CHAPTER 10 Sequence Mining

Many real-world applications such as bioinformatics, Web mining, and text mining have to deal with sequential and temporal data. Sequence mining helps discover patterns across time or positions in a given dataset. In this chapter we consider methods to mine frequent sequences, which allow gaps between elements, as well as methods to mine frequent substrings, which do not allow gaps between consecutive elements.

10.1 FREQUENT SEQUENCES

Let Σ denote an *alphabet*, defined as a finite set of characters or symbols, and let $|\Sigma|$ denote its cardinality. A *sequence* or a *string* is defined as an ordered list of symbols, and is written as $\mathbf{s} = s_1 s_2 \dots s_k$, where $s_i \in \Sigma$ is a symbol at position i, also denoted as $\mathbf{s}[i]$. Here $|\mathbf{s}| = k$ denotes the *length* of the sequence. A sequence with length k is also called a *k-sequence*. We use the notation $\mathbf{s}[i:j] = s_i s_{i+1} \cdots s_{j-1} s_j$ to denote the *substring* or sequence of consecutive symbols in positions i through j, where $j > i$. Define the *prefix* of a sequence $\mathbf{s}$ as any substring of the form $\mathbf{s}[1:i] = s_1 s_2 \dots s_i$, with $0 \le i \le n$. Also, define the *suffix* of $\mathbf{s}$ as any substring of the form $\mathbf{s}[i:n] = s_i s_{i+1} \dots s_n$, with $1 \le i \le n+1$. Note that $\mathbf{s}[1:0]$ is the empty prefix, and $\mathbf{s}[n+1:n]$ is the empty suffix. Let $\Sigma^\star$ be the set of all possible sequences that can be constructed using the symbols in Σ, including the empty sequence $\emptyset$ (which has length zero).

Let $\mathbf{s} = s_1 s_2 \dots s_n$ and $\mathbf{r} = r_1 r_2 \dots r_m$ be two sequences over Σ. We say that $\mathbf{r}$ is a *subsequence* of $\mathbf{s}$ denoted $\mathbf{r} \subseteq \mathbf{s}$, if there exists a one-to-one mapping $\phi : [1,m] \to [1,n]$, such that $\mathbf{r}[i] = \mathbf{s}[\phi(i)]$ and for any two positions i, j in $\mathbf{r}$, $i < j \implies \phi(i) < \phi(j)$. In other words, each position in $\mathbf{r}$ is mapped to a different position in $\mathbf{s}$, and the order of symbols is preserved, even though there may be intervening gaps between consecutive elements of $\mathbf{r}$ in the mapping. If $\mathbf{r} \subseteq \mathbf{s}$, we also say that $\mathbf{s}$ *contains* $\mathbf{r}$. The sequence $\mathbf{r}$ is called a *consecutive subsequence* or substring of $\mathbf{s}$ provided $r_1 r_2 \dots r_m = s_j s_{j+1} \dots s_{j+m-1}$, i.e., $\mathbf{r}[1:m] = \mathbf{s}[j : j+m-1]$, with $1 \le j \le n-m+1$. For substrings we do not allow any gaps between the elements of $\mathbf{r}$ in the mapping.

Example 10.1. Let $\Sigma = \{A, C, G, T\}$, and let $\mathbf{s} = ACTGAACG$. Then $\mathbf{r}_1 = CGAAG$ is a subsequence of $\mathbf{s}$, and $\mathbf{r}_2 = CTGA$ is a substring of $\mathbf{s}$. The sequence $\mathbf{r}_3 = ACT$ is a prefix of $\mathbf{s}$, and so is $\mathbf{r}_4 = ACTGA$, whereas $\mathbf{r}_5 = GAACG$ is one of the suffixes of $\mathbf{s}$.

Given a database $\mathbf{D} = \{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_N\}$ of N sequences, and given some sequence $\mathbf{r}$, the *support* of $\mathbf{r}$ in the database $\mathbf{D}$ is defined as the total number of sequences in $\mathbf{D}$ that contain $\mathbf{r}$

$$sup(\mathbf{r}) = \left|\left\{\mathbf{s}_i \in \mathbf{D} | \mathbf{r} \subseteq \mathbf{s}_i\right\}\right|$$

The *relative support* of $\mathbf{r}$ is the fraction of sequences that contain $\mathbf{r}$

$$rsup(\mathbf{r}) = sup(\mathbf{r})/N$$

Given a user-specified *minsup* threshold, we say that a sequence $\mathbf{r}$ is *frequent* in database $\mathbf{D}$ if $sup(\mathbf{r}) \geq minsup$. A frequent sequence is *maximal* if it is not a subsequence of any other frequent sequence, and a frequent sequence is *closed* if it is not a subsequence of any other frequent sequence with the same support.

10.2 MINING FREQUENT SEQUENCES

For sequence mining the order of the symbols matters, and thus we have to consider all possible *permutations* of the symbols as the possible frequent candidates. Contrast this with itemset mining, where we had only to consider *combinations* of the items. The sequence search space can be organized in a prefix search tree. The root of the tree, at level 0, contains the empty sequence, with each symbol $x \in \Sigma$ as one of its children. As such, a node labeled with the sequence $\mathbf{s} = s_1 s_2 \ldots s_k$ at level k has children of the form $\mathbf{s}' = s_1 s_2 \ldots s_k s_{k+1}$ at level $k+1$. In other words, $\mathbf{s}$ is a prefix of each child $\mathbf{s}'$, which is also called an *extension* of $\mathbf{s}$.

Example 10.2. Let $\Sigma = \{A, C, G, T\}$ and let the sequence database $\mathbf{D}$ consist of the three sequences shown in Table 10.1. The sequence search space organized as a prefix search tree is illustrated in Figure 10.1. The support of each sequence is shown within brackets. For example, the node labeled A has three extensions AA, AG, and AT, out of which AT is infrequent if $minsup = 3$.

The subsequence search space is conceptually infinite because it comprises all sequences in Σ^*, that is, all sequences of length zero or more that can be created using symbols in Σ. In practice, the database $\mathbf{D}$ consists of bounded length sequences. Let l denote the length of the longest sequence in the database, then, in the worst case, we will have to consider all candidate sequences of length up to l, which gives the following

Table 10.1. Example sequence database

Id	Sequence
$\mathbf{s}_1$	$CAGAAGT$
$\mathbf{s}_2$	$TGACAG$
$\mathbf{s}_3$	$GAAGT$

ALGORITHM 10.1. Algorithm GSP

```
GSP (D, $\Sigma$, minsup):
1  $\mathcal{F} \leftarrow \emptyset$
2  $\mathcal{C}^{(1)} \leftarrow \{\emptyset\}$ // Initial prefix tree with single symbols
3  foreach $s \in \Sigma$ do Add $s$ as child of $\emptyset$ in $\mathcal{C}^{(1)}$ with $sup(s) \leftarrow 0$
4  $k \leftarrow 1$ // $k$ denotes the level
5  while $\mathcal{C}^{(k)} \neq \emptyset$ do
6      ComputeSupport ($\mathcal{C}^{(k)}, \mathbf{D}$)
7      foreach leaf $\mathbf{s} \in \mathcal{C}^{(k)}$ do
8          if $sup(\mathbf{r}) \geq minsup$ then $\mathcal{F} \leftarrow \mathcal{F} \cup \{(\mathbf{r}, sup(\mathbf{r}))\}$
9          else remove $\mathbf{s}$ from $\mathcal{C}^{(k)}$
10     $\mathcal{C}^{(k+1)} \leftarrow$ ExtendPrefixTree ($\mathcal{C}^{(k)}$)
11     $k \leftarrow k+1$
12 return $\mathcal{F}^{(k)}$

ComputeSupport ($\mathcal{C}^{(k)}, \mathbf{D}$):
13 foreach $\mathbf{s}_i \in \mathbf{D}$ do
14     foreach $\mathbf{r} \in \mathcal{C}^{(k)}$ do
15         if $\mathbf{r} \subseteq \mathbf{s}_i$ then $sup(\mathbf{r}) \leftarrow sup(\mathbf{r}) + 1$

ExtendPrefixTree ($\mathcal{C}^{(k)}$):
16 foreach leaf $\mathbf{r}_a \in \mathcal{C}^{(k)}$ do
17     foreach leaf $\mathbf{r}_b \in$ Children(Parent($\mathbf{r}_a$)) do
18         $\mathbf{r}_{ab} \leftarrow \mathbf{r}_a + \mathbf{r}_b[k]$ // extend $\mathbf{r}_a$ with last item of $\mathbf{r}_b$
           // prune if there are any infrequent subsequences
19         if $\mathbf{r}_c \in \mathcal{C}^{(k)}$, for all $\mathbf{r}_c \subset \mathbf{r}_{ab}$, such that $|\mathbf{r}_c| = |\mathbf{r}_{ab}| - 1$ then
20             Add $\mathbf{r}_{ab}$ as child of $\mathbf{r}_a$ with $sup(\mathbf{r}_{ab}) \leftarrow 0$
21     if no extensions from $\mathbf{r}_a$ then
22         remove $\mathbf{r}_a$, and all ancestors of $\mathbf{r}_a$ with no extensions, from $\mathcal{C}^{(k)}$
23 return $\mathcal{C}^{(k)}$
```

bound on the size of the search space:

$$|\Sigma|^1 + |\Sigma|^2 + \cdots + |\Sigma|^l = O(|\Sigma|^l) \tag{10.1}$$

since at level k there are $|\Sigma|^k$ possible subsequences of length k.

