Incremental Discovery of Sequential Patterns

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

Abstract

In this paper, incremental discovery of sequential patterns from a large sequence database is studied. A sequential pattern has the form $\alpha \rightarrow \beta$, where α and β are subsequences in the database, which says that events α will be followed by events β with some minimum support and confidence. We consider the scenario that sequential patterns have previously been discovered and materialized and an update is subsequently made to the database. Rediscovering all patterns by scanning the whole database is not acceptable in a dynamic and large database environment. We propose an incremental discovery algorithm that produces the updated patterns by scanning only the affected part of the database and data structures. In addition, the algorithm handles the dynamism of the minimum support and confidence without recomputation, allowing the user to tune these parameters and focus on most interesting patterns at little overhead. Experiments and comparisons were conducted to test the effectiveness of the proposed algorithm.

1 Introduction

In the daily and scientific life, sequential data are available and used everywhere. Examples are weather data, satellite data streams, stock prices, experiment runs, DNA sequences, histories of medical records, etc. Reoccurrences of activities and phenomena are captured in the form of repeated patterns in sequential data. Discovering interesting patterns can benefit the user or scientist by predicting coming activities, interpreting certain phenomena, extracting outstanding similarities and differences for close attention.

Discovering patterns in sequence data has been an area of active research in AI (see, for example, [MD85]), where the database is static and usually small. This topic was recently considered from the database perspective where the data is stored on the secondary storage, e.g., [AS95, MTV95, W*94]. To our knowledge, the following incremental discovery of patterns from a sequential database has not been addressed, which is important for large and dynamic databases where rediscovering all patterns from scratch is too expensive. Assume that patterns have previously been discovered and materialized and an update is now made to the database. We like the patterns to be updated by examining only the affected part of the database and data structures. In addition, two practical requirements have to be addressed. The user may like to zoom in the detail of patterns for explanation of patterns, therefore, all positions in which patterns occur need to be maintained. The user may also like to fine-tune the minimum support and confidence a few times in search for interesting patterns. It is important to reflect the new support and confidence without much recomputation.

Three models of incremental discovery of sequential patterns are considered in this paper. (a) The database is a single but long sequence and a pattern is a subsequence whose number of occurrences exceeds some threshold. An update is either insertion or deletion of a subsequence at either end of the sequence. (b) The database is a collection of sequences and a pattern is a subsequence whose total number of occurrences in all sequences exceeds some threshold. An update is insertion or deletion of a whole sequence. (c) This is the same as (b), except that only the number of sequences in which a pattern occurs is counted. We will present our solutions to these models of the incremental discovery problem.

We use an index structure that is a modification of the *compact suffix tree* (suffix tree for short) [We73]. The suffix tree has previously been used to search for subsequences similar to a given subsequence, especially in the area of string matching and text editing. New issues have to be addressed in applying the suffix tree to incremental discovery of sequential patterns. First, we have to figure out what patterns to try, which is different from search for subsequences matching a given pattern. Second, updating the highly structured suffix tree in response to the change of sequences requires insightful analysis of the suffix tree. Third, the statistical information about patterns have to be maintained and updated efficiently. Fourth, to allow the user to focus on the most interesting patterns, the flexibility of tuning the minimum support and confidence of patterns without additional work is required. In the rest of the paper, we will adopt an informal description of our approach, mostly by examples, to allow better understanding of the essence.

2 Suffix Trees and Sequential Patterns

A sequence S is a list of records ordered by position number starting with 1 and delimited by the special symbol \$ that occurs only at the right end of the list. For clarity, records are represented by integers. In real applications, a record is a description of events of interest. For example, record $(sunny, low, dry)$ denotes a sunny, dry, and cold day. Additional information may be associated with a record and can be retrieved by the position number of the record. For example, the above record may be associated with the detailed weather description of the day.

