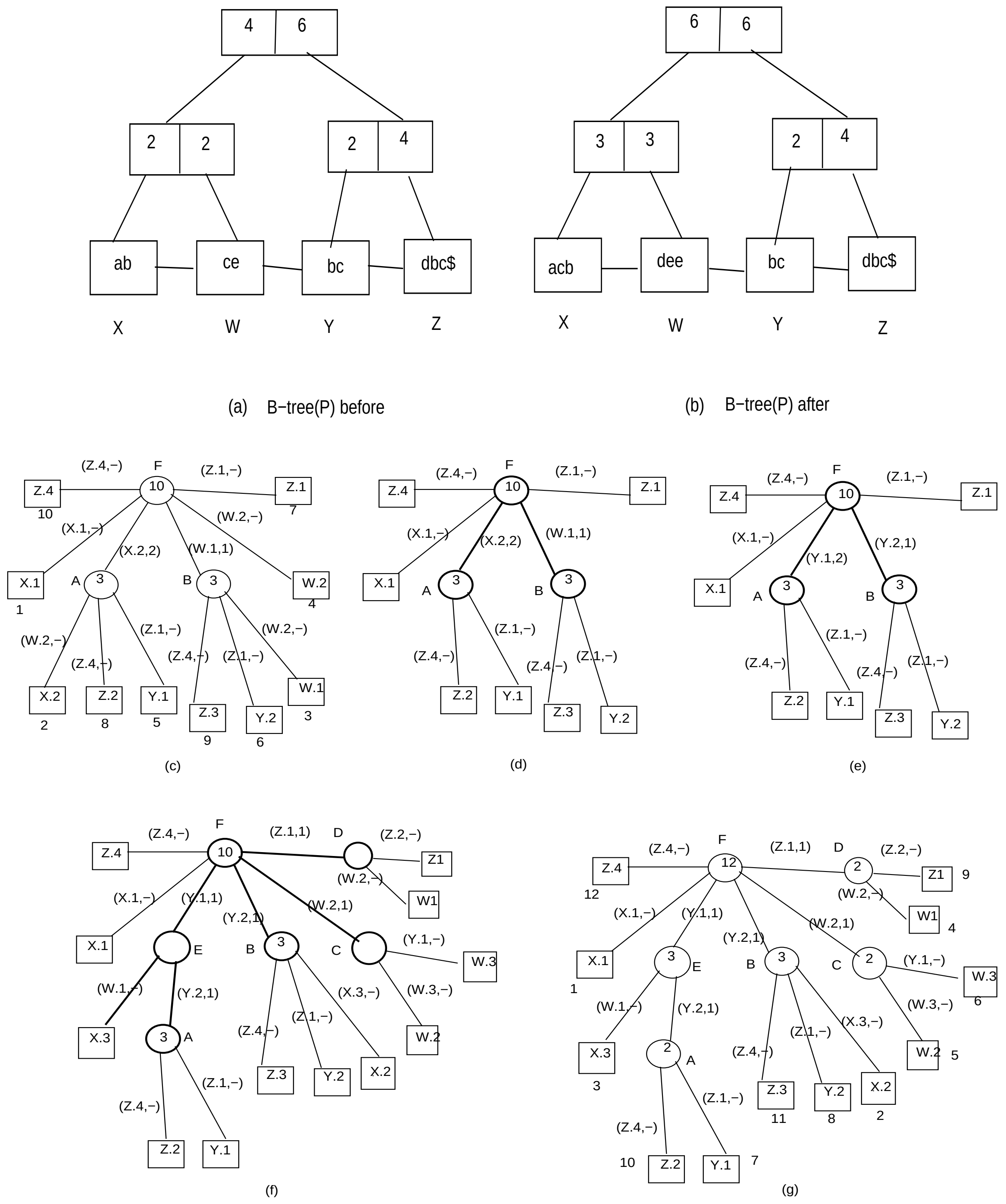


Figure 6: Example 5.1

2-distance from address $Z.1$. *Similarly,* $(W.1, 1)$ *on* (F, B) *is adjusted to* $(Y.2, 1)$.