10.2.1 Level-wise Mining: GSP

We can devise an effective sequence mining algorithm that searches the sequence prefix tree using a level-wise or breadth-first search. Given the set of frequent sequences at level k, we generate all possible sequence extensions or *candidates* at level $k+1$. We next compute the support of each candidate and prune those that are not frequent. The search stops when no more frequent extensions are possible.

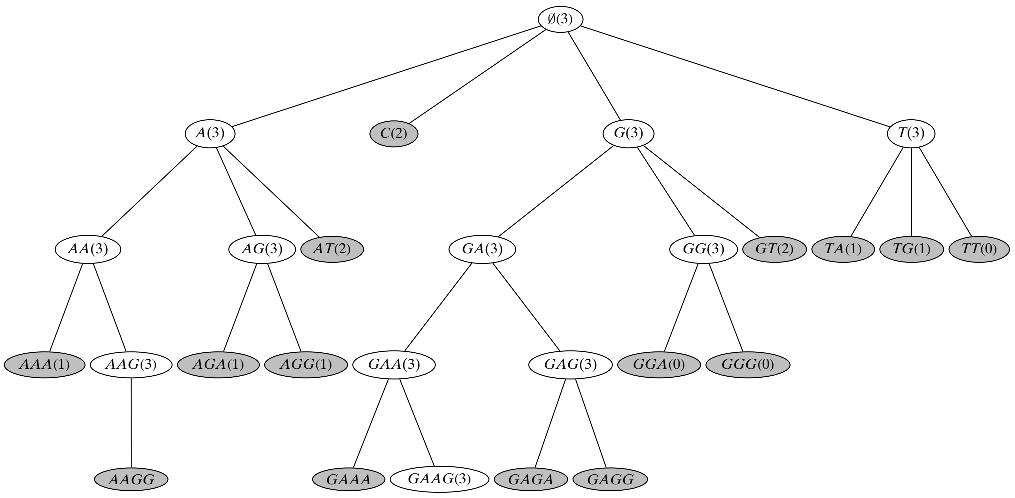

Figure 10.1. Sequence search space: shaded ovals represent candidates that are infrequent; those without support in brackets can be pruned based on an infrequent subsequence. Unshaded ovals represent frequent sequences.

The pseudo-code for the level-wise, generalized sequential pattern (GSP) mining method is shown in Algorithm 10.1. It uses the antimonotonic property of support to prune candidate patterns, that is, no supersequence of an infrequent sequence can be frequent, and all subsequences of a frequent sequence must be frequent. The prefix search tree at level k is denoted $\mathcal{C}^{(k)}$. Initially $\mathcal{C}^{(1)}$ comprises all the symbols in Σ. Given the current set of candidate k-sequences $\mathcal{C}^{(k)}$, the method first computes their support (line 6). For each database sequence $\mathbf{s}_i \in \mathbf{D}$, we check whether a candidate sequence $\mathbf{r} \in \mathcal{C}^{(k)}$ is a subsequence of $\mathbf{s}_i$. If so, we increment the support of $\mathbf{r}$. Once the frequent sequences at level k have been found, we generate the candidates for level $k+1$ (line 10). For the extension, each leaf $\mathbf{r}_a$ is extended with the last symbol of any other leaf $\mathbf{r}_b$ that shares the same prefix (i.e., has the same parent), to obtain the new candidate $(k+1)$-sequence $\mathbf{r}_{ab} = \mathbf{r}_a + \mathbf{r}_b[k]$ (line 18). If the new candidate $\mathbf{r}_{ab}$ contains any infrequent k-sequence, we prune it.

Example 10.3. For example, let us mine the database shown in Table 10.1 using $minsup = 3$. That is, we want to find only those subsequences that occur in all three database sequences. Figure 10.1 shows that we begin by extending the empty sequence ∅ at level 0, to obtain the candidates A, C, G, and T at level 1. Out of these C can be pruned because it is not frequent. Next we generate all possible candidates at level 2. Notice that using A as the prefix we generate all possible extensions AA, AG, and AT. A similar process is repeated for the other two symbols G and T. Some candidate extensions can be pruned without counting. For example, the extension $GAAA$ obtained from GAA can be pruned because it has an infrequent subsequence AAA. The figure shows all the frequent sequences (unshaded), out of which $GAAG(3)$ and $T(3)$ are the maximal ones.

The computational complexity of GSP is $O(|\Sigma|^l)$ as per Eq. (10.1), where l is the length of the longest frequent sequence. The I/O complexity is $O(l \cdot \mathbf{D})$ because we compute the support of an entire level in one scan of the database.

10.2.2 Vertical Sequence Mining: Spade

The Spade algorithm uses a vertical database representation for sequence mining. The idea is to record for each symbol the sequence identifiers and the positions where it occurs. For each symbol $s \in \Sigma$, we keep a set of tuples of the form $\langle i, pos(s)\rangle$, where $pos(s)$ is the set of positions in the database sequence $\mathbf{s}_i \in \mathbf{D}$ where symbol s appears. Let $\mathcal{L}(s)$ denote the set of such sequence-position tuples for symbol s, which we refer to as the *poslist*. The set of poslists for each symbol $s \in \Sigma$ thus constitutes a vertical representation of the input database. In general, given k-sequence $\mathbf{r}$, its poslist $\mathcal{L}(\mathbf{r})$ maintains the list of positions for the occurrences of the last symbol $\mathbf{r}[k]$ in each database sequence $\mathbf{s}_i$, provided $\mathbf{r} \subseteq \mathbf{s}_i$. The support of sequence $\mathbf{r}$ is simply the number of distinct sequences in which $\mathbf{r}$ occurs, that is, $sup(\mathbf{r}) = |\mathcal{L}(\mathbf{r})|$.

Example 10.4. In Table 10.1, the symbol A occurs in $\mathbf{s}_1$ at positions 2, 4, and 5. Thus, we add the tuple $\langle 1, \{2,4,5\}\rangle$ to $\mathcal{L}(A)$. Because A also occurs at positions 3 and 5 in sequence $\mathbf{s}_2$, and at positions 2 and 3 in $\mathbf{s}_3$, the complete poslist for A is $\{\langle 1,\{2,4,5\}\rangle, \langle 2,\{3,5\}\rangle, \langle 1,\{2,3\}\rangle\}$. We have $sup(A) = 3$, as its poslist contains three tuples. Figure 10.2 shows the poslist for each symbol, as well as other sequences. For example, for sequence GT, we find that it is a subsequence of $\mathbf{s}_1$ and $\mathbf{s}_3$.

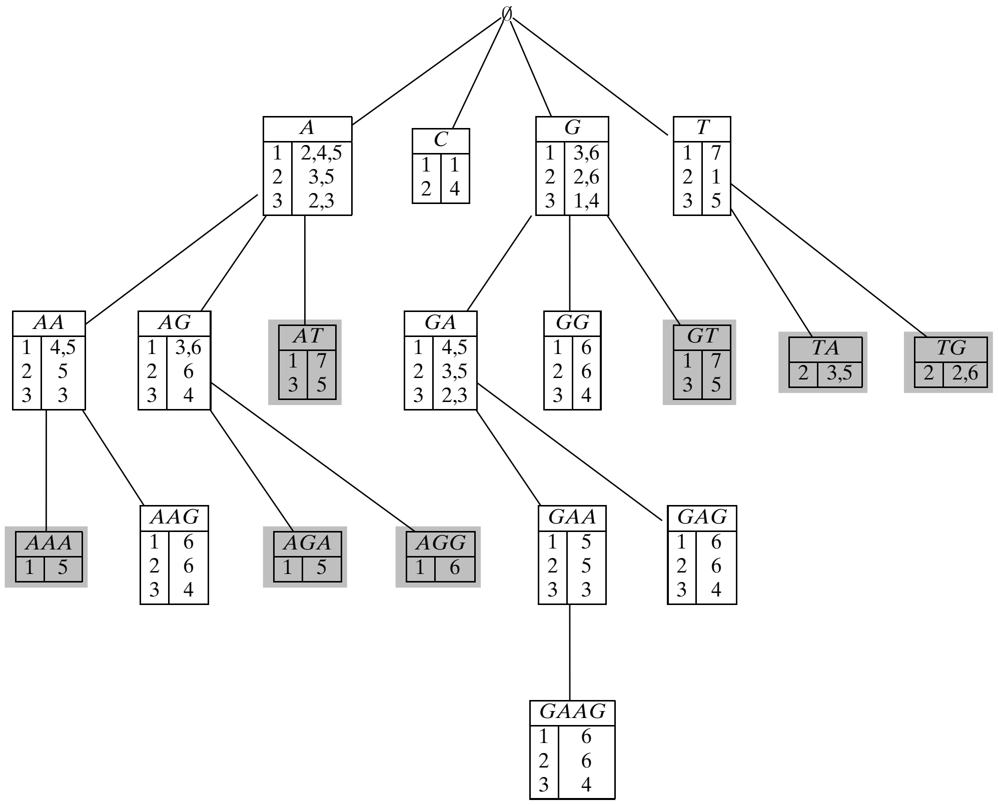

Figure 10.2. Sequence mining via Spade: infrequent sequences with at least one occurrence are shown shaded; those with zero support are not shown.

