Data Mining and Analysis: Fundamental Concepts and Algorithms
dataminingbook.info

Mohammed J. Zaki\textsuperscript{1} Wagner Meira Jr.\textsuperscript{2}

\textsuperscript{1}Department of Computer Science
Rensselaer Polytechnic Institute, Troy, NY, USA

\textsuperscript{2}Department of Computer Science
Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Chapter 10: Sequence Mining
Let \( \Sigma \) be the *alphabet*, a set of symbols. A *sequence* or a *string* is defined as an ordered list of symbols, and is written as \( s = s_1 s_2 \ldots s_k \), where \( s_i \in \Sigma \) is a symbol at position \( i \), also denoted as \( s[i] \). \( |s| = k \) denotes the *length* of the sequence.

The notation \( s[i : j] = s_i s_{i+1} \ldots s_{j-1} s_j \) denotes 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 \( s[1 : i] = s_1 s_2 \ldots s_i \), with \( 0 \leq i \leq n \).

Define the *suffix* of \( s \) as any substring of the form \( s[i : n] = s_i s_{i+1} \ldots s_n \), with \( 1 \leq i \leq n+1 \).

\( s[1 : 0] \) 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 $s = s_1 s_2 \ldots s_n$ and $r = r_1 r_2 \ldots r_m$ 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 $\phi : [1, m] \rightarrow [1, n]$, such that $r[i] = s[\phi(i)]$ and for any two positions $i, j$ in $r$, $i < j \implies \phi(i) < \phi(j)$. In if $r \subseteq s$, we also say that $s$ contains $r$.

The sequence $r$ is called a consecutive subsequence or substring of $s$ provided $r_1 r_2 \ldots r_m = s_j s_{j+1} \ldots s_{j+m-1}$, i.e., $r[1 : m] = s[j : j + m - 1]$, with $1 \leq j \leq n - m + 1$.

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 $s$. 
Frequent Sequences

Given a database $D = \{s_1, s_2, \ldots, s_N\}$ of $N$ sequences, and given some sequence $r$, the \textit{support} of $r$ in the database $D$ is defined as the total number of sequences in $D$ that contain $r$

$$sup(r) = \left| \{s_i \in D | r \subseteq s_i \} \right|$$

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

$$rsup(r) = \frac{sup(r)}{N}$$

Given a user-specified $\text{minsup}$ threshold, we say that a sequence $r$ is \textit{frequent} in database $D$ if $sup(r) \geq \text{minsup}$. A frequent sequence is \textit{maximal} if it is not a subsequence of any other frequent sequence, and a frequent sequence is \textit{closed} if it is not a subsequence of any other frequent sequence with the same support.
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 \( s' = s_1s_2 \ldots s_ks_{k+1} \) at level \( k+1 \). In other words, \( s \) is a prefix of each child \( s' \), which is also called an extension of \( s \).
Using $minsup = 3$, the set of frequent subsequences is given as:

\[
\mathcal{F}^{(1)} = \{A(3), G(3), T(3)\}
\]

\[
\mathcal{F}^{(2)} = \{AA(3), AG(3), GA(3), GG(3)\}
\]

\[
\mathcal{F}^{(3)} = \{AAG(3), GAA(3), GAG(3)\}
\]

\[
\mathcal{F}^{(4)} = \{GAAG(3)\}
\]
The GSP algorithm 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 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.

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 $k + 1$.

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 $r_{ab} = r_a + r_b[k]$. If the new candidate $r_{ab}$ contains any infrequent $k$-sequence, we prune it.
Algorithm GSP

GSP (D, Σ, \( \text{minsup} \)):

1. \( \mathcal{F} \leftarrow \emptyset \)
2. \( C^{(1)} \leftarrow \{\emptyset\} \) // Initial prefix tree with single symbols
3. \textbf{foreach} \( s \in \Sigma \) do Add \( s \) as child of \( \emptyset \) in \( C^{(1)} \) with \( \text{sup}(s) \leftarrow 0 \)
4. \( k \leftarrow 1 \) // \( k \) denotes the level
5. \textbf{while} \( C^{(k)} \neq \emptyset \) do
6. \textbf{COMPUTE} SUPPORT \((C^{(k)}, D)\)
7. \textbf{foreach} leaf \( s \in C^{(k)} \) do
8. \hspace{1em} \textbf{if} \( \text{sup}(r) \geq \text{minsup} \) \textbf{then} \( \mathcal{F} \leftarrow \mathcal{F} \cup \{(r, \text{sup}(r))\} \)
9. \hspace{1em} \textbf{else} remove \( s \) from \( C^{(k)} \)
10. \( C^{(k+1)} \leftarrow \text{EXTENDPREFIXTREE} \( (C^{(k)}) \) \)
11. \( k \leftarrow k + 1 \)
12. \textbf{return} \( \mathcal{F}^{(k)} \)
Algorithm **COMPUTE_SUPPORT**

