CHAPTER 4

THE WONDERS OF TEXT INDEXING STRUCTURES

"That's the secret to life:
replace one worry with another."
CHARLIE BROWN

Forget for a moment the problem of finding approximate patterns outlined in the previous chapter, and reconsider the simpler problem of finding exact occurrences of a pattern in a string. There are scores of algorithms for this problem. The "classical" ones, like Knuth–Morris–Pratt and Boyer–Moore algorithms [54, 18, 8], usually align the left end of the pattern with the left end of the string, and compare the characters of the pattern with the corresponding ones of the string until either all the characters of the pattern are exhausted (the pattern has been found), or a mismatch is encountered. In either case, the pattern is then shifted to the right by one or more positions, and the process is repeated until the right end of the string is reached. The trick is to shift the pattern by as many positions as possible, so as to reduce the number of character comparisons between the pattern and the string. These algorithms are quite efficient, having a theoretical time complexity of $O(n + m)$, where n is the length of the string and m is the length of the pattern. Term n in the sum comes from the character comparisons, while m comes from a pre–processing of the pattern needed to compute the longest possible shifts at each position.

However, suppose that you have a large number of candidate patterns, say, a few thousands, and the text is very long, perhaps a whole genome of many megabytes, or even a whole database of genomic sequences. Having to scan the whole string for each pattern, although efficiently, perhaps is

not the best solution.

While working on this manuscript, I often had to consult a LaTeX manual [29] for explanations about some command. But I did not scan each time the whole book looking for the command I was interested in: I just searched for it in the alphabetically ordered analytical index, that told me in which pages that command could be found. Now, can we do something similar for strings, and, in our case, for our genomic and protein sequences? In other words, given a string or a set of strings, can we build an index of all the patterns, or the substrings, that appear in them, in order to simplify the task of finding a pattern? And, more important, can we do it *efficiently*?

The best pattern matching algorithms take $O(n+m)$ time, and it would be great to match this result. So, at our disposal we have $O(n)$ time for building the index, and $O(m)$ time for finding a pattern (or vice versa, but it sounds a bit unfeasible). On the other hand, there are $O(n^2)$ substrings in a string of length n, a quite discouraging fact. And, furthermore, we'd expect our indexing structure not to take too much space.

But, as we will see in a few moments, such structures actually *do* exist, and have been around the block for quite a while: they take linear time to be constructed and need linear space [25].

4.1 SUFFIX TREES

Let Σ be a nonempty finite alphabet, and Σ^* the set of strings over Σ. If $S = s_1 \dots s_n$ is a string in Σ^*, we will denote by $|S|$ its length, and $S[i..j]$ its *substring* (*factor*) $s_i \dots s_j$. If $S = \alpha\beta\gamma$, with $\alpha, \beta, \gamma \in \Sigma^*$, then α is a *prefix* of S, and γ is a *suffix* of S. With $Suf(S)$ we will denote the set of all suffixes of S.

The most widely used and known text indexing structure is perhaps the *suffix tree*, first introduced in [103] in 1973. At the beginning, it did not have a great success in the computer science community, and received far less attention than deserved. This was probably because the first papers that introduced it were regarded as kind of obscure, and difficult to understand[1]. The revenge of the suffix tree came with the advent of bioinformatics, and the string problems that the new discipline brought along. For some problems, like the longest common substring problem introduced in Chapter 3, the most efficient solutions can be obtained only with the preliminary construction of the suffix tree or analogous structures.

Here it is:

[1] Well, for this matter many computer scientists admit they've never been able to fully understand where the $O(n+m)$ time bound for classical pattern matching algorithms comes from.

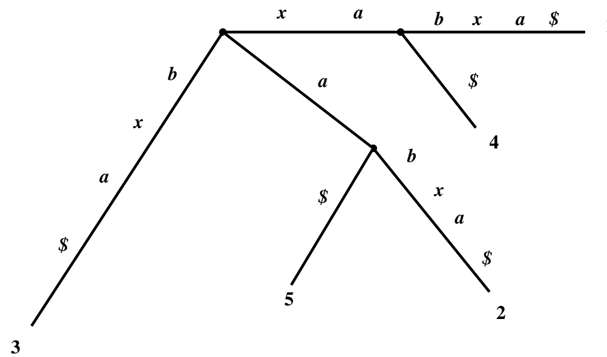

Figure 4.1: Suffix tree for string $xabxa$. Character \$ is used as a marker.

Definition 1 *A suffix tree $\mathcal{T}$ for a string $S = s_1 \ldots s_n$ of length n is a rooted directed tree such that:*

1. *It has n leaves numbered from 1 to n.*
2. *Each internal node other than the root has at least two children nodes.*
3. *Each edge is labeled with a non–empty substring of S.*
4. *The labels of two edges leaving the same node cannot begin with the same character.*
5. *For any leaf i, the concatenation of the edge labels on the path from the root to leaf i exactly spells out the* suffix *of S starting at position i, that is, it spells out $S[i..n]$.*

This definition of suffix trees, however, does not guarantee that a suffix tree can be built for any string S. The reason is that if one suffix of S matches a prefix of another suffix of S, then no suffix tree respecting the conditions given in the above definition can exist, since the path for the first suffix would not end up at a leaf. For example, in string $axabxa$, suffix xa is the prefix of suffix $xabxa$, and the path spelling it cannot end up at a leaf. The existence of a suffix tree for a given string S is however guaranteed when the last character of S does not appear elsewhere in the string. Thus, this limitation can be overcome with a simple trick. It suffices to append to S a character (that we will call *marker* character) that *does not* belong to the string alphabet Σ (see Fig. 4.1). Since, given any position i of the string, there exists a path in the tree spelling out $S[i..n]$, we have that every substring starting at i is found on the path from the root to leaf i.

