Data Mining and Analysis: Fundamental Concepts and Algorithms

dataminingbook.info

Mohammed J. Zaki[1] Wagner Meira Jr.[2]

[1]Department of Computer Science
Rensselaer Polytechnic Institute, Troy, NY, USA

[2]Department of Computer Science
Universidade Federal de Minas Gerais, Belo Horizonte, Brazil

Chapter 10: Sequence Mining

Sequence Mining: Terminology

Let Σ be the *alphabet*, a set of symbols. 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 $\mathbf{s}[i]$. $|\mathbf{s}| = k$ denotes the *length* of the sequence.

The notation $\mathbf{s}[i:j] = s_i s_{i+1} \cdots s_{j-1} s_j$ denotes the *substring* or sequence of consecutive ymbols 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 $0 \leq i \leq n$.

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 $1 \leq i \leq n+1$.

$\mathbf{s}[1:0]$ 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).

Sequence Mining: Terminology

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 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 $1 \le j \le n - m + 1$.

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 $\mathbf{s}$.

Frequent Sequences

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 $\mathbf{r}$

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

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.

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 $\mathbf{s} = s_1 s_2 \dots s_k$ at level k has children of the form $\mathbf{s}' = s_1 s_2 \dots s_k s_{k+1}$ 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 Sequence Database

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

Using $minsup = 3$, the set of frequent subsequences is given as:

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

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

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

$$\mathcal{F}^{(4)} = GAAG(3)$$

Level-wise Sequence Mining: GSP Algorithm

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 $\mathcal{C}^{(k)}$. 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.

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 $k+1$.

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 $\mathbf{r}_{ab} = \mathbf{r}_a + \mathbf{r}_b[k]$. If the new candidate $\mathbf{r}_{ab}$ contains any infrequent k-sequence, we prune it.

Algorithm GSP

GSP (D, Σ, *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 ($\mathcal{C}^{(k)}$, **D**)
7.   **foreach** *leaf* **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 **s** from $\mathcal{C}^{(k)}$
10. $\mathcal{C}^{(k+1)} \leftarrow$ ExtendPrefixTree ($\mathcal{C}^{(k)}$)
11. $k \leftarrow k + 1$
12. **return** $\mathcal{F}^{(k)}$