**COMPUTE_SUPPORT** \((C^{(k)}, D)\):

1. **foreach** \(s_i \in D\) **do**
2.   **foreach** \(r \in C^{(k)}\) **do**
3.     **if** \(r \subseteq s_i\) **then** \(sup(r) \leftarrow sup(r) + 1\)

**EXTENDPREFIXTREE** \((C^{(k)})\):

1. **foreach** \(leaf r_a \in C^{(k)}\) **do**
2.   **foreach** \(leaf r_b \in CHILDREN(PARENT(r_a))\) **do**
3.     \(r_{ab} \leftarrow r_a + r_b[k]\) // extend \(r_a\) with last item of \(r_b\)
     // prune if there are any infrequent subsequences
4.     **if** \(r_c \in C^{(k)}, for all r_c \subset r_{ab}, such that |r_c| = |r_{ab}| - 1\) **then**
5.       **Add** \(r_{ab}\) as child of \(r_a\) with \(sup(r_{ab}) \leftarrow 0\)
6. **if no extensions from \(r_a\) then**
7.   **remove** \(r_a\), and all ancestors of \(r_a\) with no extensions, from \(C^{(k)}\)
8. **return** \(C^{(k)}\)
Sequence Search Space

shaded ovals are infrequent sequences
The Spade algorithm uses a vertical database representation for sequence mining. For each symbol $s \in \Sigma$, we keep a set of tuples of the form $\langle i, \text{pos}(s) \rangle$, where $\text{pos}(s)$ is the set of positions in the database sequence $s_i \in D$ 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.

Let $\mathcal{L}(s)$ denote the set of such sequence-position tuples Given $k$-sequence $r$, its poslist $\mathcal{L}(r)$ maintains the list of positions for the occurrences of the last symbol $r[k]$ in each database sequence $s_i$, provided $r \subseteq s_i$. The support of sequence $r$ is simply the number of distinct sequences in which $r$ occurs, that is, $\text{sup}(r) = |\mathcal{L}(r)|$. 
Spade Algorithm

Support computation in Spade is done via *sequential join* operations.

Given the poslists for any two $k$-sequences $r_a$ and $r_b$ that share the same $(k - 1)$ length prefix, a sequential join on the poslists is used to compute the support for the new $(k + 1)$ length candidate sequence $r_{ab} = r_a + r_b[k]$.

Given a tuple $\langle i, \text{pos}(r_b[k]) \rangle \in \mathcal{L}(r_b)$, we first check if there exists a tuple $\langle i, \text{pos}(r_a[k]) \rangle \in \mathcal{L}(r_a)$, that is, both sequences must occur in the same database sequence $s_i$.

Next, for each position $p \in \text{pos}(r_b[k])$, we check whether there exists a position $q \in \text{pos}(r_a[k])$ such that $q < p$. If yes, this means that the symbol $r_b[k]$ occurs after the last position of $r_a$ and thus we retain $p$ as a valid occurrence of $r_{ab}$. The poslist $\mathcal{L}(r_{ab})$ comprises all such valid occurrences.

We keep track of positions only for the last symbol in the candidate sequence since we extend sequences from a common prefix, and so there is no need to keep track of all the occurrences of the symbols in the prefix.

We denote the sequential join as $\mathcal{L}(r_{ab}) = \mathcal{L}(r_a) \cap \mathcal{L}(r_b)$.
Spade Algorithm

// Initial Call: \( F \leftarrow \emptyset, k \leftarrow 0, \)
\( P \leftarrow \{ \langle s, L(s) \rangle \mid s \in \Sigma, \text{sup}(s) \geq \text{minsup} \} \)

**SPADE** \((P, \text{minsup}, F, k)\):

1. **foreach** \( r_a \in P \) do
   2. \( F \leftarrow F \cup \{ (r_a, \text{sup}(r_a)) \} \)
   3. \( P_a \leftarrow \emptyset \)
   4. **foreach** \( r_b \in P \) do
      5. \( r_{ab} = r_a + r_b[k] \)
      6. \( L(r_{ab}) = L(r_a) \cap L(r_b) \)
      7. **if** \( \text{sup}(r_{ab}) \geq \text{minsup} \) **then**
      8. \( P_a \leftarrow P_a \cup \{ \langle r_{ab}, L(r_{ab}) \rangle \} \)
   9. **if** \( P_a \neq \emptyset \) **then** **SPADE** \((P, \text{minsup}, F, k + 1)\)
Let $D$ denote a database, and let $s \in \Sigma$ be any symbol. The *projected database* with respect to $s$, denoted $D_s$, is obtained by finding the first occurrence of $s$ in $s_i$, say at position $p$. Next, we retain in $D_s$ only the suffix of $s_i$ starting at position $p + 1$. Further, any infrequent symbols are removed from the suffix. This is done for each sequence $s_i \in D$.