2.1 Suffix Trees

The suffix tree for a sequence S was first introduced in [We73] primarily for string matching. For more applications of suffix trees, the reader can refer to the book [S94]. Since no suffix of S is a prefix of a different suffix of S due to \$, S can be mapped to a suffix tree T whose paths are the suffixes of S, and whose terminal nodes correspond uniquely to positions within S. The suffix tree T for S satisfies the following properties.

T1 An arc of T may represent any nonempty subsequence of S.

T2 Each nonterminal node of T, except the root, must have at least two offspring arcs.

T3 The subsequences represented by sibling arcs of T must begin with different records.

T is a multiway Patrica tree and thus contains at most n nonterminal nodes [Mc76, S94], where n is the number of records in S. A linear time (in the number of node access) and space construction of the suffix tree for a sequence were presented in [Mc76, We73]. The following example illustrates the idea.

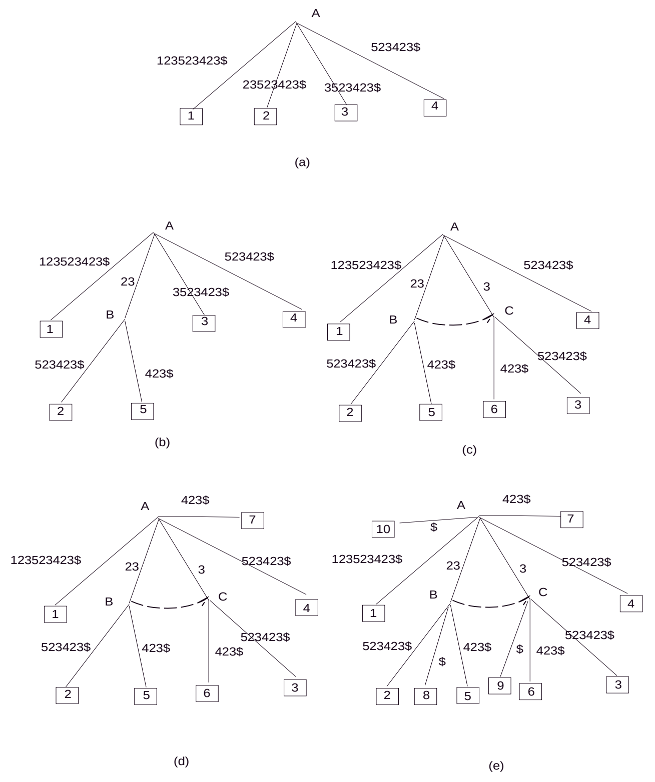


Figure 1: Construction of suffix trees

Example 2.1 *Consider the sequence* $S = 123523423\$$*, where each single digital denotes a record. Suffixes of* S *are inserted into the suffix tree* T *one at a time, starting from the longest suffix, as shown in Figure 1. (a) shows the suffix tree after the first four suffixes are inserted. Since none of these suffixes share a prefix, they all go to different branches. A box represents a terminal node and contains the starting position of the suffix represented by the terminal node. (b) shows the suffix tree after the suffix* 23423\$ *is inserted.* 23423\$ *shares the prefix* 23 *with* 23523423\$*, so the arc for* 23523423\$ *is split into two, one for* 23 *and one for* 523423\$*. (c) is the tree after suffix* 3423\$ *is inserted, in which the old arc for* 3523423\$ *is split into one arc for* 3 *and one for* 523423\$*. At the moment, let us ignore the dashed line. (d) is the tree after suffix* 423\$ *is inserted. In (e), suffixes* 23\$*,* 3\$*, and* \$ *are inserted. The tree in (e) is the suffix tree for* S*. The number in a terminal node denotes the starting position of the subsequence from the root to the terminal node.*

(In the implementation, the subsequence associated with an arc can be represented by the pair of starting position and ending position of the subsequence.) A straightforward way of inserting a suffix into the suffix tree is entering the tree from the root and searching along a path until the next record in the path and the next record in the suffix disagree. Unfortunately, this construction requires $O(n^2)$ in time. A linear time construction, first proposed in [We73] and later modified in [Mc76], is to start the search at the lowest possible level, by maintaining auxiliary links called *suffix links* between nonterminal nodes of the tree. For two nonterminal nodes u and v, a suffix link from u to v is created if the paths from the root to u and v have the form $x\alpha$ and α, respectively, where x is a single record and α is a subsequence. Let suf_i denote the suffix of S beginning at position i, with suf_1 being the longest, and let T_i denote the suffix tree after $suf_1, \ldots, suf_i$ are inserted. Note that $suf_i = xsuf_{i+1}$,where x is the record at position i. α and β represent any subsequences in S. Below is an informal description of the linear time construction in [Mc76].

