

Chapter 7

Sequence Mining

7.1 Introduction

Many real world applications, such as in bioinformatics, web mining, text mining and so on, have to deal with sequential/temporal data. Sequence mining helps to discover frequent sequential patterns across time or positions in a given data set.

Let Σ denote an *alphabet*, defined as a finite set of characters or symbols, and let $|\Sigma|$ denote its cardinality. A *sequence* is defined as an ordered list of symbols, and is written as $\mathbf{s} = s_1s_2 \dots s_k$, where $s_i \in \Sigma$ is a symbol at position i . Here $|\mathbf{s}| = k$ denotes the *length* of the sequence. A sequence with length k is also called a *k-sequence*. Let Σ^* be the set of all possible sequences (or strings) that can be constructed using the symbols in Σ , including the empty string ϵ (which has length zero).

A sequence $\mathbf{s} = s_1s_2 \dots s_{n_s}$ is called a *subsequence* of another sequence $\mathbf{r} = r_1r_2 \dots r_{n_r}$, denoted as $\mathbf{s} \subseteq \mathbf{r}$, if for all $s_i \in \mathbf{s}$, there exists $r_{j_i} \in \mathbf{r}$, such that $s_i = r_{j_i}$, and the order of symbols is preserved, i.e., if $j_1 < j_2 < \dots < j_{n_s}$. In other words, the sequence \mathbf{s} is embedded in \mathbf{r} , but there might be intervening gaps between consecutive elements of \mathbf{s} in the embedding. If $\mathbf{s} \subseteq \mathbf{r}$, we also say that \mathbf{r} *contains* \mathbf{s} . \mathbf{s} is called a *substring* or *consecutive subsequence* of \mathbf{r} provided $s_1s_2 \dots s_{n_s} = r_jr_{j+1} \dots r_{j+n_s-1}$. In other words for substrings we do not allow any gaps between the elements of \mathbf{s} in the embedding. For example, let $\Sigma = \{A, C, G, T\}$, and let $\mathbf{s} = ACTGAACG$. Let $\mathbf{s}_1 = CGAAG$ and $\mathbf{s}_2 = CTGA$. Then \mathbf{s}_1 is a subsequence of \mathbf{s} , and \mathbf{s}_2 is a substring of \mathbf{s} . Define the *prefix* of a sequence \mathbf{s} as any substring of the form $s_1s_2 \dots s_i$, with $0 \leq i \leq n_s$, and define the *suffix* of \mathbf{s} as any substring of the form $s_i s_{i+1} \dots s_{n_s+1}$, with $1 \leq i \leq n_s + 1$. Note that for any sequence \mathbf{s} , we define $s_0 = \epsilon$ and $s_{n_s+1} = \epsilon$. For example, *ACT* is one of the prefixes of \mathbf{s} in the example above, and so is *ACTGA*, whereas *GAACG* is one of the suffixes of \mathbf{s} .

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

$$sup(\mathbf{r}) = |\{\mathbf{s}_i \in \mathcal{D} \mid \mathbf{r} \subseteq \mathbf{s}_i\}| \quad (7.1)$$

As in the case of itemsets, given a user-specified *minsup* threshold, we say that a sequence \mathbf{r} is *frequent* in database \mathcal{D} if $sup(\mathbf{r}) \geq minsup$. Furthermore, 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.

Below we will separately consider the tasks of mining all frequent subsequences, and all frequent substrings.