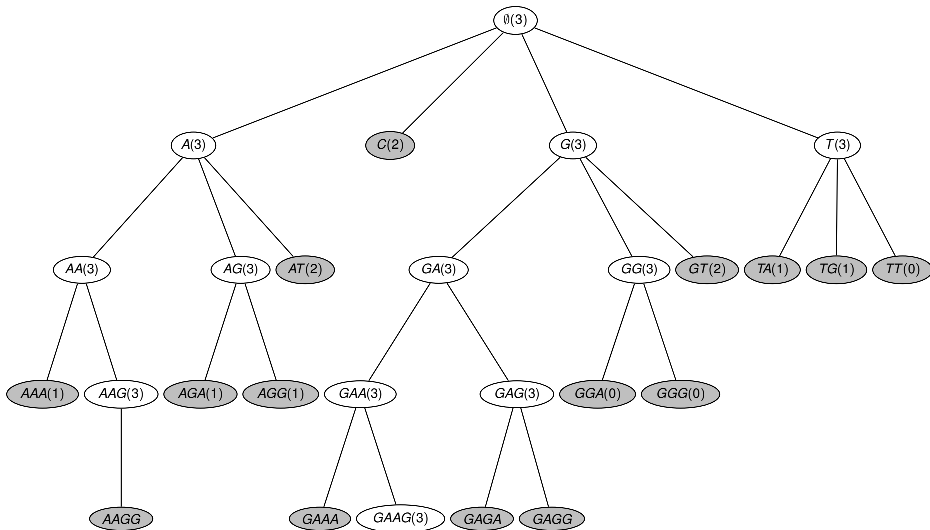
Algorithm COMPUTESUPPORT

```
COMPUTESUPPORT (C^(k), D):
1 foreach s_i ∈ D do
2     foreach r ∈ C^(k) do
3         if r ⊆ s_i then sup(r) ← sup(r) + 1

EXTENDPREFIXTREE (C^(k)):
1 foreach leaf r_a ∈ C^(k) do
2     foreach leaf r_b ∈ CHILDREN(PARENT(r_a)) do
3         r_ab ← 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 ∈ C^(k), for all r_c ⊂ r_ab, such that |r_c| = |r_ab| − 1 then
5             Add r_ab as child of r_a with sup(r_ab) ← 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



Vertical Sequence Mining: Spade

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 $\langle i, pos(s) \rangle$, 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.

Let $\mathcal{L}(s)$ denote the set of such sequence-position tuples 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, $sup(\mathbf{r}) = |\mathcal{L}(\mathbf{r})|$.

Spade Algorithm

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, a sequential join on the poslists is used 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.

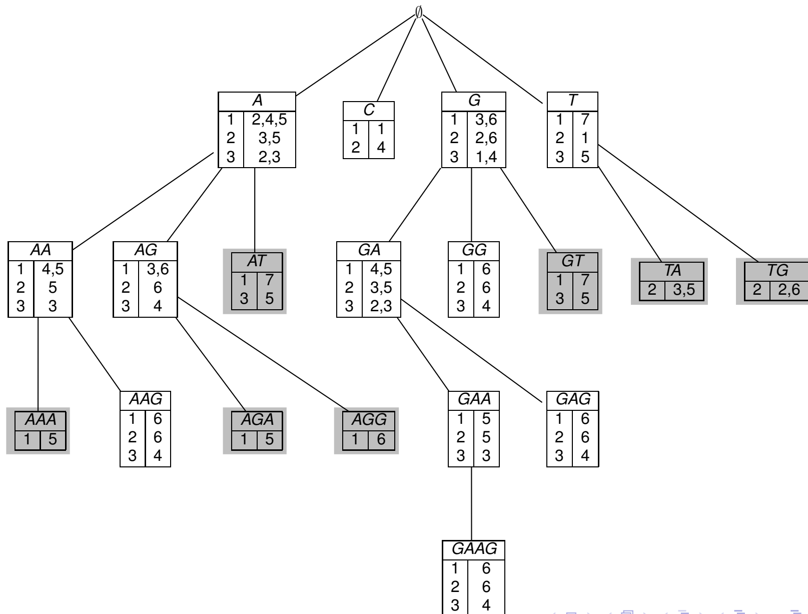
We keep track of positions only for the last symbol in the candidate sequence since we extend sequences from a common prefix, and so there is no need to keep track of all the occurrences of the symbols in the prefix.

We denote the sequential join as $\mathcal{L}(\mathbf{r}_{ab}) = \mathcal{L}(\mathbf{r}_a) \cap \mathcal{L}(\mathbf{r}_b)$.

Spade Algorithm

```
// Initial Call:  F ← ∅,  k ← 0,
    P ← {⟨s, L(s)⟩ | s ∈ Σ, sup(s) ≥ minsup}
SPADE (P, minsup, F, k):
1 foreach r_a ∈ P do
2     F ← F ∪ {(r_a, sup(r_a))}
3     P_a ← ∅
4     foreach r_b ∈ P do
5         r_ab = r_a + r_b[k]
6         L(r_ab) = L(r_a) ∩ L(r_b)
7         if sup(r_ab) ≥ minsup then
8             P_a ← P_a ∪ {⟨r_ab, L(r_ab)⟩}
9     if P_a ≠ ∅ then  SPADE (P, minsup, F, k + 1)
```

Sequence Mining via Spade



Projection-Based Sequence Mining: PrefixSpan

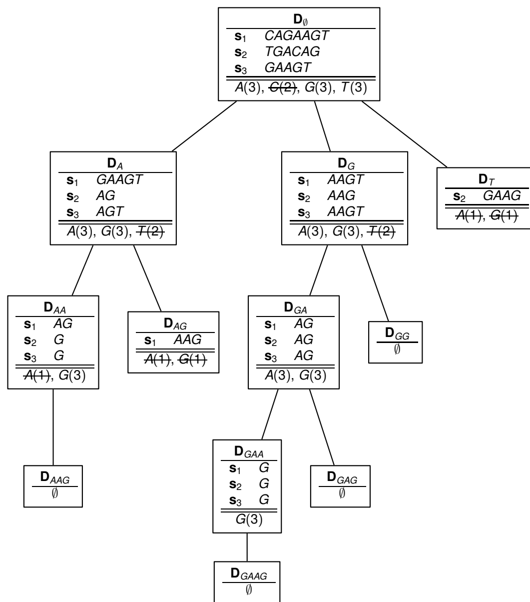
Let $\mathbf{D}$ denote a database, and let $s \in \Sigma$ be any symbol. The *projected database* with respect to s, denoted $\mathbf{D}_s$, is obtained by finding the the first occurrence of s in $\mathbf{s}_i$, say at position p. Next, we retain in $\mathbf{D}_s$ only the suffix of $\mathbf{s}_i$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

PrefixSpan computes the support for only the individual symbols in the projected database $\mathbf{D}_s$; it then performs recursive projections on the frequent symbols in a depth-first manner.

Given a frequent subsequence $\mathbf{r}$, let $\mathbf{D_r}$ be the projected dataset for $\mathbf{r}$. Initially $\mathbf{r}$ is empty and $\mathbf{D_r}$ is the entire input dataset $\mathbf{D}$. Given a database of (projected) sequences $\mathbf{D_r}$, 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 $\mathbf{D_r}$ on symbol s. A recursive call to PrefixSpan is then made with $\mathbf{r}_s$ and $\mathbf{D}_s$.