Initial construction. The suffix tree for S is constructed in the order $T_1, T_2, \ldots$. Initially, T_0 contains only the root and its suffix link points to the root itself. Suppose that T_i was constructed. On the path for suf_i, let u_i be the lowest nonterminal node with a suffix link and correspond to path $x\alpha$, where x is a subsequence of at most one record, and let v_i be the lowest nonterminal node and correspond to path $x\alpha\beta$. Suppose that every nonterminal node in T_i, except v_i, has 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 of u_i. From the definition, the node pointed by the suffix link corresponds to the path α, which is a prefix of suf_{i+1}. Starting at this node (rather than starting from the root) we move down the tree along a sequence of arcs that spells out β. If β does not end exactly at a node, the last arc in the sequence will be split and a new node d is inserted at the end of β. Make the suffix link of v_i point to d. Then the algorithm continues search from d deeper into the tree. The algorithm "falls out of the tree" when the longest repeating prefix of suf_{i+1} (in S) is consumed in the search. If this prefix does not end exactly at a node, the arc containing its end is split and a new nonterminal node is inserted. Finally, a new terminal node is inserted for the rest of suf_{i+1}. This new tree is T_{i+1} with suf_{i+1} inserted. At most one nonterminal node and one terminal node are actually created during the transformation from T_i to T_{i+1} [Mc76].

For example, for suffix trees in Figure 1(a and b), all suffix links of nonterminal nodes point to the root (thus, no short-cuts provided). In Figure 1(c), after the suffix 3423\$ is inserted, the suffix link drawn by the dashed line is created. In Figure 1(e), after suffix 23\$ is inserted, the dashed suffix line provides a short-cut into a position to insert suffix 3\$. For longer sequences, the short-cut provided by suffix links will avoid long traversing from the root.

2.2 Sequential Patterns

The *support* of a subsequence α with respect to S is the number of positions in S at which α occurs. For subsequence $\alpha\beta$, $\alpha \rightarrow \beta$ denotes the *(sequential) pattern* that α is followed by β. The *support* of pattern $\alpha \rightarrow \beta$ is the support of $\alpha\beta$. The *confidence* of pattern $\alpha \rightarrow \beta$ is the ratio of the support of $\alpha\beta$ over the support of α. Given a minimum support s and a minimum confidence c, the problem of discovering sequential patterns is to find all patterns $\alpha \rightarrow \beta$ that have a support no less than s and a confidence no less than c.

Consider the suffix tree T of sequence S. Any nonempty subsequence α in S can be spelled out by following a unique path from the root of T. The *extended locus* of α, denoted $locus(\alpha)$, is the first node in T encountered after α is spelled out. Let $v.support$ denote the number of terminal nodes in the subtree rooted at a nonterminal node v. For the initial suffix tree T, $v.support$ can be obtained by a postorder traverse of T. With respect to the user-specified minimum support s and minimum confidence c, $\alpha \rightarrow \beta$ is a *pattern* if

$$locus(\alpha\beta).support \geq s$$

and

$$locus(\alpha\beta).support/locus(\alpha).support \geq c.$$

Hence, the portion of T above all nonterminal nodes v such that $v.support \geq s$ is all we need to produce all patterns. We call this portion the *pattern tree* (with respect to the specified minimum support s). In the pattern tree, for each path starting at the root that spells out subsequence $\alpha\beta$, where α and β are nonempty, $\alpha \rightarrow \beta$ is a pattern if and only if $locus(\alpha\beta).support/locus(\alpha).support \geq c$. For this reason, we use the pattern tree to represent patterns. Since the whole suffix tree is maintained over time, discovering sequential patterns with respect to any minimum support without further computation is possible. This is highly desirable because in most cases the user tends to try a few minimum supports before being satisfied with the result.