7.2 Mining Frequent Subsequences

id	sequence
s_1	CAGAAGT
s_2	TGACAG
s_3	GAAGT

Table 7.1: Example Sequence Database

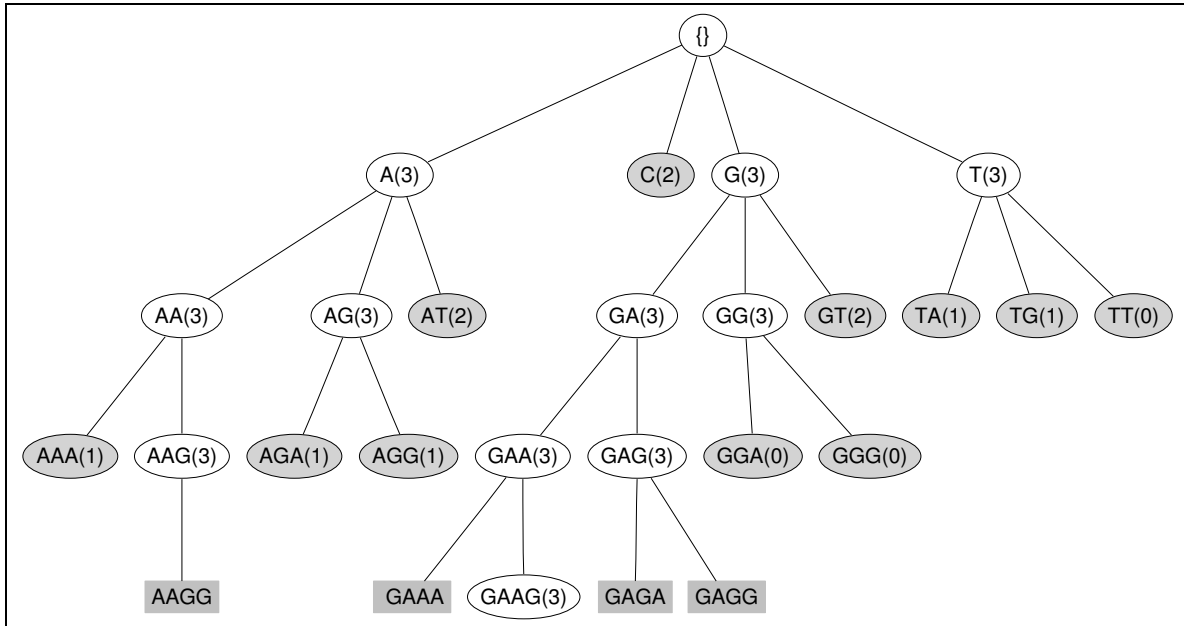


Figure 7.1: Sequence Search Space: Shaded ovals represent candidates that are infrequent, and the shaded boxes represent those that can be pruned based on an infrequent subsequence. All other subsequences are frequent.

Let $\Sigma = A, C, G, T$ and let the database consist of the three sequences shown in Table 7.1. We want to find all subsequences that occur in at least *minsup* sequences. The sequence search space can also be organized in a prefix search tree, as illustrated in Figure 7.1. Starting from the empty sequence, at the root at level 0, each node $s = s_1s_2 \cdots s_i$ at level i has children of the form $s' = s_1s_2 \cdots s_i s_{i+1}$ at level $i + 1$. In other words, s is a prefix of each s' , which are also called the *extensions* of s . For example, A has children AA , AG and AT . The key difference between itemset and sequence mining, in terms of the search space, is that for sequences, order of the symbols matters, and thus we have to consider all possible *permutations* of the symbols as the possible frequent candidates, whereas for itemsets, we only had to consider *combinations* of the items. The subsequence search space is conceptually infinite, since it consists of all sequences in Σ^* , i.e., all sequences of length zero or more that can be created using symbols in Σ . In practice, the database \mathcal{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 from length one through l , which gives the bound

$$|\Sigma|^1 + |\Sigma|^2 + \cdots + |\Sigma|^l = O(|\Sigma|^{l+1}) \quad (7.2)$$

since for any intermediate length j , there are $|\Sigma|^j$ possible subsequences of length j .

7.2.1 Level-Wise Mining

Like in the case of itemsets, we can devise an effective algorithm that searches the sequence prefix tree using a level-wise or breadth first search. Starting at level 0, at any intermediate level i , we generate all possible sequence extensions or *candidates* at level $i+1$. We will next compute the support of each candidate and prune those that are not frequent. The search stops when no more frequent extensions are possible.

For example, let us mine the database shown in Table 7.1 using $minsup = 3$. That is, we want to find only those subsequences that occur in all three database sequences. Figure 7.1 shows that we begin by extending the empty sequence ϵ at level 0, to obtain the candidates A , C , G and T . Out of these C can be pruned since 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 . One point worth emphasizing is that we can sometimes prune a candidate extension, even without counting. For example, consider the extension $GAAA$ obtained from GAA . We can see immediately that $GAAA$ has an infrequent subsequence AAA , which implies that $GAAA$ cannot possibly be frequent. We thus prune it without counting. The figure also shows other cases where we can prune based on an infrequent subsequence.

In terms of the I/O complexity of the level-wise search, since we compute the support for an entire level in one scan of the database, the total database access cost is $O(l \cdot \mathcal{D})$, where l is the length of the longest frequent sequence. The computational complexity is $O(|\Sigma|^{l+1})$, since we had to enumerate all the subsequences of longest sequence of length l , giving the number of candidates as: $\sum_{i=0}^l |\Sigma|^i = |\Sigma|^{l+1}$.

7.2.2 Vertical Sequence Mining

Here we consider a vertical database representation for sequence mining. The idea is to record for each symbol, the sequences and the positions, where it occurs. For each symbol $x \in \Sigma$, we keep a set of tuples of the form $\langle i, pos(x) \rangle$, where $pos(x)$ is the set of positions in s_i where symbol x appears. Let $\mathcal{L}(x)$ denote the set of such sequence-position tuples, which we refer to as the *enhanced tidlist* (or just tidlist). For example, in Table 7.1 A occurs in s_1 at positions 2, 4 and 5. Thus we add the tuple $\langle 1, \{2, 4, 5\} \rangle$ to $\mathcal{L}(x)$. Figure 7.2 shows the enhanced tidlists for each symbol.

To enumerate all the frequent sequences, the idea is to perform recursive sequential joins on the tidlists to compute the support. For example, to obtain $\mathcal{L}(AG)$, we perform a sequential join over the tidlists $\mathcal{L}(A)$ and $\mathcal{L}(G)$ as follows: Let $\langle i, pos(A) \rangle \in \mathcal{L}(A)$, and let $\langle j, pos(G) \rangle \in \mathcal{L}(G)$, such that $i = j$, which requires that both symbols appear in the same sequence. Next for each $p \in pos(G)$, if there exists a position $q \in pos(A)$, such that $q < p$, then this means that A occurs before the G , and we can add p to the set of positions for s_i in $\mathcal{L}(AG)$. For example, for the tuples $\langle 1, \{2, 4, 5\} \rangle \in \mathcal{L}(A)$ and $\langle 1, \{3, 6\} \rangle \in \mathcal{L}(G)$, both the positions 3 and 6, for G , occur after some occurrence of A , e.g., at position 2. Thus we add the tuple $\langle 1, \{3, 6\} \rangle$ to $\mathcal{L}(AG)$. 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. Figure 7.2 shows the complete working of the algorithm, along with all the candidates and their tidlists. 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.