```
// Initial Call: D_r ← D, r ← ∅, F ← ∅
PREFIXSPAN (D_r, r, minsup, F):
1 foreach s ∈ Σ such that sup(s, D_r) ≥ minsup do
2     r_s = r + s // extend r by symbol s
3     F ← F ∪ {(r_s, sup(s, D_r))}
4     D_s ← ∅ // create projected data for symbol s
5     foreach s_i ∈ D_r do
6         s'_i ← projection of s_i w.r.t symbol s
7         Remove any infrequent symbols from s'_i
8         Add s'_i to D_s if s'_i ≠ ∅
9     if D_s ≠ ∅ then PREFIXSPAN (D_s, r_s, minsup, F)
```

Projection-based Sequence Mining: PrefixSpan



Substring Mining via Suffix Trees

Let $\mathbf{s}$ be a sequence having length n, then there are at most $O(n^2)$ possible distinct substrings contained in $\mathbf{s}$. 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 $\mathbf{D} = \{\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_N\}$ with N sequences.

We will show that all sequences can be mined in $O(Nn)$ time via Suffix Trees.

Suffix Tree

Given a sequence **s**, we append a terminal character $\$ \notin \Sigma$ so that $\mathbf{s} = s_1 s_2 \ldots s_n s_{n+1}$, where $s_{n+1} = \$$, and the jth suffix of **s** is given as $\mathbf{s}[j : n+1] = s_j s_{j+1} \ldots s_{n+1}$.

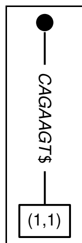
The *suffix tree* of the sequences in the database **D**, denoted $\mathcal{T}$, stores all the suffixes for each $\mathbf{s}_i \in \mathbf{D}$ in a tree structure, where suffixes that share a common prefix lie on the same path from the root of the tree.

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 $L(v_a, v_b)$.

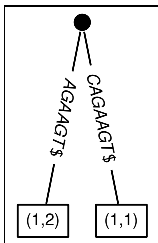
A suffix tree has two kinds of nodes: internal and leaf nodes. An internal node in the suffix tree (except for the root) has at least two children, where each edge label to a child begins with a different symbol. Because the terminal character is unique, there are as many leaves in the suffix tree as there are unique suffixes over all the sequences. Each leaf node corresponds to a suffix shared by one or more sequences in **D**.