Even though there are two occurrences of GT in $\mathbf{s}_1$, the last symbol T occurs at position 7 in both occurrences, thus the poslist for GT has the tuple $\langle 1,7\rangle$. The full poslist for GT is $\mathcal{L}(GT) = \{\langle 1,7\rangle, \langle 3,5\rangle\}$. The support of GT is $sup(GT) = |\mathcal{L}(GT)| = 2$.

Support computation in Spade is done via *sequential join* operations. Given the poslists for any two k-sequences $\mathbf{r}_a$ and $\mathbf{r}_b$ that share the same $(k-1)$ length prefix, the idea is to perform sequential joins on the poslists to compute the support for the new $(k+1)$ length candidate sequence $\mathbf{r}_{ab} = \mathbf{r}_a + \mathbf{r}_b[k]$. Given a tuple $\langle i, pos(\mathbf{r}_b[k])\rangle \in \mathcal{L}(\mathbf{r}_b)$, we first check if there exists a tuple $\langle i, pos(\mathbf{r}_a[k])\rangle \in \mathcal{L}(\mathbf{r}_a)$, that is, both sequences must occur in the same database sequence $\mathbf{s}_i$. Next, for each position $p \in pos(\mathbf{r}_b[k])$, we check whether there exists a position $q \in pos(\mathbf{r}_a[k])$ such that $q < p$. If yes, this means that the symbol $\mathbf{r}_b[k]$ occurs after the last position of $\mathbf{r}_a$ and thus we retain p as a valid occurrence of $\mathbf{r}_{ab}$. The poslist $\mathcal{L}(\mathbf{r}_{ab})$ comprises all such valid occurrences. Notice how we keep track of positions only for the last symbol in the candidate sequence. This is because we extend sequences from a common prefix, so there is no need to keep track of all the occurrences of the symbols in the prefix. We denote the sequential join as $\mathcal{L}(\mathbf{r}_{ab}) = \mathcal{L}(\mathbf{r}_a) \cap \mathcal{L}(\mathbf{r}_b)$.

The main advantage of the vertical approach is that it enables different search strategies over the sequence search space, including breadth or depth-first search. Algorithm 10.2 shows the pseudo-code for Spade. Given a set of sequences P that share the same prefix, along with their poslists, the method creates a new prefix equivalence class P_a for each sequence $\mathbf{r}_a \in P$ by performing sequential joins with every sequence $\mathbf{r}_b \in P$, including self-joins. After removing the infrequent extensions, the new equivalence class P_a is then processed recursively.

ALGORITHM 10.2. Algorithm Spade

```
// Initial Call: $\mathcal{F} \leftarrow \emptyset$, $k \leftarrow 0$,
    $P \leftarrow \{\langle s, \mathcal{L}(s)\rangle \mid s \in \Sigma, sup(s) \geq minsup\}$
SPADE ($P$, $minsup$, $\mathcal{F}$, $k$):
1 foreach $\mathbf{r}_a \in P$ do
2     $\mathcal{F} \leftarrow \mathcal{F} \cup \{(\mathbf{r}_a, sup(\mathbf{r}_a))\}$
3     $P_a \leftarrow \emptyset$
4     foreach $\mathbf{r}_b \in P$ do
5         $\mathbf{r}_{ab} = \mathbf{r}_a + \mathbf{r}_b[k]$
6         $\mathcal{L}(\mathbf{r}_{ab}) = \mathcal{L}(\mathbf{r}_a) \cap \mathcal{L}(\mathbf{r}_b)$
7         if $sup(\mathbf{r}_{ab}) \geq minsup$ then
8             $P_a \leftarrow P_a \cup \{\langle \mathbf{r}_{ab}, \mathcal{L}(\mathbf{r}_{ab})\rangle\}$
9     if $P_a \neq \emptyset$ then SPADE ($\mathbf{P}$, $minsup$, $\mathcal{F}$, $k+1$)
```

Example 10.5. Consider the poslists for A and G shown in Figure 10.2. To obtain $\mathcal{L}(AG)$, we perform a sequential join over the poslists $\mathcal{L}(A)$ and $\mathcal{L}(G)$. For the tuples $\langle 1, \{2,4,5\}\rangle \in \mathcal{L}(A)$ and $\langle 1, \{3,6\}\rangle \in \mathcal{L}(G)$, both positions 3 and 6 for G, occur after some occurrence of A, for example, at position 2. Thus, we add the tuple $\langle 1, \{3,6\}\rangle$ to $\mathcal{L}(AG)$. The complete poslist for AG is $\mathcal{L}(AG) = \{\langle 1, \{3,6\}\rangle, \langle 2, 6\rangle, \langle 3, 4\rangle\}$.

Figure 10.2 illustrates the complete working of the Spade algorithm, along with all the candidates and their poslists.

10.2.3 Projection-Based Sequence Mining: PrefixSpan

Let $\mathbf{D}$ denote a database, and let $s \in \Sigma$ be any symbol. The *projected database* with respect to s, denoted $\mathbf{D}_s$, is obtained by finding the the first occurrence of s in $\mathbf{s}_i$, say at position p. Next, we retain in $\mathbf{D}_s$ only the suffix of $\mathbf{s}_i$ starting at position $p+1$. Further, any infrequent symbols are removed from the suffix. This is done for each sequence $\mathbf{s}_i \in \mathbf{D}$.

Example 10.6. Consider the three database sequences in Table 10.1. Given that the symbol G first occurs at position 3 in $\mathbf{s}_1 = CAGAAGT$, the projection of $\mathbf{s}_1$ with respect to G is the suffix $AAGT$. The projected database for G, denoted $\mathbf{D}_G$ is therefore given as: $\{\mathbf{s}_1 : AAGT,\ \mathbf{s}_2 : AAG,\ \mathbf{s}_3 : AAGT\}$.

The main idea in PrefixSpan is to compute the support for only the individual symbols in the projected database $\mathbf{D}_s$, and then to perform recursive projections on the frequent symbols in a depth-first manner. The PrefixSpan method is outlined in Algorithm 10.3. Here $\mathbf{r}$ is a frequent subsequence, and $\mathbf{D}_\mathbf{r}$ is the projected dataset for $\mathbf{r}$. Initially $\mathbf{r}$ is empty and $\mathbf{D}_\mathbf{r}$ is the entire input dataset $\mathbf{D}$. Given a database of (projected) sequences $\mathbf{D}_\mathbf{r}$, PrefixSpan first finds all the frequent symbols in the projected dataset. For each such symbol s, we extend $\mathbf{r}$ by appending s to obtain the new frequent subsequence $\mathbf{r}_s$. Next, we create the projected dataset $\mathbf{D}_s$ by projecting $\mathbf{D}_\mathbf{r}$ on symbol s. A recursive call to PrefixSpan is then made with $\mathbf{r}_s$ and $\mathbf{D}_s$.

ALGORITHM 10.3. Algorithm PrefixSpan

```
   // Initial Call: $\mathbf{D_r} \leftarrow \mathbf{D}$, $\mathbf{r} \leftarrow \emptyset$, $\mathcal{F} \leftarrow \emptyset$
   PREFIXSPAN ($\mathbf{D_r}$, $\mathbf{r}$, $minsup$, $\mathcal{F}$):
1  foreach $s \in \Sigma$ such that $sup(s, \mathbf{D_r}) \geq minsup$ do
2      $\mathbf{r}_s = \mathbf{r} + s$ // extend $\mathbf{r}$ by symbol $s$
3      $\mathcal{F} \leftarrow \mathcal{F} \cup \{(\mathbf{r}_s, sup(s, \mathbf{D_r}))\}$
4      $\mathbf{D}_s \leftarrow \emptyset$ // create projected data for symbol $s$
5      foreach $\mathbf{s}_i \in \mathbf{D_r}$ do
6          $\mathbf{s}'_i \leftarrow$ projection of $\mathbf{s}_i$ w.r.t symbol $s$
7          Remove any infrequent symbols from $\mathbf{s}'_i$
8          Add $\mathbf{s}'_i$ to $\mathbf{D}_s$ if $\mathbf{s}'_i \neq \emptyset$
9      if $\mathbf{D}_s \neq \emptyset$ then PREFIXSPAN ($\mathbf{D}_s$, $\mathbf{r}_s$, $minsup$, $\mathcal{F}$)
```

Example 10.7. Figure 10.3 shows the projection-based PrefixSpan mining approach for the example dataset in Table 10.1 using *minsup* = 3. Initially we start with the whole database $\mathbf{D}$, which can also be denoted as $\mathbf{D}_\emptyset$. We compute the support of each symbol, and find that C is not frequent (shown crossed out). Among the frequent symbols, we first create a new projected dataset $\mathbf{D}_A$. For $\mathbf{s}_1$, we find that the first A occurs at position 2, so we retain only the suffix $GAAGT$. In $\mathbf{s}_2$, the first A occurs at position 3, so the suffix is CAG. After removing C (because it is infrequent), we are left with only AG as the projection of $\mathbf{s}_2$ on A. In a similar manner we obtain the projection for $\mathbf{s}_3$ as AGT. The left child of the root shows the final projected dataset $\mathbf{D}_A$. Now the mining proceeds recursively. Given $\mathbf{D}_A$, we count the symbol supports in $\mathbf{D}_A$, finding that only A and G are frequent, which will lead to the projection $\mathbf{D}_{AA}$ and then $\mathbf{D}_{AG}$, and so on. The complete projection-based approach is illustrated in Figure 10.3.