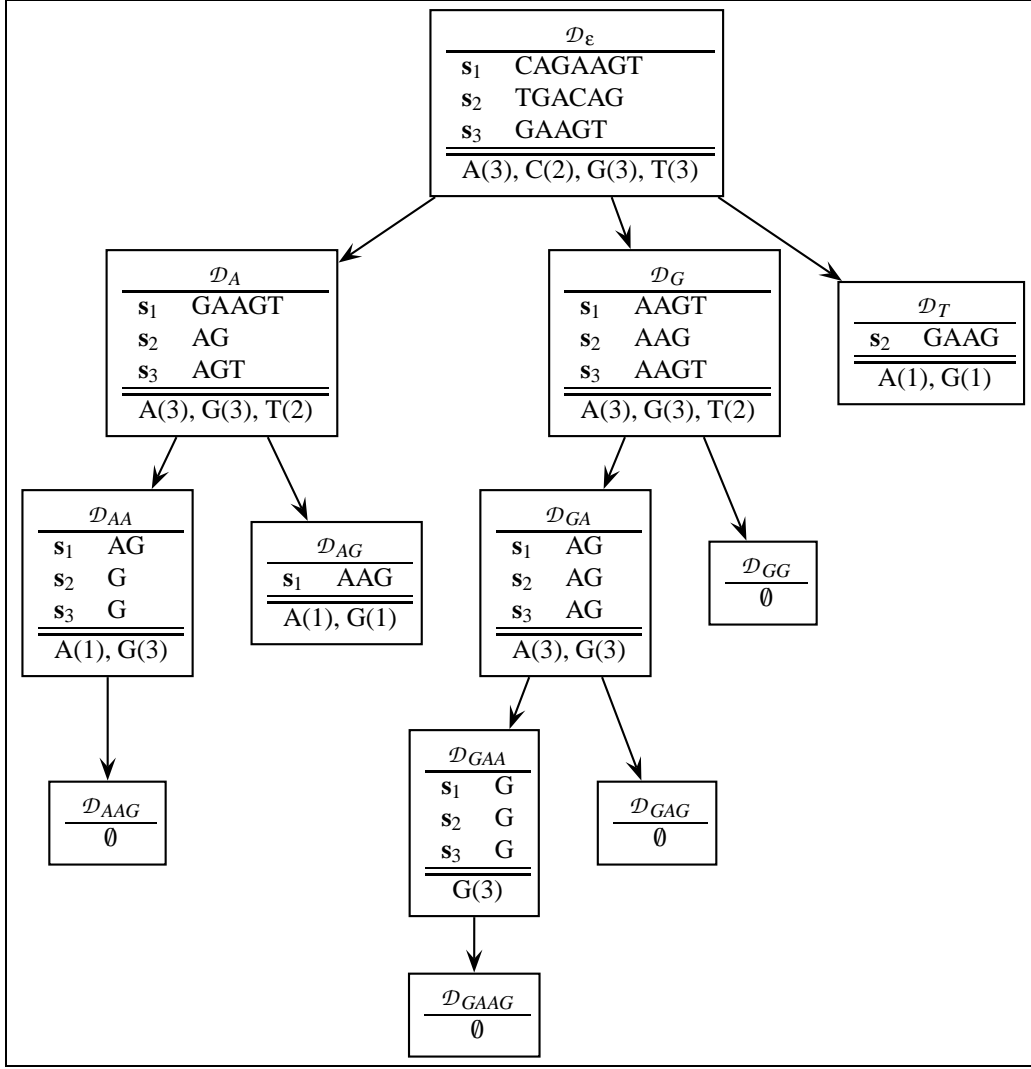


Figure 7.3: Projection-based Sequence Mining

length of a substring. Then there are at most $O(l^2)$ possible distinct substrings contained in s , which can be obtained by varying w from one to l . For a given substring length w , there are $l - w + 1$ possible substrings in s . Adding over all substring lengths we get:

$$\sum_{w=1}^l (l - w + 1) = l + (l - 1) + \dots + 2 + 1 = O(l^2) \quad (7.3)$$

This is a much smaller search space compared to subsequences, and consequently we desire more efficient algorithms for solving the frequent substring mining task.