Figure 6(f) shows the tree after inserting the δ'*-splitters in Step 2a. From Figure 6(b), the five* δ'*-splitters* $cbdee\gamma'$, $bdee\gamma'$, $dee\gamma'$, $ee\gamma'$, $e\gamma'$ *start at addresses* $X.2, X.3, W.1, W.2, W.3$, *respectively, and they are represented in Figure 6(f) by terminal nodes containing these addresses. In descending an arc, the substring associated with the arc is retrieved using the reference pair on the arc. Since new terminal nodes are inserted under* B, C, D, E, Δ^+ *contains* B, C, D, E. *All arcs marked in Step 1b and 2b are in bold face in Figure 6(f). Finally, Step 2c traverses all marked arcs in the postorder to compute the change of* $count(v)$ *for ancestors* v *of nodes in* $\Delta^- \cup \Delta^+$. *Figure 6(g) shows the dynamic suffix tree after the update* U'.

6 Extension to Multiple Strings

We extend the discovery framework and algorithms to multiple strings.

6.1 The problems

Consider a set $S = \{S_1, \ldots, S_k\}$ of strings. Each string S_i is identified by an unique identifier i and is delimited by an unique symbol $\$_i$. Let $|S| = |S_1| + \ldots + |S_k|$. In the case of multiple strings, there are two ways to define the support of sequential patterns. The *1-support* of substring α is the ratio $m/|S|$, where m is the total number of occurrences of α in $S_1, \ldots, S_k$. The *2-support* of substring α is the ratio n/k, where n is the number of strings in S in which α occurs. Let $sup_1(\alpha)$ and $sup_2(\alpha)$ denote the 1-support and 2-support of α, respectively. For $i = 1, 2$, the *i-support* of sequential pattern $\alpha \rightarrow \beta$ is $sup_i(\alpha\beta)$, and the *i-confidence* of sequential pattern $\alpha \rightarrow \beta$ is the ratio $sup_i(\alpha\beta)/sup_i(\alpha)$. With respect to the user-specified $minisup$ and $miniconf$, the *i-discovery problem* is to find all *i-interesting* sequential patterns, i.e., sequential patterns with i-support not less than $minisup$ and i-confidence not less than $miniconf$. Given a set S of strings, the set of i-interesting sequential patterns for S, and an update that either adds a new string to S or removes an old string from S, the *i-incremental discovery problem* is to find the set of i-interesting sequential patterns for the updated S. The problems defined in Section 2 are the 2-discovery problem and 2-incremental discovery problem for the special case of a single string.

6.2 The algorithms

The dynamic suffix tree and the incremental discovery algorithm can be extended to multiple strings $S = \{S_1, \ldots, S_k\}$. A straightforward extension is to maintain a separate dynamic suffix tree for each string S_i. This wastes storage because no path can be shared among different strings. In addition, it needs to traverse every suffix tree to determine the support of a substring. The same is true of finding all positions of a substring. Another approach is to represent all strings in a single dynamic suffix tree, based on the *generalized suffix tree (GST)* [H92] designed for a set of static strings. In the following, we extend the GST to dynamic strings to solve the incremental discovery problems for multiple strings.

In the case of multiple strings, the position of a character in a string consists of a string identifier and a position number within that string. First, the B-tree(P) in Section 4 is extended to multiple strings $\{S_1, \ldots, S_k\}$ as follows. The concatenation $1S_1 2S_2 \ldots kS_k$ is stored at the terminal level of the B-tree(P) for S, where i is the identifier of S_i. In an entry of the B-tree(P), the c-value has the form $i.c$, where i is the identifier for S_i and c is the number of characters in S_i in the subtree under the entry. When searching or updating the B-tree(P), the accumulative sum and comparison of c-values are performed only for those carrying the same identifier i. The mapping from addresses to positions returns both a string identifier and a position number within the string. With these modifications, the generalization of search and update algorithms of the B-tree(P) is routine.

Now we extend the dynamic suffix-tree T to multiple strings $\{S_1, \ldots, S_k\}$. T is constructed by inserting all suffixes of S_i, $i = 1, \ldots, k$. Since S_i has an unique delimiter $\$_i$, suffixes of different S_i are represented by different terminal nodes in T. All terminal nodes for suffixes in the same string S_i are chained up by the position chain to facilitate deletion of β-splitters. Note that each terminal node is on exactly one position chain.