3 A Single Sequence

In this section, the database consists of a single and long sequence S. We assume that the suffix tree T for S is materialized. Therefore, pattern trees with respect to any minimum support and confidence are available for retrieval. We consider how to update T when sequence $S = \alpha\beta\gamma$ is updated to $\alpha\delta\gamma$.

The update strategy. The following strategy was suggested in [Mc76] to update the suffix tree T for S. Let α^* be the longest suffix of α which occurs in at least two different places in $\alpha\beta\gamma$. With respect to the update $\alpha\beta\gamma \rightarrow \alpha\delta\gamma$, we define as *$\beta$-splitters* those subsequences (or their paths) of the form $\epsilon\gamma$, where ϵ is a nonempty suffix of $\alpha^*\beta$. Equivalently, β-splitters are paths in T which properly contain the suffix γ, but whose terminal arcs do not properly contain $\beta\gamma$, because a β-splitter does not go to a terminal arc before running out of its suffix of α^* (if any). Two cases of a β-splitter will be considered. In the first case, some characters of β are shared by another path, thus, the path of the β-splitter is affected by the update. In the second case, the terminal arc of a β-splitter contains exactly $\beta\gamma$, and replacing β by δ may cause the terminal arc to be split and merged with its sibling arcs to maintain T3. β-splitters are the only paths whose structures might be affected by the change from β to δ. All other paths in T reflect the change either because they are too short

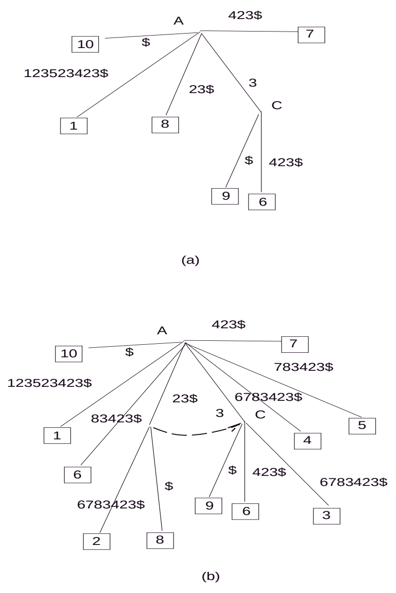


Figure 2: Replacing 52 by 678

to contain any character of β, or because they are so long that β is buried in a terminal arc and the change from β to δ cannot affect the structure of the path. The strategy of replacing β with δ is to delete all β-splitters from T and insert into T all *δ-splitters* of the form $\omega\gamma$, where ω is a nonempty suffix of $\alpha^*\delta$. Let us illustrate these concepts by an example.

Example 3.1 *Suppose we replace $\beta = 52$ with $\delta = 678$ in the sequence $S = 1235234 23\$$, given the suffix tree in Figure 1(e) for S. We have $\alpha = 123$ and $\gamma = 3423\$$. The longest repeating suffix of α is $\alpha^* = 23$, and so $\alpha^*\beta = 2352$. There are four β-splitters: $2\gamma, 52\gamma, 352\gamma, 2352\gamma$, starting at positions 5,4,3,2, respectively. The terminal arcs for these positions do not properly contain $\beta\gamma$, therefore, the paths for these β-splitters are affected by the update and should be deleted. Deleting a path corresponds to deleting the terminal arc. After the terminal arc for position 2 is deleted, node B has only one offspring arc left, which violates T2. In this case the two arcs on the path are merged into one arc. The tree after deleting all β-splitters is shown in Figure 2(a).*

Then we insert δ-splitters into the tree in Figure 2(a). Since $\alpha^*\delta = 23678$, there are five δ-splitters: 8γ, 78γ, 678γ, 3678γ, 23678γ. The insertion starts with the longest δ-splitters, i.e., 23678γ, as if suf_1 has just been inserted in the construction algorithm. The tree after inserting these δ-splitters is given in Figure 2(b).