Let Σ denote the alphabet, and let $\$ \notin \Sigma$ be the *terminal* character, used to mark the end of a string. Given a sequence $\mathbf{s} = s_1 s_2 \dots s_{n_s}$, the i^{th} suffix of \mathbf{s} is represented as $\mathbf{s}[i : n_s] = s_i s_{i+1} \dots s_{n_s}$. For convenience, we append the terminal character to the string, and refer to it by s_{n_s+1} . The suffix tree of the sequences in the database \mathcal{D} , denoted as $T_{\mathcal{D}}$, stores all the suffixes for each $\mathbf{s}_i \in \mathcal{D}$, 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

characters 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_i, v_j) , is called the *edge label*, and is denoted as $L(v_i, v_j)$. A suffix tree has two kinds of nodes: internal and leaf nodes. An internal node in the suffix tree, except the root, has at least two children, where each edge label to a child begins with a different character. Since 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 thus corresponds to a unique suffix shared by one or more sequences.

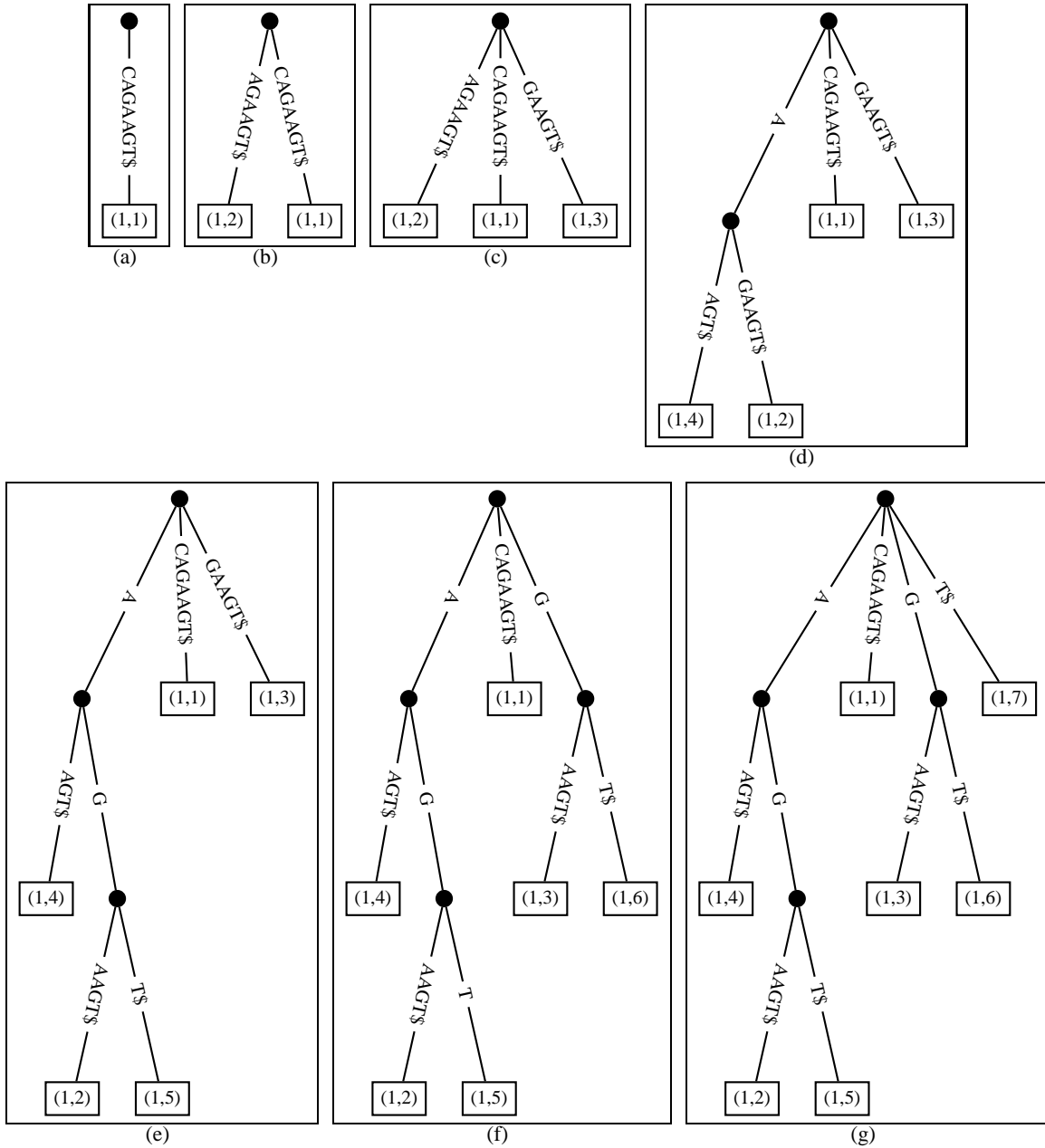


Figure 7.4: Suffix Tree Construction: (a)-(g) shows the successive changes to the tree, after we add each new suffix of s_1 from suffix 1 to 7.

7.3.1 Naive Algorithm

It is straightforward to obtain a quadratic time and space suffix tree construction algorithm. Initially, the suffix tree T_D is empty. Then for each sequence $s_i \in \mathcal{D}$, we generate all its suffixes $s_i[j : n_{s_i}]$, with $1 \leq j \leq n_{s_i}$, and insert it 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 characters (which can happen on an edge). If we reach a leaf, we insert the pair (i, j) into the leaf, noting that this is the j -th suffix of sequence s_i . If there is a mismatch in one of the symbols, say at position $p \geq j$, we add a new edge labeled $s_i[p : n_{s_i}]$ leading to a new leaf node, and then add (i, j) to the leaf node. As an example let us consider the database in Table 7.1. First we focus on $s_1 = CAGAAGT$. Figure 7.4 shows what the suffix tree looks like after inserting the j -th suffix of s_1 into T_D . The complete suffix tree for s_1 is shown in Figure 7.4(g), and the complete suffix tree for all three sequences is shown in Figure 7.5.