A string S_i is deleted by deleting all suffixes of S_i from T. To find the suffixes of S_i, a B-tree on string identifiers i is maintained. For every identifier i, there is a pointer $head_i$ at the terminal level of the B-tree that points to the beginning of the position chain for S_i. The suffixes of S_i are found by searching the B-tree using the search key i, entering the position chain pointed by $head_i$, and scanning the position chain. The end of the position chain is marked by a special symbol. See Figure 7. All suffixes of S_i are deleted as β-splitters are in Section 3. As for a single string case, dangling reference pairs caused by deletion must be adjusted. A reference pair $(addr, l)$ is dangling if it refers to a character of the deleted S_i, which can be found out by mapping address

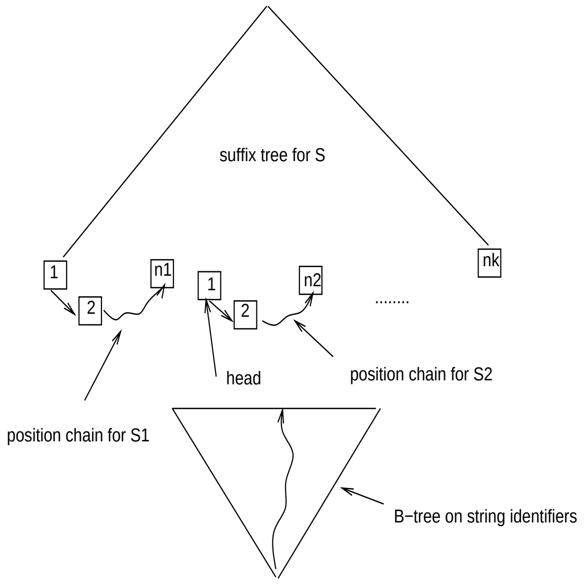


Figure 7: Locating suffixes of a string

addr to position through the generalized B-tree(P) for multiple strings. A new string S_i is inserted by inserting all its suffixes into the suffix tree and chaining up new terminal nodes in the position chain for S_i. An entry for identifier i is inserted into the B-tree with pointer $head_i$ pointing to the beginning of the new position chain.

The update of $count(v)$ at a non-terminal node v depends the type of the incremental discovery problem. For the 1-incremental discovery problem, $count(v)$ is equal to the number of terminal nodes in $subtree(v)$ and the update of $count(v)$ is the same as in Section 5. For the 2-incremental discovery problem, $count(v)$ is equal to the number of distinct string identifiers contained in terminal nodes in $subtree(v)$. Hence, if $subtree(v)$ contains a terminal node for the inserted or deleted string, $count(v)$ is increased or decreased by one, respectively. Therefore, $count(v)$ can be updated by the postorder traversal of arcs on splitters as in Section 5. With these modifications, the algorithm in Section 5 provides a solution to the incremental discovery problem for multiple strings.

7 Evaluation

This section evaluates the performance and discovery power of the proposed framework and algorithm. We report the study only for the case of a single string. We didn't find much difference for the case of multiple strings.

7.1 Cost analysis

We measure the cost of the update by the number of tree node access. This is reasonable because trees are stored on the disk and accessing tree nodes is the dominating activity performed by the algorithm. Let $|\alpha|$ denote the number of characters in substring α. Consider the update $\alpha\beta\gamma$ to $\alpha\delta\gamma$. From [M76], the number of node access to delete a β-splitter is no more than 3 and the average number of node access to insert a δ-splitter is no more than 3. Therefore, deletion and insertion of splitters in Step 1a and Step 2a can be done in $3(|\alpha^*\beta| + |\alpha^*\delta|)$ on average, where $|\alpha^*\beta| + |\alpha^*\delta|$ is equal to the number of α-splitters and β-splitters. In Steps 1b,1c,2b,2c, each node on splitters is accessed at most four times.

Another part of the cost comes from accessing the B-tree(P). In Step 1c, adjusting the reference pair on an arc requires to map an address to the corresponding position and map a position to the corresponding address. Also, the update $\alpha\beta\gamma$ to $\alpha\delta\gamma$ requires to delete β and insert δ in the B-tree(P). These costs depend on the height of the B-tree(P) and the length of the substring updated, not on the data size. It is commonly known that the height of a balanced B-tree with a large branching factor is very small, even for a very large database. Therefore, the total cost of an update is the number of distinct nodes on splitters with a small constant factor. For a large string and a local update, which is a scenario assumed for most dynamic environments, it is expected that only a small number of suffixes (i.e., paths) are affected compared to the whole suffix tree, therefore, the incremental approach is more efficient than the naive approach in most cases, as verified by the experiment below. Of course, if most characters of the string database are updated, the incremental approach could be worse than the naive computation. However, such cases do not occur often.