However, some important problems remain to be solved before applying this strategy to our incremental discovery problem. We have to efficiently maintain the position information of records, which is difficult for general updates. Therefore, we consider only updates that are made to either end of S, as usually the case for sequential data where old records are deleted at the left end and new records are added at the right end. Also, we have to solve the problem that the starting and ending positions associated with an arc may not exist due to deletion of subsequences. Finally, after each insertion or deletion we have to update *v.support* for affected nodes v. An efficient counting method should access only affected nodes once. All these issues are left unaddressed in [Mc76] but are important for the discovery problem.

3.1 Insertion

Insertion at the right end. In this case, the general update $\alpha\beta\gamma \rightarrow \alpha\delta\gamma$ becomes $\alpha\$ \rightarrow \alpha\delta\$$, with $\beta = \emptyset$ and $\gamma = \$$. β-splitters are of form $\epsilon\$$, where ϵ is a nonempty suffix of α^*. The algorithm maintains two pointers that point to the terminal nodes for the longest and shortest suffixes of S. Thus, we can find the shortest β-splitter $\alpha^1\$$ directly, where α^i is the suffix of α that has length i. To delete all β-splitters efficiently, we modify the construction algorithm to chain up all terminal nodes of T in the ascending position order. This can be easily done because suffixes are inserted into T in the same order. With this modification, we delete all β-splitters by following the terminal chain towards lower positions (i.e., longer β-splitters) until a path that is not a β-splitter is encountered. Note that a path is a β-splitter if and only if its terminal arc does not properly contain $\beta\gamma$, which can be easily tested by reading the starting position of the terminal arc. Note that deleting a path can be done by accessing only a fixed number of nodes. The detail can be found in [Mc76]. Next, we insert all δ-splitters $\omega\$$, where ω is a nonempty suffix of $\alpha^*\delta$, as in the construction of Section 2. Since terminal arcs store only the starting position of the rest of the suffixes, there is no need to insert δ into terminal arcs of the tree.

The frontal set update. The insertion and deletion of splitters may change the support *v.support* associated with nonterminal nodes v. Refreshing the support by the postorder traversing of T will access all nodes in the tree and defeat the purpose of incremental discovery. We propose a counting method to update the support by accessing only affected nodes. Let $v_1, \ldots, v_k$ be the terminal nodes of the paths newly inserted. The support of all nonterminal nodes on these paths are affected. Initially, let F contain all nodes $v_1, \ldots, v_k$ and each v_i is associated with an increment Δ_i that is set to 1 at the beginning. In each iteration, we traverse up one arc from the deepest nodes v_i in F and increase the support of the nodes u_i reached from v_i by the amount of Δ_i. Then we update F by replacing v_i with u_i. If $u_{i_1}, \ldots, u_{i_k}$ are identical (that is, $v_{i_1}, \ldots, v_{i_k}$ are siblings), the sum $\Delta_{i_1} + \ldots + \Delta_{i_k}$ will be associated with this node. This process terminates when the support of the root is updated. In this counting algorithm, each node on the inserted paths is accessed only once.

Insertion at the left end. This is the case that the general update $\alpha\beta\gamma \rightarrow \alpha\delta\gamma$ becomes $\gamma \rightarrow \delta\gamma$, by letting $\alpha = \beta = \emptyset$. There is no β-splitter and so deletion of β-splitters is omitted. We apply the normal construction algorithm to insert all δ-splitters of the form $\omega\gamma$, where ω is a nonempty suffix of δ. Upon completion of the insertion, we apply the above frontal set update algorithm to update the support of nodes. Unlike insertion at the right end, the insertion of δ at the left end will increase the position counts of all records in S by the length of δ. To keep track of this change, an offset *Count* is maintained that will be increased by the length of δ whenever δ is inserted at the left end. The correct position of a record in S is its original position plus the offset *Count*.