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

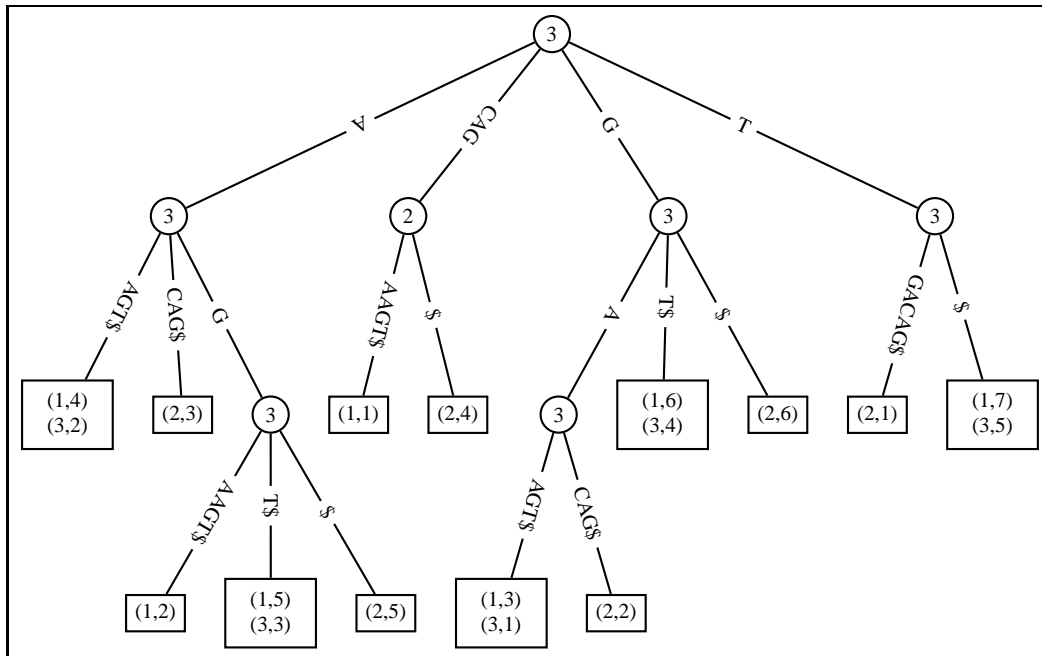


Figure 7.5: Suffix tree for all three sequences in \mathcal{D} . Internal nodes store the support of the substrings

Frequent Substrings Once the suffix tree is built, we can compute all the frequent substrings by checking how many different sequences appear in the leaf nodes under an internal node, i.e., we compute the support for each internal as well as leaf node. Those that have support at least *minsup* are frequent substrings. For example, if *minsup* = 3, then the frequent substrings are A, AG, G, GA and T. Out of these the maximal ones are AG and GA. If *minsup* = 2, then we will find that the maximal frequent substrings are GAAGT and CAG. The suffix tree can also support ad-hoc queries for finding all the occurrences in the database for any

query substring q . For example, if $q = GAA$, then we would search for each character of q starting from the root of the tree shown in Figure 7.5. In this case we will find the query, since all characters match, and we report the occurrences $(1, 3)$ and $(3, 1)$, which means that GAA appears at position 3 in s_1 and at position 1 in s_3 . If there is a mismatch in any character, that means the query does not appear as a substring in the database.

In terms of the query time complexity, since 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.

7.3.2 Ukkonen's Linear Time Algorithm

We now present a linear time and space algorithm for constructing suffix trees. We will first consider how to build the suffix tree for a single sequence $\mathbf{s} = s_1s_2 \cdots s_{n_s}$.