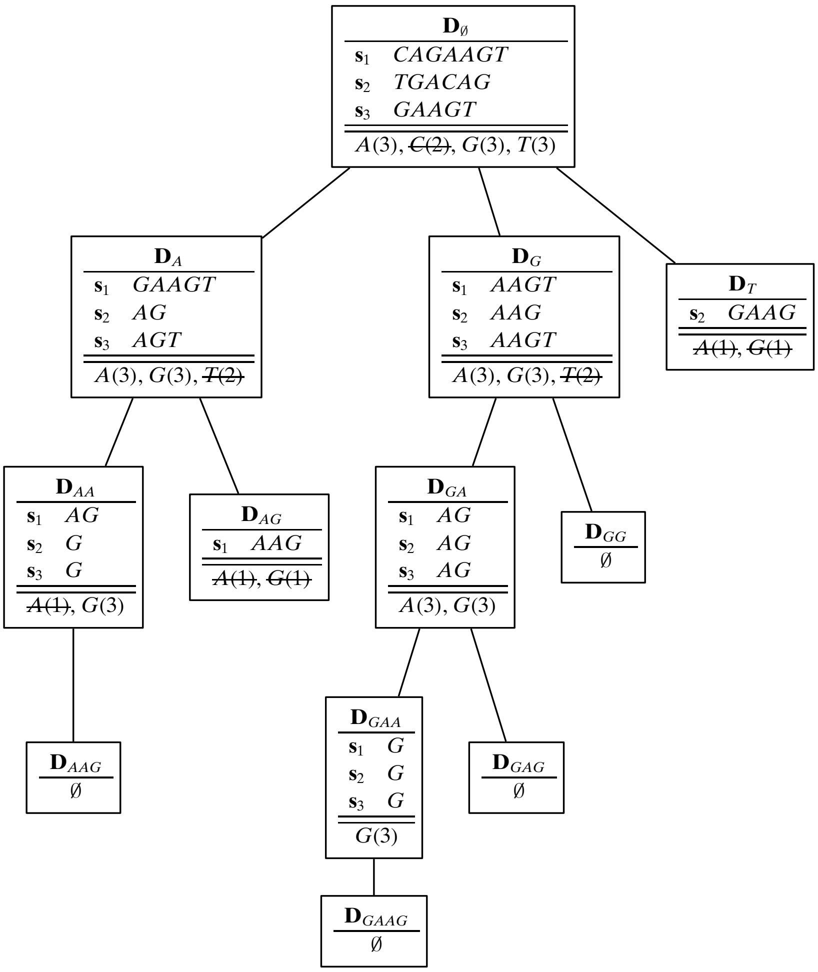

Figure 10.3. Projection-based sequence mining: PrefixSpan.

10.3 SUBSTRING MINING VIA SUFFIX TREES

We now look at efficient methods for mining frequent substrings. Let $\mathbf{s}$ be a sequence having length n, then there are at most $O(n^2)$ possible distinct substrings contained in $\mathbf{s}$. To see this consider substrings of length w, of which there are $n-w+1$ possible ones in $\mathbf{s}$. Adding over all substring lengths we get

$$\sum_{w=1}^{n}(n-w+1) = n+(n-1)+\cdots+2+1 = O(n^2)$$

This is a much smaller search space compared to subsequences, and consequently we can design more efficient algorithms for solving the frequent substring mining task. In fact, we can mine all the frequent substrings in worst case $O(Nn^2)$ time for a dataset $\mathbf{D} = \{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_N\}$ with N sequences.

10.3.1 Suffix Tree

Let Σ denote the alphabet, and let $\$ \notin \Sigma$ be a *terminal* character used to mark the end of a string. Given a sequence $\mathbf{s}$, we append the terminal character so that $\mathbf{s} = s_1 s_2 \ldots s_n s_{n+1}$, where $s_{n+1} = \$$, and the jth suffix of $\mathbf{s}$ is given as $\mathbf{s}[j:n+1] = s_j s_{j+1} \ldots s_{n+1}$. The *suffix tree* of the sequences in the database $\mathbf{D}$, denoted $\mathcal{T}$, stores all the suffixes for each $\mathbf{s}_i \in \mathbf{D}$ in a tree structure, where suffixes that share a common prefix lie on the same path from the root of the tree. The substring obtained by concatenating all the symbols from the root node to a node v is called the *node label* of v, and is denoted as $L(v)$. The substring that appears on an edge (v_a, v_b) is called an *edge label*, and is denoted as $L(v_a, v_b)$. A suffix tree has two kinds of nodes: internal and leaf nodes. An internal node in the suffix tree (except for the root) has at least two children, where each edge label to a child begins with a different symbol. Because the terminal character is unique, there are as many leaves in the suffix tree as there are unique suffixes over all the sequences. Each leaf node corresponds to a suffix shared by one or more sequences in $\mathbf{D}$.

It is straightforward to obtain a quadratic time and space suffix tree construction algorithm. Initially, the suffix tree $\mathcal{T}$ is empty. Next, for each sequence $\mathbf{s}_i \in \mathbf{D}$, with $|\mathbf{s}_i| = n_i$, we generate all its suffixes $\mathbf{s}_i[j:n_i+1]$, with $1 \le j \le n_i$, and insert each of them into the tree by following the path from the root until we either reach a leaf or there is a mismatch in one of the symbols along an edge. If we reach a leaf, we insert the pair (i,j) into the leaf, noting that this is the jth suffix of sequence $\mathbf{s}_i$. If there is a mismatch in one of the symbols, say at position $p \ge j$, we add an internal vertex just before the mismatch, and create a new leaf node containing (i,j) with edge label $\mathbf{s}_i[p:n_i+1]$.

Example 10.8. Consider the database in Table 10.1 with three sequences. In particular, let us focus on $\mathbf{s}_1 = CAGAAGT$. Figure 10.4 shows what the suffix tree $\mathcal{T}$ looks like after inserting the jth suffix of $\mathbf{s}_1$ into $\mathcal{T}$. The first suffix is the entire sequence $\mathbf{s}_1$ appended with the terminal symbol; thus the suffix tree contains a single leaf containing $(1,1)$ under the root (Figure 10.4a). The second suffix is $AGAAGT\$$, and Figure 10.4b shows the resulting suffix tree, which now has two leaves. The third