Suffix Tree Construction for $\mathbf{s} = CAGAAGT\$$

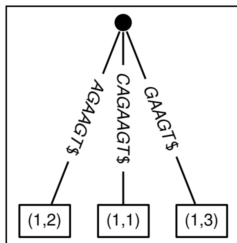
Insert each suffix j per step



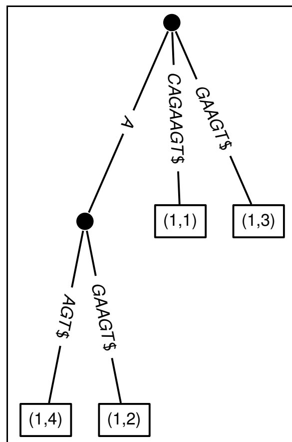
(a) $j = 1$



(b) $j = 2$



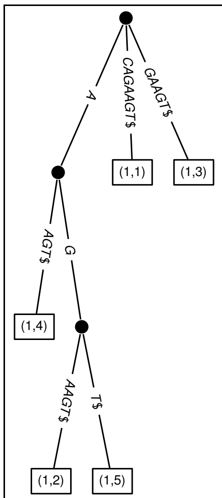
(c) $j = 3$



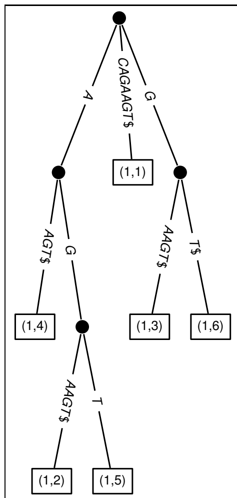
(d) $j = 4$

Suffix Tree Construction for $\mathbf{s} = CAGAAGT\$$

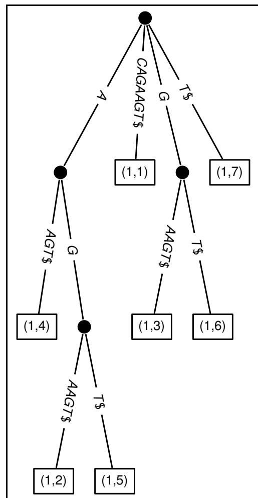
Insert each suffix j per step



(e) $j = 5$



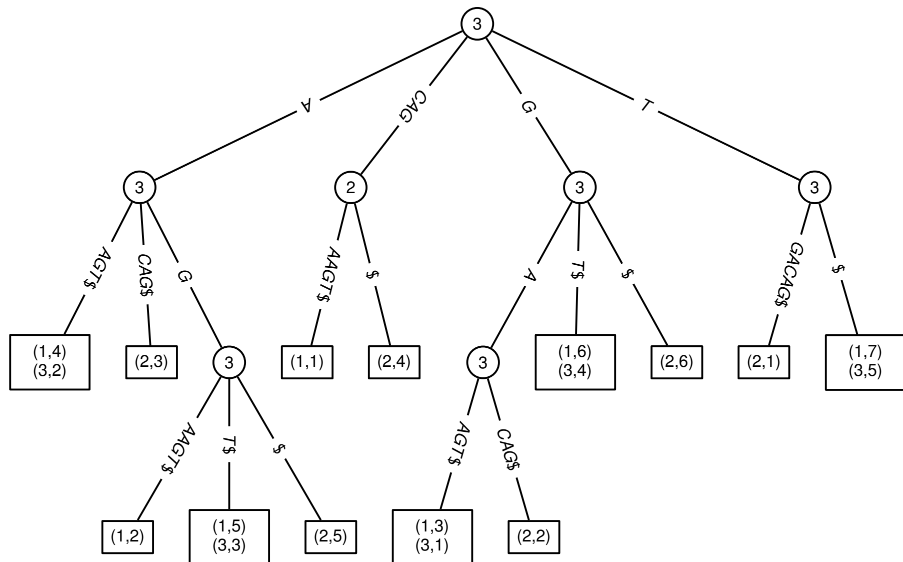
(f) $j = 6$



(g) $j = 7$

Suffix Tree for Entire Database

$\mathbf{D} = \{\mathbf{s}_1 = CAGAAGT, \mathbf{s}_2 = TGACAG, \mathbf{s}_3 = GAAGT\}$



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 *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.

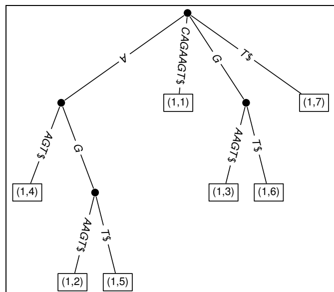
Because we have to match each character in $\mathbf{q}$, we immediately get $O(|\mathbf{q}|)$ as the time bound (assuming that $|\Sigma|$ is a constant), which is *independent* of the size of the database. Listing all the matches takes additional time, for a total time complexity of $O(|\mathbf{q}| + k)$, if there are k matches.

Ukkonen's Linear Time Suffix Tree Algorithm

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 $\mathbf{s}$. That is, if an edge label is given as $\mathbf{s}[i:j]$, then we represent is as the interval $[i, j]$.

Suffix Tree using Edge-compression: $\mathbf{s} = CAGAAGT\$$



(a) Full Tree

(b) Compressed Tree

Ukkonen Algorithm: Achieving Linear Time

Ukkonen's method is an *online* algorithm, that is, given a string $\mathbf{s} = s_1 s_2 \dots s_n\$$ it constructs the full suffix tree in phases.

Phase i builds the tree up to the i-th symbol in $\mathbf{s}$. Let $\mathcal{T}_i$ denote the suffix tree up to the ith prefix $\mathbf{s}[1:i]$, with $1 \leq i \leq n$. 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 $\mathbf{s}$.

However, this naive Ukkonen 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 $O(n^3)$. We will show that this time can be reduced to $O(n)$.