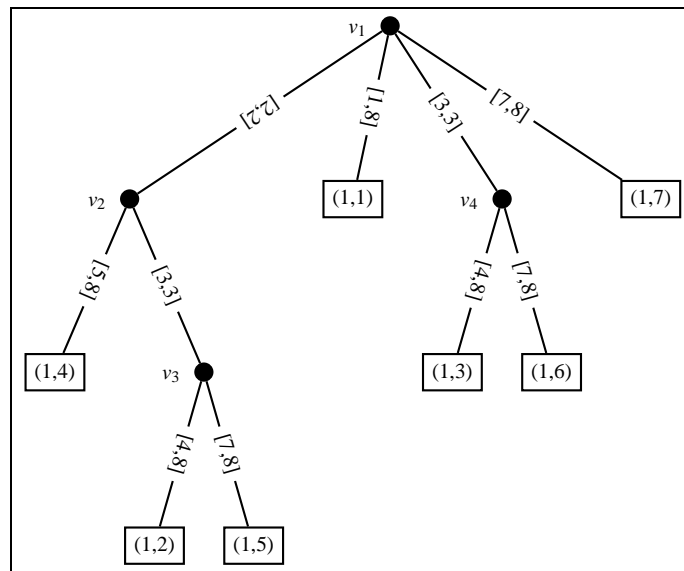


Figure 7.6: Suffix Tree for s_1 Using Edge-Compression

Achieving Linear Space Let us first see how we can reduce the space requirements of a suffix tree. If an algorithm has to write all the characters on each edge label, as shown in Figure 7.4(g) for the sequence $s_1 = CAGAAGT\$$, then we will not be able to 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 only store the starting and ending positions of the edge label. For example, consider the edge label $CAGAAGT\$$ for the suffix $(1, 1)$ in Figure 7.4(g). We can compress this as $[1, 8]$, since the edge label denotes the substring $s_1[1 : 8]$. Likewise, consider the edge label $AAGT\$$ leading to suffix $(1, 2)$, can be compressed as $[4, 8]$, since $AAGT\$ = s_1[4 : 8]$. The complete suffix tree for s_1 with all compressed edges is shown in Figure 7.6.

In terms of the space complexity, note that in the naive algorithm, when we add a new suffix to the tree, it can create at most one new internal node. Thus across all the suffixes, only $O(n_s)$ internal nodes are created. Since we store only two indexes to represent each edge, we get a total space complexity of $O(n_s)$.

Achieving Linear Time Given a string $\mathbf{s} = s_1s_2 \cdots s_{n_s}\$$. Let T_{i-1} denote the suffix tree up to the $(i-1)$ -th prefix $\mathbf{s}[1 : i-1]$, with $1 \leq i \leq n_s+1$. Ukkonen's linear time algorithm works in phases by constructing 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, we have to insert all the suffixes $\mathbf{s}[j : i]$ 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 T_{n_s+1} we obtain the complete suffix tree T_s for \mathbf{s} . At first glance, this method has cubic time complexity since to obtain T_i from T_{i-1} takes $O(i^2)$ time, with the last phase requiring $O(n_s^2)$ time. Since there are n_s phases, the total time is $O(n_s^3)$. Our goal is to show that this time can be reduced to just $O(n_s)$ via the optimizations described below.

- **Suffix Links:** This optimization avoids searching from the root, when inserting each suffix, via the use of *suffix links*. Given an internal node v , we will maintain a *suffix link* to the internal node w , where $L(w)$ is the immediate suffix of $L(v)$. For example, if $L(v) = ACGT$, then this node will point to the node w , with $L(w) = CGT$. Either w already exists in the suffix tree T_i or it will be created in the very next extension, so we can add the suffix link at that time.
- **Skip/count Trick:** Assume that in the phase i , we have just processed the $(j-1)$ -th extension, which ensures that the suffix $s_{j-1}s_js_{j+1} \cdots s_{i-1}s_i$ has been added to the tree T_i . When processing the j -th extension, $s_js_{j+1} \cdots s_{i-1}s_i$, note that the string $\gamma = s_js_{j+1} \cdots s_{i-1}$ must already exist in the tree (since it would have been created in the previous phase T_{i-1}). Thus instead of searching for each character in γ all over again starting from the root, we *count* the edge length, say m , for the edge beginning with character s_j , and then *skip* directly to the child node, say v_c , and search for the remaining string $\gamma' = s_{j+m} \cdots s_{i-1}$ from v_c using the same count/skip technique. If $m > \gamma'$, then γ ends inside the edge labeled $[s, e]$, and we can then insert the j -th suffix at position $s + |\gamma'|$. With the skip/count technique, the cost of an extension becomes proportional to the number of nodes on the path, as opposed to the number of characters.
- **Implicit Suffixes:** This optimization states that in extension $j-1$ of phase i , if $s_{j-1}s_js_{j+1} \cdots s_{i-1}s_i$ is found in the tree, then all subsequent extensions will also be found in the tree, and thus there is no need to process further extensions. In other words, these suffixes already exist in the tree implicitly, and will become explicit the first time we encounter a new substring that does not already exist in the tree. This will surely happen once we process the terminal character $\$$, since $\$ \notin \Sigma$.
- **Implicit Extensions:** With the implicit suffixes optimization, when we are in the i -th phase, not all suffixes up to s_i may be in the tree explicitly. Let the last explicit suffix be for phase k , with $k < i$, which means that all subsequent phases resulted in implicit suffixes. Note that all explicit suffixes in the tree have edge labels of the form $[p, i-1]$ leading to the leaf nodes, at the start of the i -th phase. Instead of explicitly incrementing all the ending positions, we replace the ending position by a pointer e which keeps track of the current phase or the current position being processed. Thus we replace $[p, i-1]$ with $[p, e]$. At phase i , we set $e = i$, and immediately all the existing suffixes get implicitly extended to $[p, i]$.