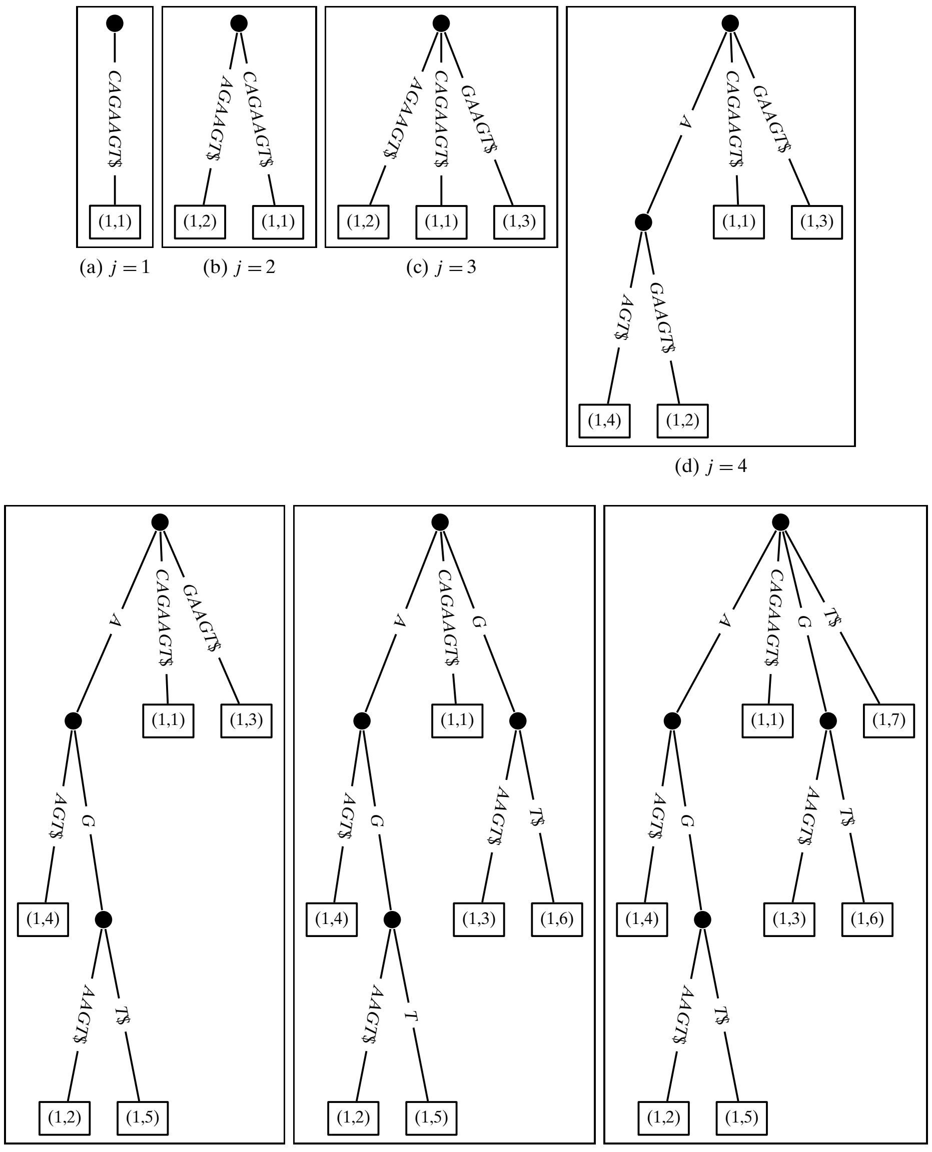

Figure 10.4. Suffix tree construction: (a)–(g) show the successive changes to the tree, after we add the jth suffix of $\mathbf{s}_1 = CAGAAGT\$$ for $j = 1,\dots,7$.

suffix $GAAGT\$$ begins with G, which has not yet been observed, so it creates a new leaf in $\mathcal{T}$ under the root. The fourth suffix $AAGT\$$ shares the prefix A with the second suffix, so it follows the path beginning with A from the root. However, because there is a mismatch at position 2, we create an internal node right before it and insert the leaf $(1,4)$, as shown in Figure 10.4d. The suffix tree obtained after inserting all of the suffixes of $\mathbf{s}_1$ is shown in Figure 10.4g, and the complete suffix tree for all three sequences is shown in Figure 10.5.

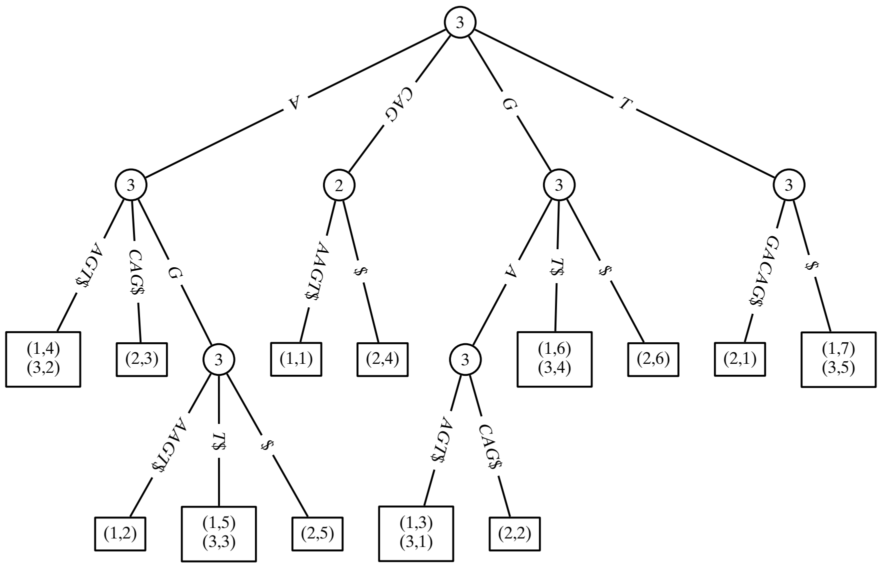

Figure 10.5. Suffix tree for all three sequences in Table 10.1. Internal nodes store support information. Leaves also record the support (not shown).

In terms of the time and space complexity, the algorithm sketched above requires $O(Nn^2)$ time and space, where N is the number of sequences in $\mathbf{D}$, and n is the longest sequence length. The time complexity follows from the fact that the method always inserts a new suffix starting from the root of the suffix tree. This means that in the worst case it compares $O(n)$ symbols per suffix insertion, giving the worst case bound of $O(n^2)$ over all n suffixes. The space complexity comes from the fact that each suffix is explicitly represented in the tree, taking $n + (n-1) + \cdots + 1 = O(n^2)$ space. Over all the N sequences in the database, we obtain $O(Nn^2)$ as the worst case time and space bounds.

Frequent Substrings

Once the suffix tree is built, we can compute all the frequent substrings by checking how many different sequences appear in a leaf node or under an internal node. The node labels for the nodes with support at least *minsup* yield the set of frequent substrings; all the prefixes of such node labels are also frequent. The suffix tree can also support ad hoc queries for finding all the occurrences in the database for any query substring $\mathbf{q}$. For each symbol in $\mathbf{q}$, we follow the path from the root until all symbols in $\mathbf{q}$ have been seen, or until there is a mismatch at any position. If $\mathbf{q}$ is found, then the set of leaves under that path is the list of occurrences of the query $\mathbf{q}$. On the other hand, if there is mismatch that means the query does not occur in the database. In terms of the query time complexity, because we have to match each character in $\mathbf{q}$, we immediately get $O(|\mathbf{q}|)$ as the time bound (assuming that $|\Sigma|$ is a constant), which is *independent* of the size of the database. Listing all the matches takes additional time, for a total time complexity of $O(|\mathbf{q}| + k)$, if there are k matches.

Example 10.9. Consider the suffix tree shown in Figure 10.5, which stores all the suffixes for the sequence database in Table 10.1. To facilitate frequent substring enumeration, we store the support for each internal as well as leaf node, that is, we store the number of distinct sequence ids that occur at or under each node. For example, the leftmost child of the root node on the path labeled A has support 3 because there are three distinct sequences under that subtree. If *minsup* $= 3$, then the frequent substrings are A, AG, G, GA, and T. Out of these, the maximal ones are AG, GA, and T. If *minsup* $= 2$, then the maximal frequent substrings are $GAAGT$ and CAG.

For ad hoc querying consider $\mathbf{q} = GAA$. Searching for symbols in $\mathbf{q}$ starting from the root leads to the leaf node containing the occurrences $(1,3)$ and $(3,1)$, which means that GAA appears at position 3 in $\mathbf{s}_1$ and at position 1 in $\mathbf{s}_3$. On the other hand if $\mathbf{q} = CAA$, then the search terminates with a mismatch at position 3 after following the branch labeled CAG from the root. This means that $\mathbf{q}$ does not occur in the database.

10.3.2 Ukkonen's Linear Time Algorithm

We now present a linear time and space algorithm for constructing suffix trees. We first consider how to build the suffix tree for a single sequence $\mathbf{s} = s_1 s_2 \dots s_n s_{n+1}$, with $s_{n+1} = \$$. The suffix tree for the entire dataset of N sequences can be obtained by inserting each sequence one by one.

Achieving Linear Space

Let us see how to reduce the space requirements of a suffix tree. If an algorithm stores all the symbols on each edge label, then the space complexity is $O(n^2)$, and we cannot achieve linear time construction either. The trick is to not explicitly store all the edge labels, but rather to use an *edge-compression* technique, where we store only the starting and ending positions of the edge label in the input string $\mathbf{s}$. That is, if an edge label is given as $\mathbf{s}[i : j]$, then we represent is as the interval $[i, j]$.

Example 10.10. Consider the suffix tree for $\mathbf{s}_1 = CAGAAGT\$$ shown in Figure 10.4g. The edge label $CAGAAGT\$$ for the suffix $(1,1)$ can be represented via the interval $[1,8]$ because the edge label denotes the substring $\mathbf{s}_1[1 : 8]$. Likewise, the edge label $AAGT\$$ leading to suffix $(1,2)$ can be compressed as $[4,8]$ because $AAGT\$ = \mathbf{s}_1[4 : 8]$. The complete suffix tree for $\mathbf{s}_1$ with compressed edge labels is shown in Figure 10.6.

In terms of space complexity, note that when we add a new suffix to the tree $\mathcal{T}$, it can create at most one new internal node. As there are n suffixes, there are n leaves in $\mathcal{T}$ and at most n internal nodes. With at most $2n$ nodes, the tree has at most $2n - 1$ edges, and thus the total space required to store an interval for each edge is $2(2n - 1) = 4n - 2 = O(n)$.