7.2 Experiments

One way to evaluate the effectiveness of a discovery algorithm is to apply it to a real data set and see what it finds. But sometimes it may be difficult to judge the quality of the findings, without knowing a prior what the algorithm is supposed to find. Thus, to evaluate our algorithms, we generated data sets that honor several patterns modeling the weather change in Singapore. Figure 8(a) gives the set of characters encoding the weather conditions on sky and temperature. Figure 8(b) gives the patterns used to generate the data and Figure 8(c) is the graphical representation of these patterns in which a node is either a left side or a right side of a pattern. For example, the three patterns with left side ab say that if the first and second days are (Sunny, High) and (Sunny, Normal) in sequence, the third day will be (Cloudy, High), (Sunny, High),

Characters	**(Sky, Temperature)**
a	(Sunny,High)
b	(Sunny,Normal)
c	(Cloudy,High)
d	(Cloudy,Normal)
e	(Rainy,High)
f	(Rainy,Normal)

(a) The encoding of weather conditions

a $\xrightarrow{1}$ b	ab $\xrightarrow{0.4}$ f	c $\xrightarrow{0.35}$ d	cd $\xrightarrow{0.7}$ c	cf $\xrightarrow{0.8}$ b	ef $\xrightarrow{0.3}$ c
ab $\xrightarrow{0.2}$ a	bf $\xrightarrow{0.7}$ a	c $\xrightarrow{0.65}$ f	cd $\xrightarrow{0.1}$ e	e $\xrightarrow{1}$ f	fb $\xrightarrow{0.55}$ a
ab $\xrightarrow{0.4}$ c	bf $\xrightarrow{0.3}$ c	cd $\xrightarrow{0.2}$ a	cf $\xrightarrow{0.2}$ a	ef $\xrightarrow{0.7}$ a	fb $\xrightarrow{0.45}$ c