Let us look at the execution of the algorithm on the sequence $\mathbf{s}_1 = CAGAAGT\$$, as shown in Figure 7.7. In phase 1, we process character $s_1 = C$ and insert the suffix $(1, 1)$ into the tree with edge label $[1, e]$. In phases, 2 and 3, new suffixes $(1, 2)$ and $(1, 3)$ are added. For phase 4, when we want to process $s_4 = A$, we note that all suffixes up to 3 are already explicit. Setting $e = 4$ implicitly extends all of them, so we only have to make sure that the last extension ($j = 4$) consisting of the single character A is in the tree. Searching

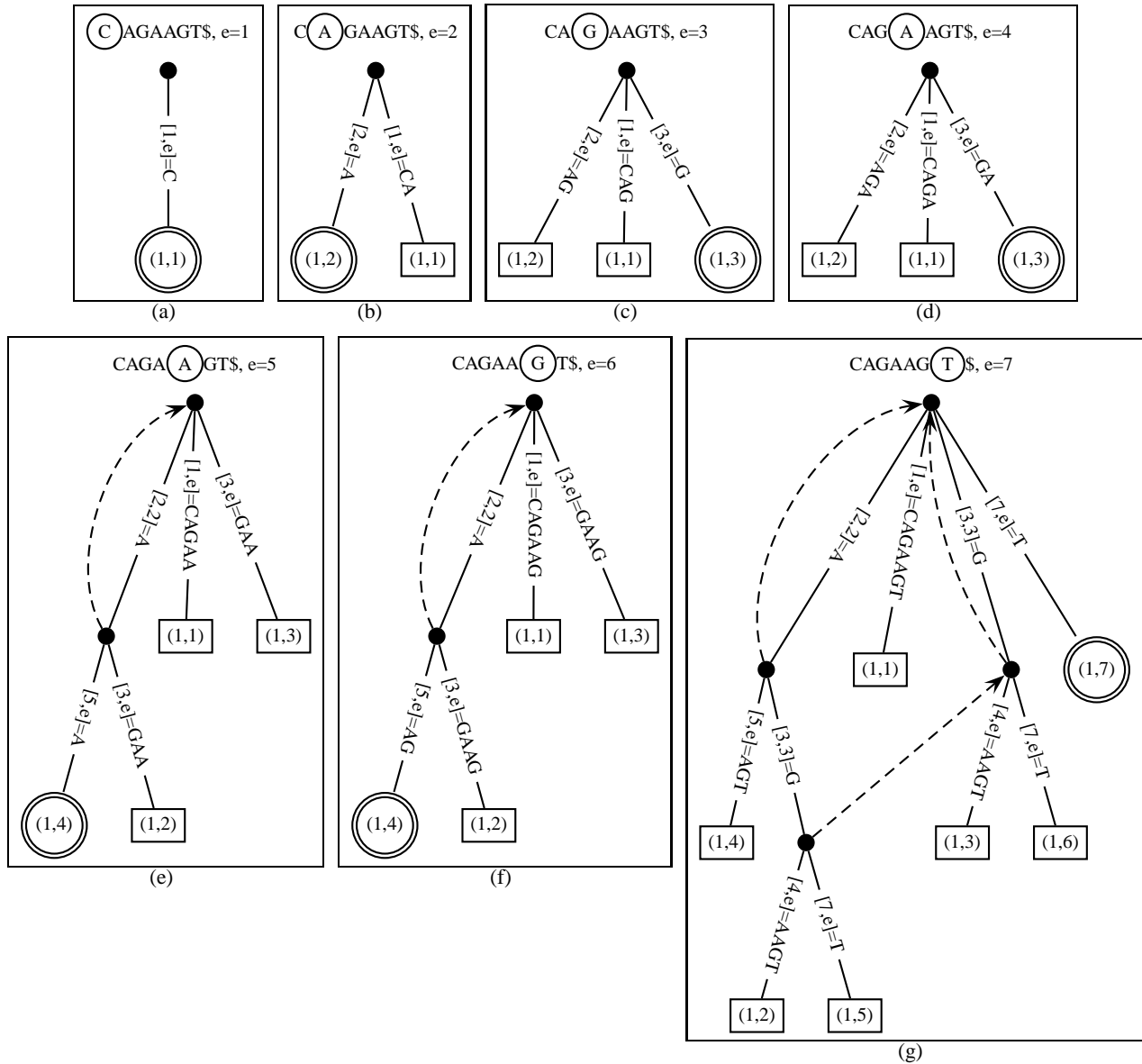


Figure 7.7: Ukkonen's Linear Time Algorithm for Suffix Tree Construction. (a)-(g) shows the successive changes to the tree after the i -th phase. The suffix links are shown with dashed arrows. The double-circled leaf denotes the last explicit suffix in the tree. The last step is not shown, since when $e = 8$, the terminal character \$ will not alter the tree. All the edge labels are shown for ease of understanding, though the actual suffix tree only keeps the $[s, e]$ indices for each edge.

from the root, we find A in the tree implicitly, and we thus proceed to the next phase. In the next phase, when $e = 5$, suffix (1,4) will become explicit, when we try to add the extension AA, and find that it 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). Since for $e = 7$, T is a previously unseen symbol, all suffixes will become explicit, as shown in Figure 7.7(g).