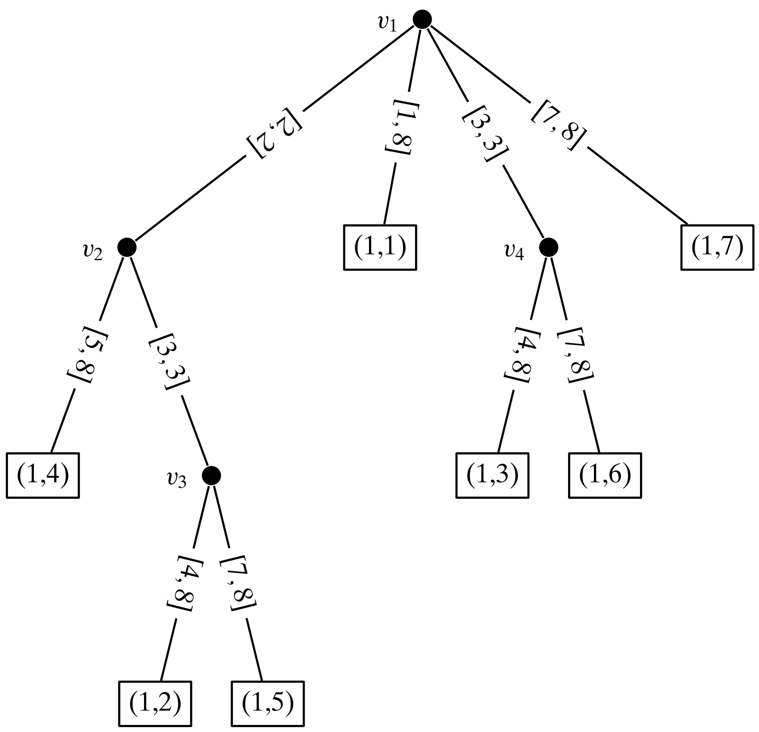

Figure 10.6. Suffix tree for $\mathbf{s}_1 = CAGAAGT\$$ using edge-compression.

Achieving Linear Time

Ukkonen's method is an *online* algorithm, that is, given a string $\mathbf{s} = s_1 s_2 \ldots s_n\$$ it constructs the full suffix tree in phases. Phase i builds the tree up to the i-th symbol in $\mathbf{s}$, that is, it updates the suffix tree from the previous phase by adding the next symbol s_i. Let $\mathcal{T}_i$ denote the suffix tree up to the ith prefix $\mathbf{s}[1:i]$, with $1 \le i \le n$. Ukkonen's algorithm constructs $\mathcal{T}_i$ from $\mathcal{T}_{i-1}$, by making sure that all suffixes including the *current* character s_i are in the new intermediate tree $\mathcal{T}_i$. In other words, in the ith phase, it inserts all the suffixes $\mathbf{s}[j:i]$ from $j = 1$ to $j = i$ into the tree $\mathcal{T}_i$. Each such insertion is called the jth *extension* of the ith *phase*. Once we process the terminal character at position $n+1$ we obtain the final suffix tree $\mathcal{T}$ for $\mathbf{s}$.

Algorithm 10.4 shows the code for a naive implementation of Ukkonen's approach. This method has cubic time complexity because to obtain $\mathcal{T}_i$ from $\mathcal{T}_{i-1}$ takes $O(i^2)$ time, with the last phase requiring $O(n^2)$ time. With n phases, the total time is $O(n^3)$. Our goal is to show that this time can be reduced to just $O(n)$ via the optimizations described in the following paragraghs.

Implicit Suffixes This optimization states that, in phase i, if the jth extension $\mathbf{s}[j:i]$ is found in the tree, then any subsequent extensions will also be found, and consequently there is no need to process further extensions in phase i. Thus, the suffix tree $\mathcal{T}_i$ at the end of phase i has *implicit suffixes* corresponding to extensions $j+1$ through i. It is important to note that all suffixes will become explicit the first time we encounter a new substring that does not already exist in the tree. This will surely happen in phase

ALGORITHM 10.4. Algorithm NaiveUkkonen

```
NaiveUkkonen (s):
1 $n \leftarrow |\mathbf{s}|$
2 $\mathbf{s}[n+1] \leftarrow \$$ // append terminal character
3 $\mathcal{T} \leftarrow \emptyset$ // add empty string as root
4 foreach $i = 1, \ldots, n+1$ do // phase $i$ - construct $\mathcal{T}_i$
5     foreach $j = 1, \ldots, i$ do // extension $j$ for phase $i$
          // Insert $\mathbf{s}[j:i]$ into the suffix tree
6         Find end of the path with label $\mathbf{s}[j:i-1]$ in $\mathcal{T}$
7         Insert $s_i$ at end of path;
8 return $\mathcal{T}$
```

$n+1$ when we process the terminal character \$, as it cannot occur anywhere else in **s** (after all, $\$ \not\in \Sigma$).

Implicit Extensions Let the current phase be i, and let $l \le i-1$ be the last explicit suffix in the previous tree $\mathcal{T}_{i-1}$. All explicit suffixes in $\mathcal{T}_{i-1}$ have edge labels of the form $[x, i-1]$ leading to the corresponding leaf nodes, where the starting position x is node specific, but the ending position must be $i-1$ because s_{i-1} was added to the end of these paths in phase $i-1$. In the current phase i, we would have to extend these paths by adding s_i at the end. However, instead of explicitly incrementing all the ending positions, we can replace the ending position by a pointer e which keeps track of the current phase being processed. If we replace $[x, i-1]$ with $[x, e]$, then in phase i, if we set $e = i$, then immediately all the l existing suffixes get *implicitly* extended to $[x, i]$. Thus, in one operation of incrementing e we have, in effect, taken care of extensions 1 through l for phase i.

Example 10.11. Let $\mathbf{s}_1 = CAGAAGT\$$. Assume that we have already performed the first six phases, which result in the tree $\mathcal{T}_6$ shown in Figure 10.7a. The last explicit suffix in $\mathcal{T}_6$ is $l = 4$. In phase $i = 7$ we have to execute the following extensions:

$CAGAAGT$	extension 1
$AGAAGT$	extension 2
$GAAGT$	extension 3
$AAGT$	extension 4
AGT	extension 5
GT	extension 6
T	extension 7

At the start of the seventh phase, we set $e = 7$, which yields implicit extensions for all suffixes explicitly in the tree, as shown in Figure 10.7b. Notice how symbol $s_7 = T$ is now implicitly on each of the leaf edges, for example, the label $[5, e] = AG$ in $\mathcal{T}_6$ now becomes $[5, e] = AGT$ in $\mathcal{T}_7$. Thus, the first four extensions listed above are taken care of by simply incrementing e. To complete phase 7 we have to process the remaining extensions.

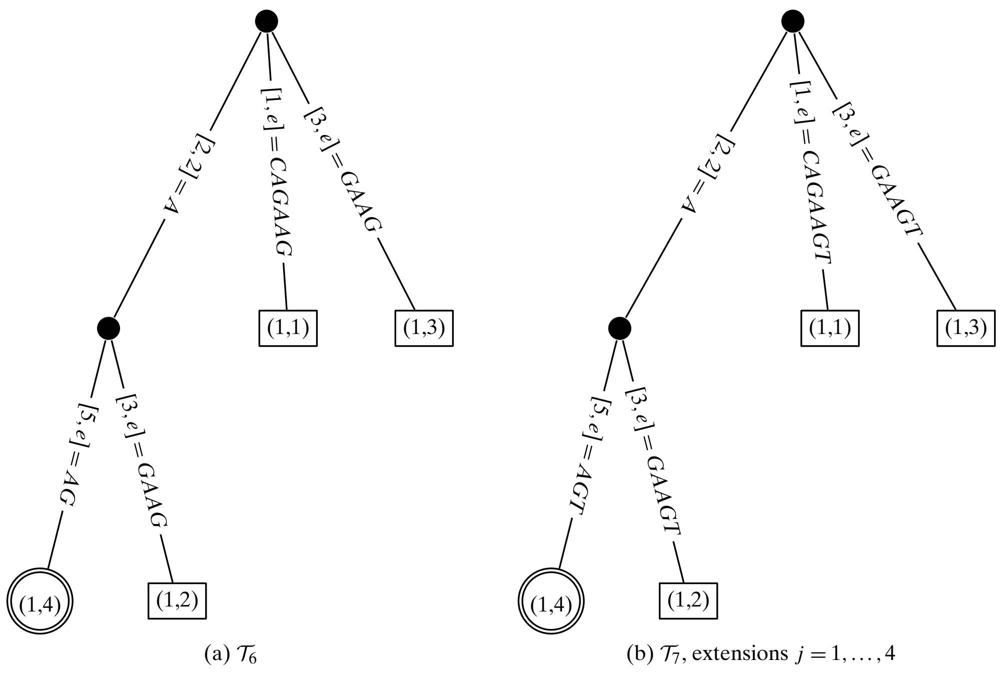

Figure 10.7. Implicit extensions in phase $i = 7$. Last explicit suffix in $\mathcal{T}_6$ is $l = 4$ (shown double-circled). Edge labels shown for convenience; only the intervals are stored.

Skip/Count Trick For the jth extension of phase i, we have to search for the substring $\mathbf{s}[j : i-1]$ so that we can add s_i at the end. However, note that this string must exist in $\mathcal{T}_{i-1}$ because we have already processed symbol s_{i-1} in the previous phase. Thus, instead of searching for each character in $\mathbf{s}[j : i-1]$ starting from the root, we first *count* the number of symbols on the edge beginning with character s_j; let this length be m. If m is longer than the length of the substring (i.e., if $m > i - j$), then the substring must end on this edge, so we simply jump to position $i - j$ and insert s_i. On the other hand, if $m \leq i - j$, then we can *skip* directly to the child node, say v_c, and search for the remaining string $\mathbf{s}[j+m : i-1]$ from v_c using the same skip/count technique. With this optimization, the cost of an extension becomes proportional to the number of nodes on the path, as opposed to the number of characters in $\mathbf{s}[j : i-1]$.