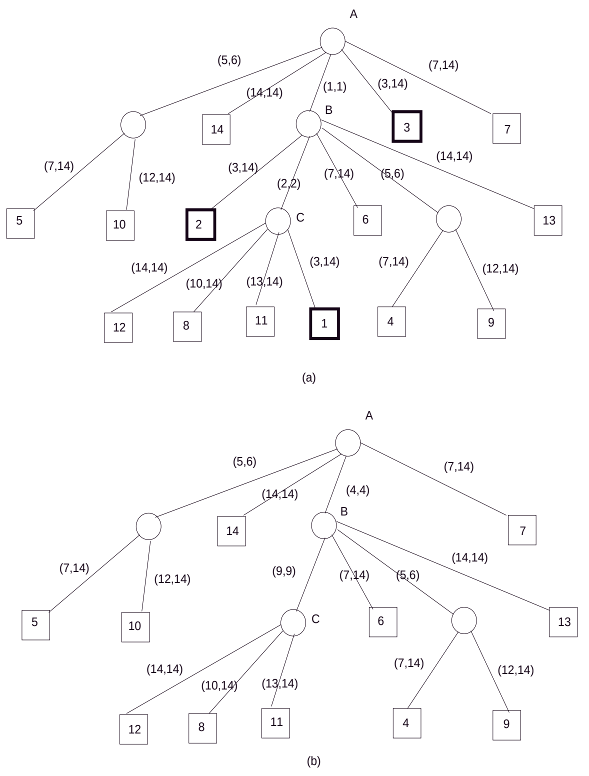
3.2 Deletion

Deletion at the right end. The general update $\alpha\beta\gamma \rightarrow \alpha\delta\gamma$ specializes to $\alpha\beta\$ \rightarrow \alpha\$$, by letting $\gamma = \$$ and $\delta = \emptyset$. β-splitters are $\epsilon\$$, where ϵ is a nonempty suffix of $\alpha^*\beta$. We delete all β-splitters and insert all δ-splitters $\omega\$$, where ω is a nonempty suffix of α^*. Finally, we apply the frontal set update algorithm to update the support of affected nonterminal nodes. For deletion, in the frontal set update algorithm we also check if the remaining arcs on a deleted path refer to deleted positions and change such references to existing positions. See more details and an example below.

Deletion at the left end. The general update $\alpha\beta\gamma \rightarrow \alpha\delta\gamma$ specializes to $\beta\gamma \rightarrow \gamma$, by letting $\alpha = \delta = \emptyset$. We delete all β-splitters $\epsilon\gamma$, where ϵ is a nonempty suffix of β. There is no δ-splitter to insert because $\alpha = \delta = \emptyset$. The frontal set update algorithm is applied to update the support of affected nonterminal nodes. The deletion of β at the left end will decrease the position counts of all records in S by the length of β, so we decrease *Count* by the length of β.

In the implementation, we store the starting and ending positions (fr, to) of the left-most occurrence of the subsequence on an arc. If records addressed by these positions are deleted, (fr, to) will have to be changed to another occurrence of the same subsequence. In the following example, (fr, to) is changed according to the next left-most existing occurrence of a subsequence.

Example 3.2 *Suppose that we delete the three left-most records 335 in the sequence* $S = 3353432334333\$$ *whose suffix tree is given in Figure 3(a). This is done by deleting the paths corresponding to leaves marked 1, 2, and 3. Before the deletion, the pairs (1,1), (2,2), and (3,14) along the path ending at the leaf containing position 1 refer to the left-most occurrences of these subsequences. Since the first three records are deleted, these pairs no longer point to the intended records.*