(b) Patterns used to generate the string

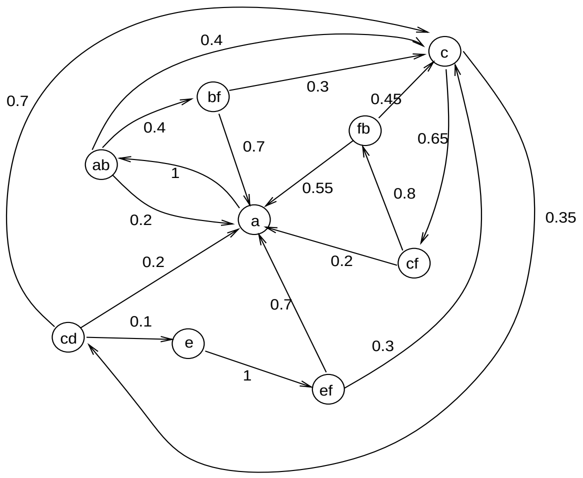


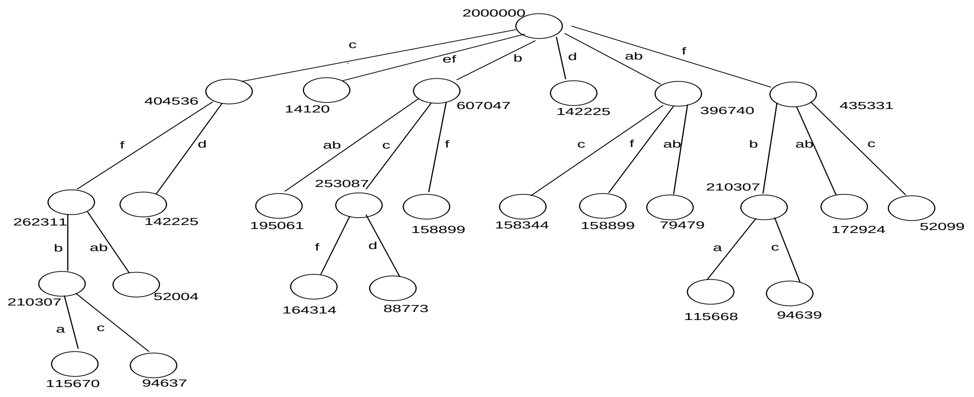
Figure 8: (c) Diagram for patterns

(Rainy, Normal) with confidence 0.4, 0.2 and 0.4, respectively. To generate a string, we apply patterns with left sides matching the last few characters of the partial string to extend the string. The confidence of a pattern determines the probability of picking up the pattern. To start with, the initial string contains the left side of a randomly chosen pattern.

Discovered patterns. We generated a string of length 2000k. The dynamic suffix tree has 3781915 nodes and depth of 30. We choose *minisup* to be 10%. Therefore, a node is "infrequent" if its $count(v)$ is less than 200k. Figure 9(a) shows the dynamic suffix tree after pruning infrequent nodes, except the first infrequent node each path (to indicate no more frequent nodes deeper in the path). Figure 9(b) shows the patterns obtained from the pruned suffix tree, as described in Section 2, of which three are among those used to generate the data and three are new. In a sense, the new patterns are those that "follow" from the original patterns. On the other hand, by reducing *minisup* to 0.21%, the algorithm actually discovered all patterns used to generate the data, as in Figure 9(c). (Much more patterns were discovered for $minisup = 0.21\%$ and we have to omit them due to the space limit.) The discovered confidences of these patterns are very close to their original confidences. What is more interesting is that the experiment also discovered the importance of these patterns which was not originally known at all. For example, the small support of $cd \rightarrow e$, $e \rightarrow f$, $ef \rightarrow a$, and $ef \rightarrow c$ suggests that these patterns are insignificant and can be ignored. Of course, it is up to the user to decide the interestingness of patterns by specifying or tuning *minisup* and *miniconf*.

Performance study. To verify the above cost analysis, we run the incremental method and the non-incremental method on same sets of data and updates. We simulated the disk in the memory and compared tree node accesses, which corresponds to disk accesses, for updating or constructing the suffix tree by the two methods. The storage used by both methods is the same and is ignored in the comparison. The data sets were generated using the above weather patterns with the size ranging from 50k to 5000k at the interval of 500k. For each data set, we considered four groups of updates, with each group containing 10 updates of the same size. The size of an update $\alpha\beta\gamma$ to $\alpha\delta\gamma$ is defined as $|\beta| + |\delta|$. The size of updates for the four groups are 10,50,100,500, respectively. An update $\alpha\beta\gamma$ to $\alpha\delta\gamma$ was generated by determining β and α in a random manner, with $|\alpha| + |\beta|$ being equal to the specified update size. We averaged the cost of performing the 10 updates in each update group. Figure 10(b) shows the cost of the incremental method for different data size. The update cost is almost not affected by the scale up of the data size beyond 500k.

To compare with the non-incremental method, Figure 10(a) shows the size of the suffix tree (the vertical axis) created by the construction algorithm for different data size. The size of the suffix tree grows linearly with the data size. The cost of the non-incremental method is at least double the size of the suffix tree because it needs to traverse the suffix tree to compute $count(v)$ for non-terminal nodes v. The comparison of Figure 10(a) and Figure 10(b) shows a clear edge of the incremental method over the non-incremental method.



(a) The suffix tree pruned by *minisup* = 10%

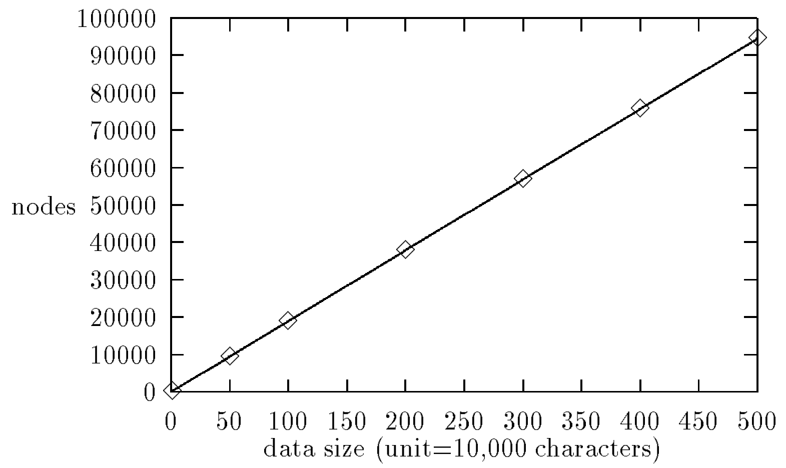
pattern	**confidence (%)**	**support (%)**	**original patterns**
a → b	100.00	19.84	yes
b → c	41.69	12.65	no
c → f	64.80	13.12	yes
c → fb	51.99	10.51	no
cf → b	80.17	10.51	yes
f → b	48.31	10.51	no

