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 \( \Sigma \) 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 \( s = s_1s_2\ldots s_k \), where \( s_i \in \Sigma \) is a symbol at position \( i \), also denoted as \( s[i] \). Here \(|s| = k\) denotes the length of the sequence. A sequence with length \( k \) is also called a \( k \)-sequence. We use the notation \( s[i:j] = s_is_{i+1}\ldots s_{j-1}s_j \) to denote the substring or sequence of consecutive symbols in positions \( i \) through \( j \), where \( j > i \). Define the prefix of a sequence \( s \) as any substring of the form \( s[1:i] = s_1s_2\ldots s_i \), with \( 0 \leq i \leq n \). Also, define the suffix of \( s \) as any substring of the form \( s[i:n] = s_is_{i+1}\ldots s_n \), with \( 1 \leq i \leq n+1 \). Note that \( s[1:0] \) is the empty prefix, and \( s[n+1:n] \) is the empty suffix. Let \( \Sigma^* \) be the set of all possible sequences that can be constructed using the symbols in \( \Sigma \), including the empty sequence \( \emptyset \) (which has length zero).

Let \( s = s_1s_2\ldots s_n \) and \( r = r_1r_2\ldots r_m \) be two sequences over \( \Sigma \). We say that \( r \) is a subsequence of \( s \) denoted \( r \subseteq s \), if there exists a one-to-one mapping \( \phi : [1,m] \to [1,n] \), such that \( r[i] = s[\phi(i)] \) and for any two positions \( i, j \) in \( r \), \( i < j \implies \phi(i) < \phi(j) \). In other words, each position in \( r \) is mapped to a different position in \( s \), and the order of symbols is preserved, even though there may be intervening gaps between consecutive elements of \( r \) in the mapping. If \( r \subseteq s \), we also say that \( s \) contains \( r \). The sequence \( r \) is called a consecutive subsequence or substring of \( s \), provided \( r_1r_2\ldots r_m = s_is_{i+1}\ldots s_{j+m-1} \), i.e., \( r[1:m] = s[j:j+m-1] \), with \( 1 \leq j \leq n-m+1 \). For substrings we do not allow any gaps between the elements of \( r \) in the mapping.

**Example 10.1.** Let \( \Sigma = \{A, C, G, T\} \), and let \( s = ACTGAACG \). Then \( r_1 = CGAAG \) is a subsequence of \( s \), and \( r_2 = CTGA \) is a substring of \( s \). The sequence \( r_3 = ACT \) is a prefix of \( s \), and so is \( r_4 = ACTGA \), whereas \( r_5 = GAACG \) is one of the suffixes of \( s \).
Given a database \( D = \{s_1, s_2, \ldots, s_N\} \) of \( N \) sequences, and given some sequence \( r \), the support of \( r \) in the database \( D \) is defined as the total number of sequences in \( D \) that contain \( r \)

\[
sup(r) = |\{s_i \in D | r \subseteq s_i\}|
\]

The relative support of \( r \) is the fraction of sequences that contain \( r \)

\[
rsup(r) = \frac{sup(r)}{N}
\]

Given a user-specified minsup threshold, we say that a sequence \( r \) is frequent in database \( D \) if \( sup(r) \geq \text{minsup} \). 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 \( s = s_1s_2\ldots s_k \) at level \( k \) has children of the form \( s' = s_1s_2\ldots s_k s_{k+1} \) at level \( k + 1 \). In other words, \( s \) is a prefix of each child \( s' \), which is also called an extension of \( s \).

**Example 10.2.** Let \( \Sigma = \{A, C, G, T\} \) and let the sequence database \( D \) 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 \( \text{minsup} = 3 \).