Suffix Links We saw that with the skip/count optimization we can search for the substring $\mathbf{s}[j : i-1]$ by following nodes from parent to child. However, we still have to start from the root node each time. We can avoid searching from the root via the use of *suffix links*. For each internal node v_a we maintain a link to the internal node v_b, where $L(v_b)$ is the immediate suffix of $L(v_a)$. In extension $j-1$, let v_p denote the internal node under which we find $\mathbf{s}[j-1 : i]$, and let m be the length of the node label of v_p. To insert the jth extension $\mathbf{s}[j : i]$, we follow the suffix link from v_p to another node, say v_s, and search for the remaining substring $\mathbf{s}[j+m-1 : i-1]$ from v_s. The use of suffix links allows us to jump internally within the tree for different extensions, as opposed to searching from the root each time. As a final observation, if extension j

ALGORITHM 10.5. Algorithm Ukkonen

```
UKKONEN (s):
1  $n \leftarrow |\mathbf{s}|$
2  $\mathbf{s}[n+1] \leftarrow \$$ // append terminal character
3  $\mathcal{T} \leftarrow \emptyset$ // add empty string as root
4  $l \leftarrow 0$ // last explicit suffix
5  foreach $i = 1, \dots, n+1$ do // phase $i$ - construct $\mathcal{T}_i$
6      $e \leftarrow i$ // implicit extensions
7      foreach $j = l+1, \dots, i$ do // extension $j$ for phase $i$
           // Insert $\mathbf{s}[j:i]$ into the suffix tree
8          Find end of $\mathbf{s}[j:i-1]$ in $\mathcal{T}$ via skip/count and suffix links
9          if $s_i \in \mathcal{T}$ then // implicit suffixes
10             break
11         else
12             Insert $s_i$ at end of path
13             Set last explicit suffix $l$ if needed
14 return $\mathcal{T}$
```

creates a new internal node, then its suffix link will point to the new internal node that will be created during extension $j+1$.

The pseudo-code for the optimized Ukkonen's algorithm is shown in Algorithm 10.5. It is important to note that it achieves linear time and space only with all of the optimizations in conjunction, namely implicit extensions (line 6), implicit suffixes (line 9), and skip/count and suffix links for inserting extensions in $\mathcal{T}$ (line 8).

Example 10.12. Let us look at the execution of Ukkonen's algorithm on the sequence $\mathbf{s}_1 = CAGAAGT\$$, as shown in Figure 10.8. In phase 1, we process character $s_1 = C$ and insert the suffix $(1,1)$ into the tree with edge label $[1,e]$ (see Figure 10.8a). In phases 2 and 3, new suffixes $(1,2)$ and $(1,3)$ are added (see Figures 10.8b–10.8c). For phase 4, when we want to process $s_4 = A$, we note that all suffixes up to $l = 3$ are already explicit. Setting $e = 4$ implicitly extends all of them, so we have only to make sure that the last extension $(j = 4)$ consisting of the single character A is in the tree. Searching from the root, we find A in the tree implicitly, and we thus proceed to the next phase. In the next phase, we set $e = 5$, and the suffix $(1,4)$ becomes explicit when we try to add the extension AA, which is not in the tree. For $e = 6$, we find the extension AG already in the tree and we skip ahead to the next phase. At this point the last explicit suffix is still $(1,4)$. For $e = 7$, T is a previously unseen symbol, and so all suffixes will become explicit, as shown in Figure 10.8g.

It is instructive to see the extensions in the last phase $(i = 7)$. As described in Example 10.11, the first four extensions will be done implicitly. Figure 10.9a shows the suffix tree after these four extensions. For extension 5, we begin at the last explicit

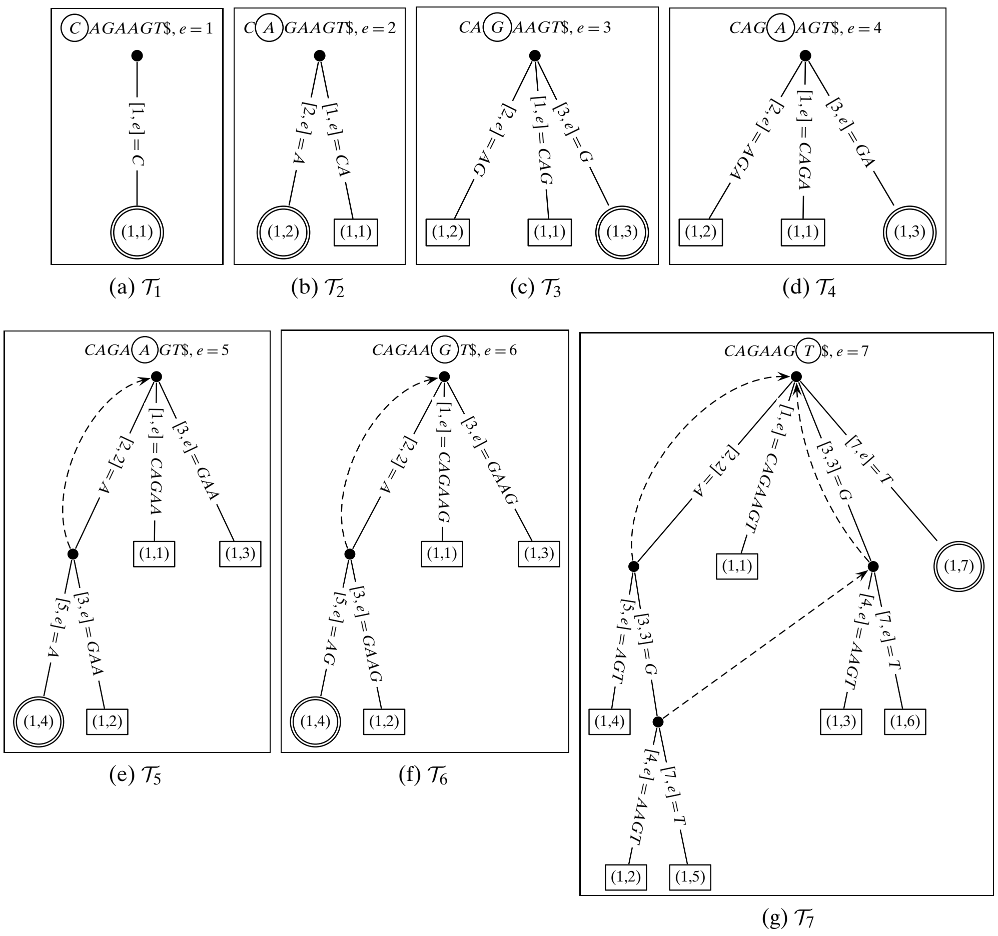

Figure 10.8. Ukkonen's linear time algorithm for suffix tree construction. Steps (a)–(g) show the successive changes to the tree after the ith phase. The suffix links are shown with dashed lines. The double-circled leaf denotes the last explicit suffix in the tree. The last step is not shown because when $e = 8$, the terminal character \$ will not alter the tree. All the edge labels are shown for ease of understanding, although the actual suffix tree keeps only the intervals for each edge.

leaf, follow its parent's suffix link, and begin searching for the remaining characters from that point. In our example, the suffix link points to the root, so we search for $\mathbf{s}[5:7] = AGT$ from the root. We skip to node v_A, and look for the remaining string GT, which has a mismatch inside the edge $[3, e]$. We thus create a new internal node after G, and insert the explicit suffix $(1,5)$, as shown in Figure 10.9b. The next extension $\mathbf{s}[6:7] = GT$ begins at the newly created leaf node $(1,5)$. Following the closest suffix link leads back to the root, and a search for GT gets a mismatch on the edge out of the root to leaf $(1,3)$. We then create a new internal node v_G at that point, add a suffix link from the previous internal node v_{AG} to v_G, and add a new explicit leaf $(1,6)$, as shown in Figure 10.9c. The last extension, namely $j = 7$, corresponding