PrefixSpan computes the support for only the individual symbols in the projected database $D_s$; it then performs recursive projections on the frequent symbols in a depth-first manner.

Given a frequent subsequence $r$, let $D_r$ be the projected dataset for $r$. Initially $r$ is empty and $D_r$ is the entire input dataset $D$. Given a database of (projected) sequences $D_r$, PrefixSpan first finds all the frequent symbols in the projected dataset. For each such symbol $s$, we extend $r$ by appending $s$ to obtain the new frequent subsequence $r_s$. Next, we create the projected dataset $D_s$ by projecting $D_r$ on symbol $s$. A recursive call to PrefixSpan is then made with $r_s$ and $D_s$. 
PrefixSpan Algorithm

// Initial Call: Dr ← D, r ← ∅, F ← ∅

PREFIXSPAN (Dr, r, minsup, F):

1. foreach s ∈ Σ such that sup(s, Dr) ≥ minsup do
   2. rs = r + s // extend r by symbol s
   3. F ← F ∪ {(rs, sup(s, Dr))}
   4. Ds ← ∅ // create projected data for symbol s
   foreach si ∈ Dr do
     5. s' = projection of si w.r.t symbol s
     6. Remove any infrequent symbols from s'
     7. Add s' to Ds if s' ≠ ∅
   8. if Ds ≠ ∅ then PREFIXSPAN (Ds, rs, minsup, F)
Projection-based Sequence Mining: PrefixSpan

D₀
- s₁: CAGAAGT
- s₂: TGACAG
- s₃: GAAGT
  - A(3), G(2), G(3), T(3)

Dₐ
- s₁: GAAGT
- s₂: AG
- s₃: AGT
  - A(3), G(3), T(2)

D₉
- s₁: AAGT
- s₂: AAG
- s₃: AAGT
  - A(3), G(3), T(2)

Dₜ
- s₂: GAAG
  - A(4), G(1)

Dₐₐ
- s₁: AG
- s₂: G
- s₃: G
  - A(1), G(3)

Dₐₙ
- s₁: AAG
- s₂: AG
- s₃: AG
  - A(1), G(4)

Dₐₜ
- s₁: G
- s₂: G
- s₃: G
  - G(3)
Let $s$ be a sequence having length $n$, then there are at most $O(n^2)$ possible distinct substrings contained in $s$. 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.

Naively, we can mine all the frequent substrings in worst case $O(Nn^2)$ time for a dataset $D = \{s_1, s_2, \ldots, s_N\}$ with $N$ sequences.

We will show that all sequences can be mined in $O(Nn)$ time via Suffix Trees.
Given a sequence $s$, we append a terminal character $\not\in \Sigma$ so that $s = s_1s_2 \ldots s_n s_{n+1}$, where $s_{n+1} = \$, and the $j$th suffix of $s$ is given as $s[j : n + 1] = s_j s_{j+1} \ldots s_{n+1}$.

The suffix tree of the sequences in the database $D$, denoted $\mathcal{T}$, stores all the suffixes for each $s_i \in D$ 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 $D$. 
Suffix Tree Construction for $s = \text{CAGAAGT}\$ $\$$

Insert each suffix $j$ per step

(a) $j = 1$

(b) $j = 2$

(c) $j = 3$

(d) $j = 4$
Suffix Tree Construction for $s = \text{CAGAAGT}$

Insert each suffix $j$ per step

(e) $j = 5$

(f) $j = 6$

(g) $j = 7$
Suffix Tree for Entire Database

\[ D = \{ s_1 = \text{CAGAAGT}, s_2 = \text{TGACAG}, s_3 = \text{GAAGT} \} \]
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 $\text{minsup}$ 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 $q$. For each symbol in $q$, we follow the path from the root until all symbols in $q$ have been seen, or until there is a mismatch at any position. If $q$ is found, then the set of leaves under that path is the list of occurrences of the query $q$. On the other hand, if there is mismatch that means the query does not occur in the database.