The subsequence search space is conceptually infinite because it comprises all sequences in \( \Sigma^* \), that is, all sequences of length zero or more that can be created using symbols in \( \Sigma \). In practice, the database \( D \) 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>
<thead>
<tr>
<th>Id</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>CAGAAGT</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>TGACAG</td>
</tr>
<tr>
<td>( s_3 )</td>
<td>GAAGT</td>
</tr>
</tbody>
</table>
 Algorithm GSP

\begin{algorithm}
\textbf{GSP} \((D, \Sigma, \text{minsup})\):
\begin{algorithmic}
\State \(\mathcal{F} \leftarrow \emptyset\)
\State \(C^{(1)} \leftarrow \{\emptyset\}\) // Initial prefix tree with single symbols
\ForAll \(s \in \Sigma\) \text{ Add } s \text{ as child of } \emptyset \text{ in } C^{(1)} \text{ with } \sup(s) \leftarrow 0\)
\State \(k \leftarrow 1\) // \(k\) denotes the level
\While \(C^{(k)} \neq \emptyset\)
\State \text{COMPUTE SUPPORT} \((C^{(k)}, D)\)
\ForAll \(s \in C^{(k)}\)
\If \(\sup(r) \geq \text{minsup}\) then \(\mathcal{F} \leftarrow \mathcal{F} \cup \{(r, \sup(r))\}\)
\Else remove \(s\) from \(C^{(k)}\)
\EndIf
\State \(C^{(k+1)} \leftarrow \text{EXTEND PREFIX TREE} \((C^{(k)}))\)
\State \(k \leftarrow k + 1\)
\EndWhile
\State \text{return } \mathcal{F}^{(k)}
\end{algorithmic}
\end{algorithm}

\textbf{COMPUTE SUPPORT} \((C^{(k)}, D)\):
\begin{algorithmic}
\ForAll \(s_i \in D\)
\ForAll \(r \in C^{(k)}\)
\If \(r \subseteq s_i\) then \(\sup(r) \leftarrow \sup(r) + 1\)
\EndIf
\EndFor
\EndFor
\end{algorithmic}

\textbf{EXTEND PREFIX TREE} \((C^{(k)})\):
\begin{algorithmic}
\ForAll \(r_a \in C^{(k)}\)
\ForAll \(r_b \in \text{CHILDREN}(\text{PARENT}(r_a))\)
\State \(r_{ab} \leftarrow r_a + r_b[k]\) // extend \(r_a\) with last item of \(r_b\)
\If there are any infrequent subsequences
\If \(r_c \in C^{(k)}, \text{ for all } r_c \subseteq r_{ab}, \text{ such that } |r_c| = |r_{ab}| - 1\) then
\State Add \(r_{ab}\) as child of \(r_a\) with \(\sup(r_{ab}) \leftarrow 0\)
\EndIf
\EndIf
\If no extensions from \(r_a\) then
\State remove \(r_a\), and all ancestors of \(r_a\) with no extensions, from \(C^{(k)}\)
\EndIf
\EndFor
\EndFor
\State \text{return } C^{(k)}
\end{algorithmic}

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.
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 \( C^{(k)} \). Initially \( C^{(1)} \) comprises all the symbols in \( \Sigma \). Given the current set of candidate \( k \)-sequences \( C^{(k)} \), the method first computes their support (line 6). For each database sequence \( s_i \in D \), we check whether a candidate sequence \( r \in C^{(k)} \) is a subsequence of \( s_i \). If so, we increment the support of \( r \). 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 \( r_a \) is extended with the last symbol of any other leaf \( r_b \) that shares the same prefix (i.e., has the same parent), to obtain the new candidate \( (k + 1) \)-sequence \( r_{ab} = r_a + r_b[k] \) (line 18). If the new candidate \( r_{ab} \) contains any infrequent \( k \)-sequence, we prune it.

**Example 10.3.** For example, let us mine the database shown in Table 10.1 using \( \text{minsup} = 3 \). 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 \( \emptyset \) 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 \( GAAG \) 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 \( O(l \cdot D) \) because we compute the support of an entire level in one scan of the database.