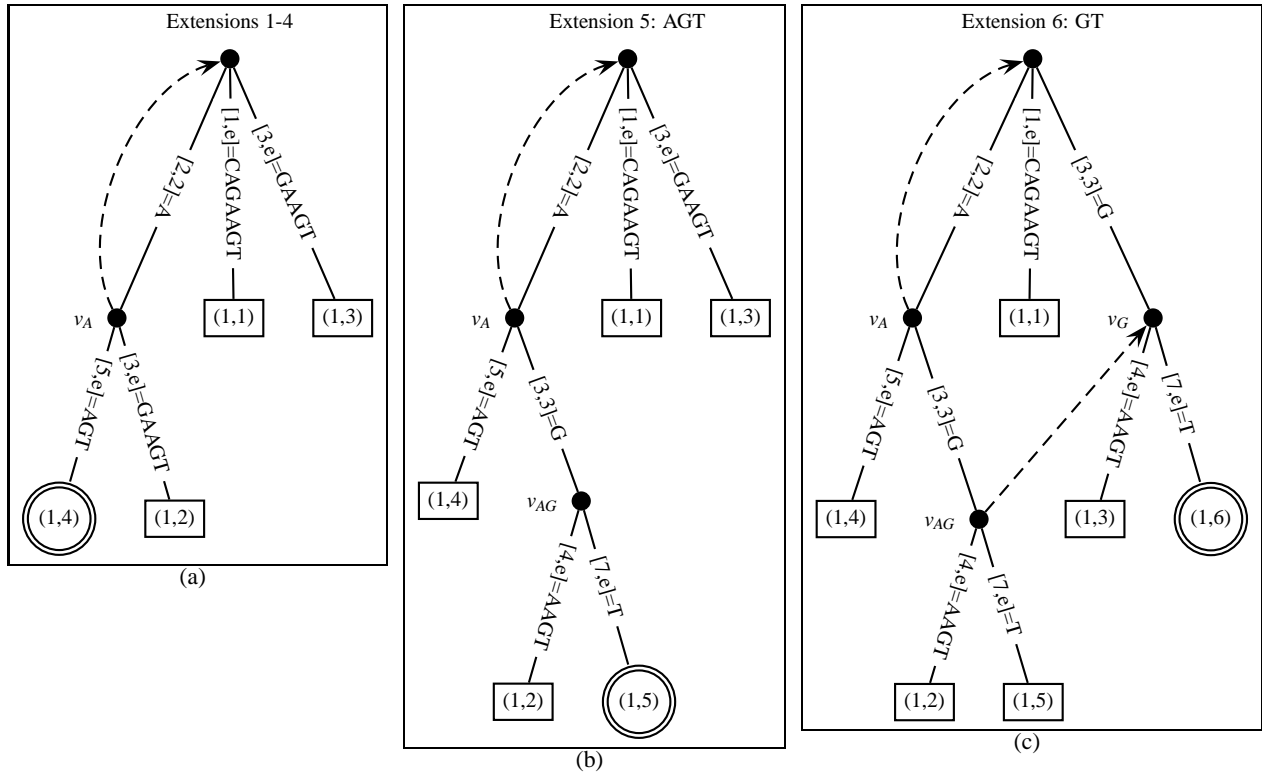


Figure 7.8: Extensions in the Last Phase

It is instructive to see the extensions of the last phase in more detail, as shown in Figure 7.8. At the start of the 7-th phase, we set $e = 7$, which makes implicit extensions for all suffixes explicitly in the tree, i.e., up to suffix 4, as shown in Figure Figure 7.8(a). Next we have to execute the following extensions:

CAG(A)AG(T)\$	extensions 1-4 implicitly done
AG(T)\$	extension 5
G(T)\$	extension 6
(T)\$	extension 7

For extension 5, we begin at the last explicit leaf, and follows its parent's suffix link, and begin searching for the remaining characters from that point. In our example, the suffix link point to the root, so we search for *AGT* from the root. We skip to node v_A , and look for the remaining string *GT*, which has the first mismatch at position 4 inside the edge $[3, e]$. At this point we create a new internal node after *G*, and insert the explicit suffix (1,5), as shown in Figure Figure 7.8(b). The next extension *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 at position 4 on the edge out of the root to leaf (1,3). We then create a new internal node at that point, v_G , add a link from the previous internal node v_{AG} to v_G , and add a new explicit leaf (1,6), as shown in Figure 7.8(c). After processing the last extension, the tree will be the same as that shown in Figure 7.7(g). Once s_1 has been processed, we can then proceed with the remaining sequences in the database \mathcal{D} . The final suffix tree for all three sequences will be as shown in Figure 7.5, with additional suffix links from all the internal nodes.

In terms of the time complexity of the algorithms, it is important to note that we achieve linear time only once all the optimizations are done in conjunction. We will only sketch a proof of why the algorithm has time $O(l)$ for a sequence of length l , and thus the total time over the entire database of n sequences will be $O(n \cdot l)$, if l is the longest sequence length. We will show that the algorithm does only a constant amount of work 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 k_1 are implicitly done. For the remaining extensions, we stop the first time some suffix is implicitly in the tree, let's call that position k_2 . Thus phase i needs to add explicit suffixes only for suffixes $k_1 + 1$ through $k_2 - 1$. For creating each implicit suffix, we do a constant number of operations, which involve following the closest suffix link, skip/counting to look for the first mismatch, and inserting a new suffix leaf node. Since each leaf becomes explicit only once, and the number of skip/count steps are bounded by $O(l)$ over the whole tree, we get a worst-case $O(l)$ time algorithm.