Because we have to match each character in $q$, we immediately get $O(|q|)$ 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 $O(|q| + k)$, if there are $k$ matches.
Achieving Linear Space: 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 $s$. That is, if an edge label is given as $s[i : j]$, then we represent is as the interval $[i, j]$. 
Suffix Tree using Edge-compression: $s = CAGAAGT$.

(a) Full Tree

(b) Compressed Tree
Ukkonen’s method is an online algorithm, that is, given a string $s = s_1s_2\ldots s_n$ it constructs the full suffix tree in phases.

Phase $i$ builds the tree up to the $i$-th symbol in $s$. Let $T_i$ denote the suffix tree up to the $i$th prefix $s[1 : i]$, with $1 \leq i \leq n$. Ukkonen’s algorithm constructs $T_i$ from $T_{i-1}$, by making sure that all suffixes including the current character $s_i$ are in the new intermediate tree $T_i$.

In other words, in the $i$th phase, it inserts all the suffixes $s[j : i]$ from $j = 1$ to $j = i$ into the tree $T_i$. Each such insertion is called the $j$th extension of the $i$th phase.

Once we process the terminal character at position $n + 1$ we obtain the final suffix tree $T$ for $s$.

However, this naive Ukkonen method has cubic time complexity because to obtain $T_i$ from $T_{i-1}$ takes $O(i^2)$ time, with the last phase requiring $O(n^2)$ time. With $n$ phases, the total time is $O(n^3)$. We will show that this time can be reduced to $O(n)$. 
Algorithm NAIVEUKKONEN

NAIVEUKKONEN (s):

1. \( n \leftarrow |s| \)
2. \( s[n + 1] \leftarrow $ // append terminal character
3. \( \mathcal{T} \leftarrow \emptyset // add empty string as root
4. \textbf{foreach} \( i = 1, \ldots, n + 1 \) \textbf{do} // phase \( i \) - construct \( \mathcal{T}_i \)
5. \hspace{1em} \textbf{foreach} \( j = 1, \ldots, i \) \textbf{do} // extension \( j \) for phase \( i \)
6. \hspace{2em} // Insert \( s[j:i] \) into the suffix tree
7. \hspace{2em} Find end of the path with label \( s[j:i-1] \) in \( \mathcal{T} \)
8. \hspace{1em} Insert \( s_i \) at end of path;
9. return \( \mathcal{T} \)
This optimization states that, in phase $i$, if the $j$th extension $s[j : i]$ 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 $T_i$ 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 $n + 1$ when we process the terminal character $\$, as it cannot occur anywhere else in $s$ (after all, $\not\in \Sigma$).
Let the current phase be $i$, and let $l \leq i - 1$ be the last explicit suffix in the previous tree $T_{i-1}$.

All explicit suffixes in $T_{i-1}$ 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 $i$. 
Implicit Extensions: $s = CAGAAGT$, Phase $i = 7$

(a) $\mathcal{T}_6$

(b) $\mathcal{T}_7$, extensions $j = 1, \ldots, 4$
For the $j$th extension of phase $i$, we have to search for the substring $s[j : i - 1]$ so that we can add $s_i$ at the end.

Note that this string must exist in $T_{i-1}$ because we have already processed symbol $s_{i-1}$ in the previous phase. Thus, instead of searching for each character in $s[j : i - 1]$ 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 $s[j + m : i - 1]$ 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 $s[j : i - 1]$. 

We can avoid searching for the substring $s[j : i - 1]$ 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 $s[j - 1 : i]$, and let $m$ be the length of the node label of $v_p$. To insert the $j$th extension $s[j : i]$, we follow the suffix link from $v_p$ to another node, say $v_s$, and search for the remaining substring $s[j + m - 1 : i - 1]$ 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.
**Ukkonen (s):**

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

14. \( \text{return } \mathcal{T} \)
Ukkonen’s Suffix Tree Construction: $s = CAGAAGT$
Ukkonen’s Suffix Tree Construction: \( s = CAGAAGT \)

- For \( e = 5 \), the suffix tree is shown. The tree includes nodes with labels such as \( CAGA \), \( G \), and \( AG \) with corresponding indices.
- For \( e = 6 \), the tree is expanded further with additional nodes and edges.
- For \( e = 7 \), the tree is shown again with updated nodes and edges.

The diagrams illustrate the construction process of the suffix tree with each step incrementing the counter \( e \) and adding new nodes and edges to the tree.
Extensions in Phase $i = 7$

(a) Extensions 1–4

(b) Extension 5: AGT

(c) Extension 6: GT