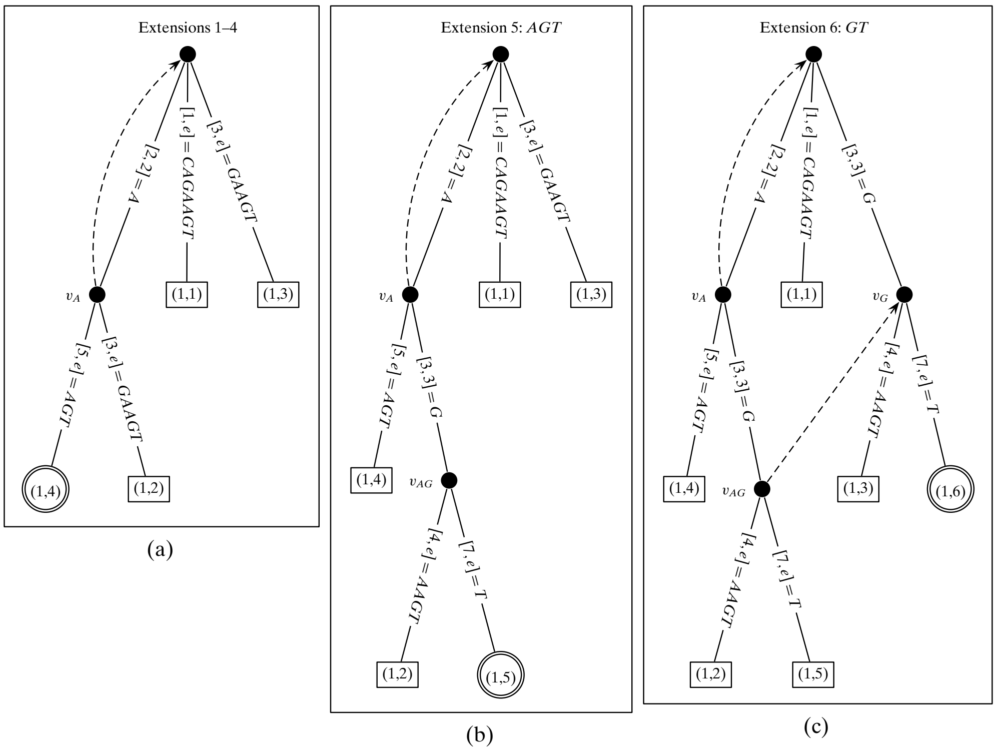

Figure 10.9. Extensions in phase $i = 7$. Initially the last explicit suffix is $l = 4$ and is shown double-circled. All the edge labels are shown for convenience; the actual suffix tree keeps only the intervals for each edge.

to $\mathbf{s}[7:7] = T$, results in making all the suffixes explicit because the symbol T has been seen for the first time. The resulting tree is shown in Figure 10.8g.

Once $\mathbf{s}_1$ has been processed, we can then insert the remaining sequences in the database $\mathbf{D}$ into the existing suffix tree. The final suffix tree for all three sequences is shown in Figure 10.5, with additional suffix links (not shown) from all the internal nodes.

Ukkonen's algorithm has time complexity of $O(n)$ for a sequence of length n because it does only a constant amount of work (amortized) to make each suffix explicit. Note that, for each phase, a certain number of extensions are done implicitly just by incrementing e. Out of the i extensions from $j = 1$ to $j = i$, let us say that l are done implicitly. For the remaining extensions, we stop the first time some suffix is implicitly in the tree; let that extension be k. Thus, phase i needs to add explicit suffixes only for suffixes $l + 1$ through $k - 1$. For creating each explicit suffix, we perform a constant number of operations, which include following the closest suffix link, skip/counting to look for the first mismatch, and inserting if needed a new suffix leaf node. Because each leaf becomes explicit only once, and the number of skip/count steps are bounded by $O(n)$ over the whole tree, we get a worst-case $O(n)$

time algorithm. The total time over the entire database of N sequences is thus $O(Nn)$, if n is the longest sequence length.

10.4 FURTHER READING

The level-wise GSP method for mining sequential patterns was proposed in Srikant and Agrawal (March 1996). Spade is described in Zaki (2001), and the PrefixSpan algorithm in Pei et al. (2004). Ukkonen's linear time suffix tree construction method appears in Ukkonen (1995). For an excellent introduction to suffix trees and their numerous applications see Gusfield (1997); the suffix tree description in this chapter has been heavily influenced by it.

Gusfield, D. (1997). *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. New York: Cambridge University Press.

Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., and Hsu, M.-C. (2004). "Mining sequential patterns by pattern-growth: The PrefixSpan approach." *IEEE Transactions on Knowledge and Data Engineering*, 16 (11): 1424–1440.

Srikant, R. and Agrawal, R. (March 1996). "Mining sequential patterns: Generalizations and performance improvements." *In Proceedings of the 5th International Conference on Extending Database Technology*. New York: Springer-Verlag.

Ukkonen, E. (1995). "On-line construction of suffix trees." *Algorithmica*, 14 (3): 249–260.

Zaki, M. J. (2001). "SPADE: An efficient algorithm for mining frequent sequences." *Machine Learning*, 42 (1–2): 31–60.

10.5 EXERCISES

Q1. Consider the database shown in Table 10.2. Answer the following questions:

(a) Let $minsup = 4$. Find all frequent sequences.

(b) Given that the alphabet is $\Sigma = \{A, C, G, T\}$. How many possible sequences of length k can there be?

Table 10.2. Sequence database for Q1

Id	Sequence
$\mathbf{s}_1$	*AATACAAGAAC*
$\mathbf{s}_2$	*GTATGGTGAT*
$\mathbf{s}_3$	*AACATGGCCAA*
$\mathbf{s}_4$	*AAGCGTGGTCAA*

Q2. Given the DNA sequence database in Table 10.3, answer the following questions using $minsup = 4$

(a) Find the maximal frequent sequences.

(b) Find all the closed frequent sequences.

(c) Find the maximal frequent substrings.
(d) Show how Spade would work on this dataset.
(e) Show the steps of the PrefixSpan algorithm.

Table 10.3. Sequence database for Q2

Id	Sequence
$\mathbf{s}_1$	$ACGTCACG$
$\mathbf{s}_2$	$TCGA$
$\mathbf{s}_3$	$GACTGCA$
$\mathbf{s}_4$	$CAGTC$
$\mathbf{s}_5$	$AGCT$
$\mathbf{s}_6$	$TGCAGCTC$
$\mathbf{s}_7$	$AGTCAG$

Q3. Given $\mathbf{s} = AABBACBBAA$, and $\Sigma = \{A, B, C\}$. Define support as the number of occurrence of a subsequence in $\mathbf{s}$. Using *minsup* $= 2$, answer the following questions:

(a) Show how the vertical Spade method can be extended to mine all frequent substrings (consecutive subsequences) in $\mathbf{s}$.
(b) Construct the suffix tree for $\mathbf{s}$ using Ukkonen's method. Show all intermediate steps, including all suffix links.
(c) Using the suffix tree from the previous step, find all the occurrences of the query $\mathbf{q} = ABBA$ allowing for at most two mismatches.
(d) Show the suffix tree when we add another character A just before the \$. That is, you must undo the effect of adding the \$, add the new symbol A, and then add \$ back again.
(e) Describe an algorithm to extract all the maximal frequent substrings from a suffix tree. Show all maximal frequent substrings in $\mathbf{s}$.

Q4. Consider a bitvector based approach for mining frequent subsequences. For instance, in Table 10.2, for $\mathbf{s}_1$, the symbol C occurs at positions 5 and 11. Thus, the bitvector for C in $\mathbf{s}_1$ is given as 00001000001. Because C does not appear in $\mathbf{s}_2$ its bitvector can be omitted for $\mathbf{s}_2$. The complete set of bitvectors for symbol C is

$$(\mathbf{s}_1, 00001000001)$$
$$(\mathbf{s}_3, 00100001100)$$
$$(\mathbf{s}_4, 000100000100)$$

Given the set of bitvectors for each symbol show how we can mine all frequent subsequences by using bit operations on the bitvectors. Show the frequent subsequences and their bitvectors using *minsup* $= 4$.

Q5. Consider the database shown in Table 10.4. Each sequence comprises itemset events that happen at the same time. For example, sequence $\mathbf{s}_1$ can be considered to be a sequence of itemsets $(AB)_{10}(B)_{20}(AB)_{30}(AC)_{40}$, where symbols within brackets are considered to co-occur at the same time, which is given in the subscripts. Describe an algorithm that can mine all the frequent subsequences over itemset events. The

Table 10.4. Sequences for Q5

Id	Time	Items
$\mathbf{s}_1$	10	A,B
	20	B
	30	A,B
	40	A,C
$\mathbf{s}_2$	20	A,C
	30	A,B,C
	50	B
$\mathbf{s}_3$	10	A
	30	B
	40	A
	50	C
	60	B
$\mathbf{s}_4$	30	A,B
	40	A
	50	B
	60	C

itemsets can be of any length as long as they are frequent. Find all frequent itemset sequences with $minsup = 3$.

Q6. The suffix tree shown in Figure 10.5 contains all suffixes for the three sequences $\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3$ in Table 10.1. Note that a pair (i, j) in a leaf denotes the jth suffix of sequence $\mathbf{s}_i$.

(a) Add a new sequence $\mathbf{s}_4 = GAAGCAGAA$ to the existing suffix tree, using the Ukkonen algorithm. Show the last character position (e), along with the suffixes (l) as they become explicit in the tree for $\mathbf{s}_4$. Show the final suffix tree after all suffixes of $\mathbf{s}_4$ have become explicit.

(b) Find all closed frequent substrings with $minsup = 2$ using the final suffix tree.

Q7. Given the following three sequences:

$$\mathbf{s}_1 : GAAGT$$
$$\mathbf{s}_2 : CAGAT$$
$$\mathbf{s}_3 : ACGT$$

Find all the frequent subsequences with $minsup = 2$, but allowing at most a gap of 1 position between successive sequence elements.