1	2	3	4	5	6	7	8	9	10	11	12	13	14
3	3	5	3	4	3	2	3	3	4	3	3	3	$

Figure 3: Update of (fr, to) for deletion

The same problem occurs for the paths corresponding to leaves marked 2 and 3. All these pairs are replaced by some surviving occurrences of the same subsequence. For example, the smallest surviving fr *under node C is 10 and we can use (9,9) to replace the non-existing (2,2) on the arc (B,C). Under node B the smallest surviving* fr *is 5 and so we can replace the non-existing (1,1) with (4,4) on the arc (A,B). Figure 3(b) is the tree after the deletion.*

The update of (fr, to) pairs can be done in the same scan as the update of support for affected nodes in the frontal set update algorithm. In particular, at a nonterminal node v with parent u, the smallest existing (i.e., excluding deleted paths) fr under v is used to compute the to reference for the arc (u, v), that is, $fr + 1$. Note that the actual starting and ending positions (fr, to) on an arc are computed by $(fr + Count, to + Count)$, where *Count* is the accumulated offset of positions due to insertion and deletion at either ends.

3.3 Time complexity

The time complexity is measured by the number of node access. The time complexity of an update consists of

(a) the complexity of deleting all β-splitters, (b) the complexity of inserting all δ-splitters, (c) the complexity of updating support and (fr, to) pairs on affected paths. For each insertion or deletion of a subsequence discussed above, the number of paths inserted or deleted is bounded by the length of β-splitters $\alpha^*\beta$ plus the length of δ-splitters $\alpha^*\delta$. Deletion of a β-splitter is done by at most one terminal node access and two nonterminal node accesses. From [Mc76], the average number of nonterminal node access to insert a path is no more than 2, and the number of terminal node access is 1. Thus, deletion and insertion of splitters can be done in a time linear in the length of $\alpha^*\beta$ and $\alpha^*\delta$. The number of node access for updating support and (fr, to) pairs incurred by the frontal set update algorithm is equal to the number of distinct nodes on the paths corresponding to the initial frontal set F. If some of these paths merge, each node on the merged part only needs to be accessed once. All three components of complexity are contributed only by the affected portion of the tree. For large sequences and small updates, which is our basic assumption for the problem of incremental discovery, only a small portion of the tree will be affected and therefore the cost of update is low.

4 Support by Occurrences

We now consider the incremental discovery problem where the database consists of multiple sequences and the *support* of a subsequence α is defined as the sum of the support of α in all sequences. Therefore, a sequence containing many occurrences of a subsequence will support the subsequence more strongly than a sequence containing few occurrences of the subsequence. Consider a sequence database $\mathcal{D} = \{S_1, \ldots, S_k\}$ of k sequences. An update is either insertion of a new sequence or deletion of an old sequence. We reduce the incremental discovery problem to that for a single sequence database by replacing $\mathcal{D}$ with sequence $S = S_1\$_1 \ldots S_k\$_k$, where $\$_i$ is a unique symbol that occurs no elsewhere in S. Since all patterns must be repeating subsequences, no pattern of S will contain $\$_i$. To locate all paths of a sequence, a B^+-tree on pairs $(id, head)$ is maintained with id being the search key, where id is a sequence identifier and $head$ points to the first terminal node of the sequence id. Then all paths of sequence id are found by entering the terminal node chain pointed by $head$ and moving to higher positions of the chain until a terminal arc delimited by a different delimiter $\$_i$ is encountered. See Figure 4.

A new sequence is inserted by inserting all its suffixes into the suffix tree for S with all new terminal nodes chained up. Then entry $(id, head)$ is inserted into the B+-tree, where id is the identifier of the inserted sequence and $head$ is the address of the head of the new terminal chain. To delete an existing sequence