(b) Discovered patterns having *minisup* = 10%

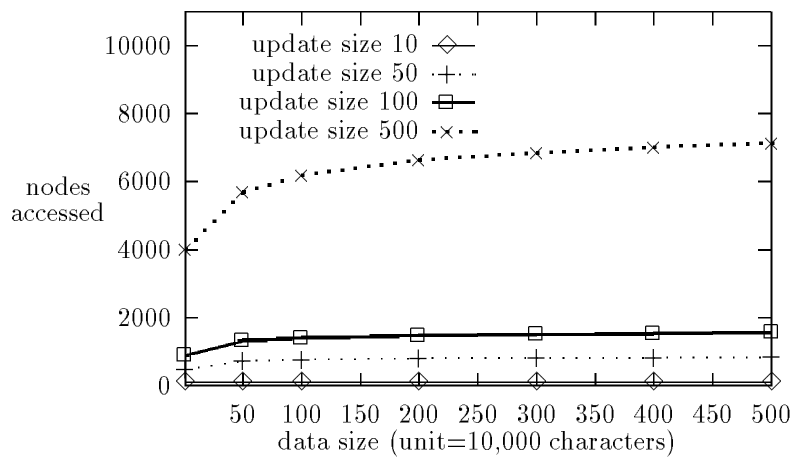
pattern	**confidence (%)**	**support (%)**
ab → a	20.04	3.97
ab → c	39.91	7.92
ab → f	40.05	7.95
bf → a	70.00	5.56
bf → c	29.99	2.38
c → d	35.16	7.11
cd → a	20.00	1.42
cd → c	69.99	4.98
cd → e	10.00	0.71
cf → a	19.83	2.60
e → f	100.00	0.71
ef → a	69.97	0.49
ef → c	30.02	0.21
fb → a	55.01	5.78
fb → c	44.99	4.73

(c) Discovered support and confidence of original patterns

Figure 9: Discovered patterns



(a) The size of the suffix tree



(b) The update cost of the dynamic suffix tree

Figure 10: Performance study

8 Conclusion

We proposed a framework of discovering sequential patterns from sequential data. Though suitable for discovering and representing sequential patterns for static strings, the suffix tree is very expensive for dynamic strings because of the sensitivity of the position to the update operation. The existing update algorithm [M76] failed to address this inefficiency. We proposed a new representation of the suffix tree for dynamic strings, called the dynamic suffix tree, in which substrings are referenced by addresses rather than positions. The address reference restricts the impact of updates to a small part of the dynamic suffix tree, making efficient update of the dynamic suffix tree possible. Based on the dynamic suffix tree, we presented an algorithm for incrementally discovering sequential patterns from large and dynamic sequential data. Experiments showed that the proposed framework finds important patterns and that the incremental method performs substantially better than the non-incremental one for large and dynamic databases.

The following areas need further investigation in the future. (a) Handling numeric values such as temperature. Discretization is a possible approach, but its effect on the discovery quality needs to be studied. (b) Discovering patterns from multi-dimensional data, with the flexibility of allowing the user to specify the dimensions for discovery. For example, in one case the user may be interested in sky patterns, in another case may be interested in temperature patterns, and in a third case in combined patterns of sky and temperature. Simply performing discovery for individual dimensions does not work, nor does performing discovery for a fixed set of dimensions. (c) Approximate patterns that allow some degree of errors or mismatches. (d) Discovery within a user-specified range of positions. (e) Discovery of periodic patterns such as "if the stock price goes up on Monday, it will drop on the next day". (f) Incremental discovery for these extensions. Solutions to these problems can generally benefit from the work in the areas of AI, combinatorial pattern matching, time series, and spatial databases, textual databases. We believe that the discovery of sequential patterns covers a major domain of knowledge discovery applications and that a viable solution to this problem is crucial to turning huge data stores into accessible and actionable knowledge.