```
NAIVEUKKONEN (s):
1 n ← |s|
2 s[n+1] ← $ // append terminal character
3 T ← ∅ // add empty string as root
4 foreach i = 1,...,n+1 do // phase i - construct T_i
5     foreach j = 1,...,i do // extension j for phase i
          // Insert s[j : i] into the suffix tree
6         Find end of the path with label s[j : i-1] in T
7         Insert s_i at end of path;
8 return T
```

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

Ukkonen's Algorithm: Implicit Extensions

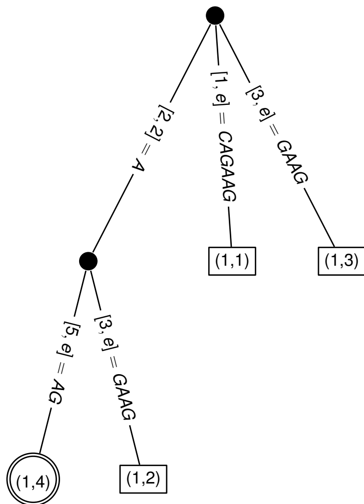
Let the current phase be i, and let $l \leq i-1$ be the last explicit suffix in the previous tree $\mathcal{T}_{i-1}$.

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 $i-1$.

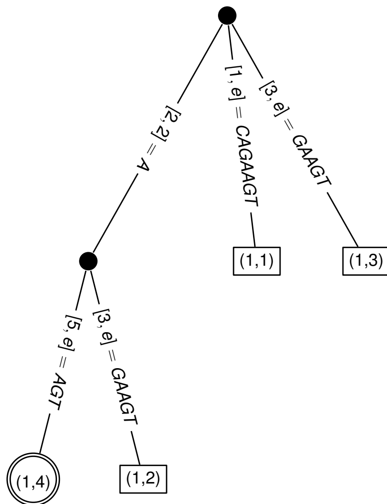
In the current phase i, we would have to extend these paths by adding s_i at the end. However, instead of explicitly incrementing all the ending positions, we can replace the ending position by a pointer e which keeps track of the current phase being processed.

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.

Implicit Extensions: $\mathbf{s} = CAGAAGT\$$, Phase $i = 7$



(a) $\mathcal{T}_6$



(b) $\mathcal{T}_7$, extensions $j = 1, \dots, 4$

Ukkonen's Algorithm: 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.

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 $\mathbf{s}[j : i-1]$.

Ukkonen's Algorithm: Suffix Links

We can avoid searching for the substring $\mathbf{s}[j:i-1]$ 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 $L(v_a)$.

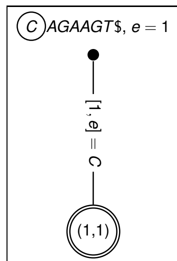
In extension $j-1$, let v_p denote the internal node under which we find $\mathbf{s}[j-1:i]$, and let m be the length of the node label of v_p. To insert the jth extension $\mathbf{s}[j:i]$, we follow the suffix link from v_p to another node, say v_s, and search for the remaining substring $\mathbf{s}[j+m-1:i-1]$ 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.

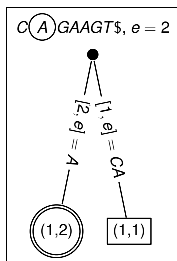
Linear Time Ukkonen Algorithm