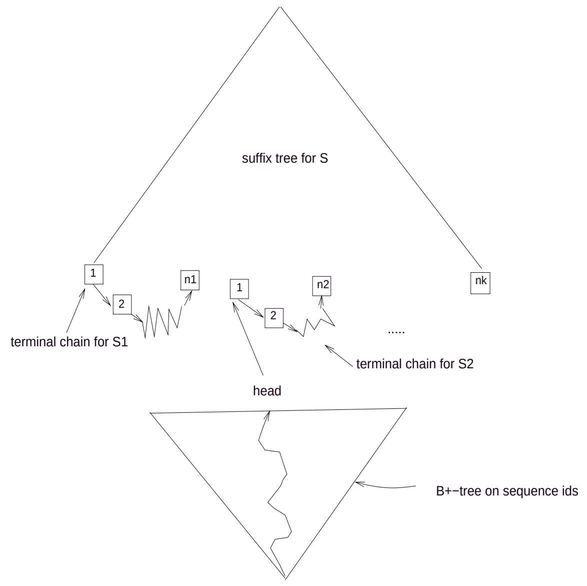


Figure 4: Locating all suffixes of a sequence

with identifier id, the head of the terminal chain of the sequence is found by searching the B+-tree using id. Once the head is found, the deletion in the suffix tree proceeds exactly as the deletion of a subsequence at the left end of a sequence in Section 3, except that the subsequence is the whole sequence identified by id. By joining all sequences in $\mathcal{D}$ into a single sequence S, common prefixes of suffixes from different sequences can share the same path in the suffix tree for S (thus saving storage) and we can find out the support of a subsequence by following only one path for the subsequence (thus saving time).

However, since insertion and deletion is performed for a whole sequence and since positions within sequences are maintained independently, the problem of altering positions and (fr, to) pairs in Section 3 does not exist any more. Also, unlike in Section 3 where an insertion or deletion of a subsequence requires deleting affected old splitters followed by inserting new splitters in general, insertion or deletion of a whole sequence needs only to insert new suffixes or deleting old suffixes, but not both. The update of support remains the same as in Section 3. With these minor changes, the algorithms in Section 3 also provide a solution to the incremental discovery problem in question.

5 Support by Sequences

This model is the same as the model in Section 4, except that the *support* of a subsequence α is defined as the number of sequences in which α occurs. As in Section 4, the sequence database $\mathcal{D} = \{S_1, \ldots, S_k\}$ can be replaced by a single sequence $S = S_1\$_1 \ldots S_k\$_k$. However, the algorithm of counting the support of nonterminal nodes needs to be modified. For a nonterminal node v, $v.support$ is now equal to the number of distinct sequence ids appearing in the subtree rooted at v. Since only one sequence is inserted or deleted each time, the frontal set update algorithm is modified such that if u_i is reached from children $v_{i_1}, \ldots, v_{i_q}$, $q \geq 1$, $u_i.support$ is always increased by 1 for insertion, or decreased by one for deletion.

6 Experiments

In one experiment, we designed a set of weather patterns to generate a sequence of synthetic weather data, and then we applied the suffix tree algorithm to the data to construct the pattern tree. All original patterns were discovered, though some additional patterns were also found. Most of these additional patterns have the form $\alpha \rightarrow \delta$ where δ is a prefix of β for an original pattern $\alpha \rightarrow \beta$. This verified the correctness of the suffix tree based discovery method. In another experiment, we compared the proposed incremental discovery algorithms with their nonincremental counterparts in terms of both the elapsed time and the number of node accesses. The database size ranges from 50k to 1000k at the interval of 50k. For each size, we performed the same set of 10 updates of length ranging from 10 to 500 records, and we averaged the cost collected. As expected, we found that the incremental discovery algorithms for the three models have similar performance and substantially outperform their nonincremental counterparts. The detailed experimental results will be reported in the full paper.

Acknowledgements. We would like to thank the referees for valuable comments and suggestions.

References

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the IEEE Data Engineering, 1995*, pages 3-14

[Mc76] E.M. McCreight. A space-economical suffix tree construction algorithm. *JACM, Vol. 23, No. 2*, April 1976, page 262-272

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

[MTV95] H. Mannila, H. Toivonen, A.I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of KDD 1995*, page 210-215

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

[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. *In Proceedings of ACM SIGMOD 1994*, page 115-125

[We73] P. Weiner. Linear pattern matching algorithms. *In Proceedings of Conf. Record, the IEEE 14th Annual Symposium on Switching and Automata Theory, 1973*, page 1-11