References

[ALSS95] R. Agrawal, K.I. Lin, H.S. Sawhney, K. Shim, " Fast similarity search in the presence of noise, scaling, and translation in time-series databases", VLDB 1995, 490-501

[AS95] R. Agrawal and R. Srikant, "Mining sequential patterns", IEEE Conference on Data Engineering 1995, 3-14

[B92] R. Baeza-Yates, "Text retrieval: theory and practice", Algorithms, software, architecture: information processing 92, Vol. 1, 465-476

[BG92] R. Baeza-Yates and G.H. Gonnet, "A new approach to text searching", CACM, Vol. 35, No. 10, Oct, 1992, 74-82

[BM77] R.S. Boyer and J.S. Moore, "A fast string searching algorithm", CACM 20, 10, Oct. 1977, 762-772

[C95] A. L. Cobbs, "Fast approximate matching using suffix trees", in proc. Combinatorial pattern matching 1995, Lecture Notes in Computer Science 937, 41-54, Springer-Verlag

[F85] C. Faloutsos, "Access methods for text", ACM computing Surveys, 17, 1985, 49-74

[FSS96] U. Fayyad, G.P. Shapiro, P. Smyth, "Knowledge discovery and data mining: towards a unifying framework", KDD 1996, 82-88

[G84] A. Guttman, "R-trees: A dynamic index structure for spatial searching", ACM SIGMOD 1984, 47-57

[GY91] G.H. Gonnet and R. Baeza-Yates, Handbook of algorithms and data structures in Pascal and C, Second Edition, 1991

[H92] L.C.K. Hui, "Color set size problem with applications to string matching", in A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, Combinatorial Patterns Matching, Lecture Notes in Computer Science 644, 230-243, Springer-Verlag, 1992

[KMP77] D.E. Knuth, J.H. Morris, V.R. Pratt, "Fast pattern matching in strings", SIAM J. Comput. 6, June 1977, 323-350

[LV89] G.M. Landau and U. Vishkin, "Fast parallel and serial approximate string matching", Journal of Algorithms, Vol. 10, No. 2, 157-169, 1989

[M76] E.M. McCreight, "A space-economical suffix tree construction algorithm", JACM, Vol. 23, No. 2, April 1976, 262-272

[MD85] T.G. Dietterich and R.S. Michalski, "Discovering patterns in strings of events", Artificial Intelligence, Vol. 25, 187-232, 1985

[S94] G.A. Stephen, "String searching algorithms", Lectures Notes Series on Computing, Vol. 3, 1994, World Scientific

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A dynamic index for multi-dimensional objects", VLDB 1987, 507-518

[TMS94] A. Tomasic, H. Garcia-Molina, and K. Shoens, "Incremental updates of inverted lists for text document retrievals", ACM SIGMOD, 1994,

[U92] E. Ukkonen, "Constructing suffix-trees on-line in linear time", Algorithms, Software, Architecture: Information Processing 92, Vol. 1, 484-492, Elsevier, Amsterdam

[U93] E. Ukkonen, "Approximate matching over suffix trees", in proc. Combinatorial pattern matching, Vol. 4, 228-242, Springer-Verlag, June 1993

[W73] P. Weiner, "Linear pattern matching algorithms", Conf. Record, IEEE 14th Annual Symposium on Switching and Automata Theory 1973, 1-11

[W*94] J.T. L. Wang, G.W. Chirn, T.G. Marr, B. Shapiro, D. Shasha, and K. Zhang, "Combinatorial pattern discovery for scientific data: some preliminary results", ACM SIGMOD 1994, 115-125

[WM92] S. Wu and U. Manber, "Fast text searching allowing errors", CACM, Vol. 35, No. 10, Oct, 1992, 83-91

[ZMD93] J. Zobel, A. Moffat, and R. Sacks-Davis, "Searching large lexicons for partially specified terms using compressed inverted files", VLDB 1993, 290-301