```
UKKONEN (s):
1  n ← |s|
2  s[n+1] ← $ // append terminal character
3  T ← ∅ // add empty string as root
4  l ← 0 // last explicit suffix
5  foreach i = 1,...,n+1 do // phase i - construct T_i
6      e ← i // implicit extensions
7      foreach j = l+1,...,i do // extension j for phase i
           // Insert s[j : i] into the suffix tree
8          Find end of s[j : i−1] in T via skip/count and suffix links
9          if s_i ∈ 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 T
```

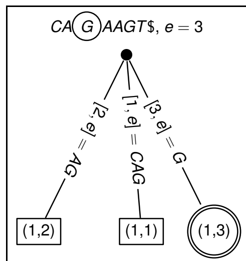
Ukkonen's Suffix Tree Construction: $\mathbf{s} = CAGAAGT\$$



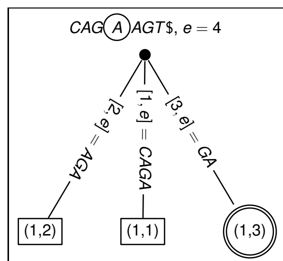
(a) $\mathcal{T}_1$



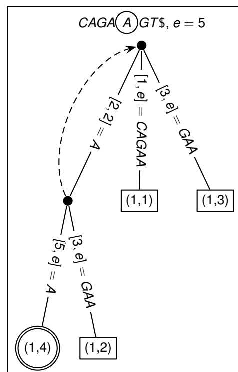
(b) $\mathcal{T}_2$



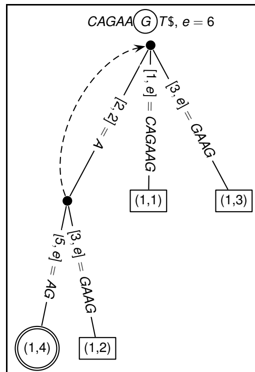
(c) $\mathcal{T}_3$



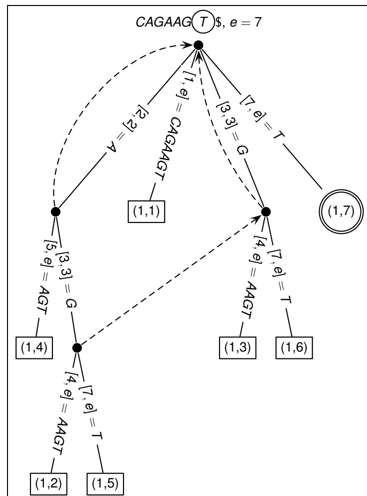
(d) $\mathcal{T}_4$



(e) $\mathcal{T}_5$

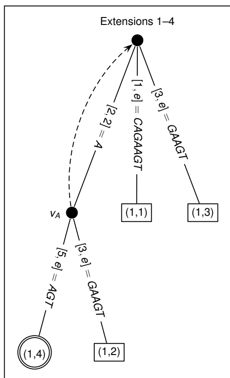


(f) $\mathcal{T}_6$

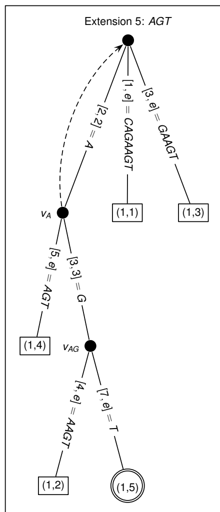


(g) $\mathcal{T}_7$

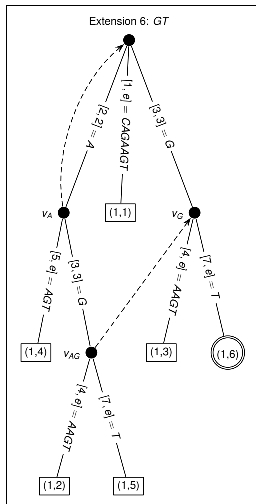
Extensions in Phase $i = 7$



(a)



(b)



(c)