Now, let us first consider the space complexity of suffix trees. How is it possible to compact $O(n^2)$ substrings in a structure that takes linear space?

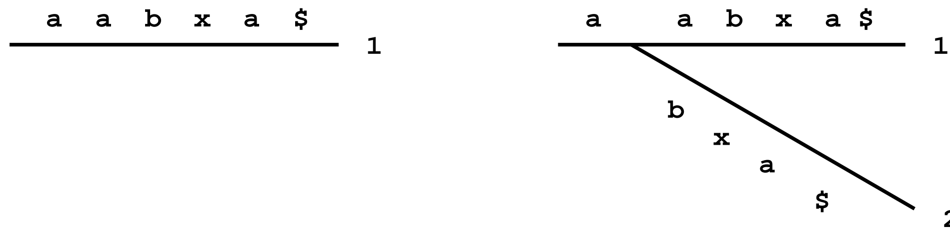

Figure 4.2: The first two steps in the naïve construction of the suffix tree for string $aabxa\$$. First suffix $aabxa\$$ is put in the tree (left). Then the characters of suffix $abxa\$$ are matched along the edge. A mismatch is encountered after the first a. A new internal node is created at that point, the edge is broken accordingly, and a new edge leading to leaf 2 is created, with label corresponding to the unmatched part of the suffix (right).

4.1.1 SPACE COMPLEXITY OF SUFFIX TREES

A suffix tree for a n character string S has exactly n leaves. Since each internal node must have at least two children, there are at most $n/2$ ancestor nodes for the leaves, and so on. Thus, the overall number of nodes is at most $2n$, that is, $O(n)$. Another issue are the edge labels. All in all, they might require a quadratic number of characters to be represented. Consider string $S = qwertyuiopasdfghjklzxcvbnm$. Each suffix begins with a different character, thus there are 26 edges leaving the root, and each one is labeled with a different suffix, requiring in all $(26 \times 27)/2 = O(n^2)$ characters. This problem, however, can be overcome with another trick: the edge labels are substrings of the input string. Thus, we can keep the input string in main memory, and label each edge with a pair of integers, denoting the initial and final position in the input string of the substring labeling it. Since there are $O(n)$ edges in the tree (each edge enters a node, and there are $O(n)$ nodes), the result is that we need linear space to label the edges. There is a last thing that must be taken into account. Although many textbooks report suffix trees as needing $O(n)$ space, each node of the tree can have at most $|\Sigma|$ children. If we implement the tree by storing $|\Sigma|$ pointers in each node, one for each possible child, the actual space complexity is $O(|\Sigma|n)$, that is, depends also on the size of the input alphabet. We will return to this point in Section 4.1.5.

4.1.2 LINEAR TIME CONSTRUCTION OF SUFFIX TREES

The naïvest method for constructing a suffix tree takes quadratic time. First, a single edge for suffix $S[1..n]\$$ (the whole string plus the marker) is put

into the tree (that at this point consists of the root connected to leaf 1 with an edge labeled $S[1..n]\$$), then all the other suffixes $S[i..n]\$$ are added, with i increasing from 2 to n. To insert suffix $S[i..n]\$$, the characters of the suffix are matched to characters on the tree edges, starting from the root, until no further matches are possible. The matching path is unique, because two edges leaving the same node cannot have labels beginning with the same character. Once a mismatch has been encountered, we have two cases: we have reached an internal node p, or we are along an edge. In the former case, we create leaf i, and connect p with it with an edge labeled with the remaining part of $S[i..n]\$$ (all the characters that have not yet successfully been matched in the tree). If we ended along an edge, we create a new node p breaking the edge in two just after the last character of the edge that was successfully matched to a character of $S[i..n]\$$. Also in this case, we create a new leaf i, as well as an edge from p to it labeled with the remaining part of $S[i..n]\$$ (see Fig. 4.2). In theory, the maximum number of character comparisons that has to be done is $\sum_{i=1}^{n} i = O(n^2)$.

But, quite amazingly, the suffix tree for a string can be constructed in time *linear* on the length of the string, as first introduced in [103], and later on, for example, in [68]. However, in this work we focus our attention in the linear time *on line* algorithm introduced by Esko Ukkonen [98]. On line here means that the algorithm processes the characters of the string one at a time, from left to right, with no need to know the whole string beforehand. This feature can be useful in some cases. Here, I present it following the lines of [36], an explanation that gave me a deeper understanding of the power of suffix trees and the methods for their construction. To present the algorithm we first have to provide a few definitions.

Definition 2 *Given a node p of a suffix tree, we call* path label *of p the concatenation of the edge labels of the path from the root to p in the tree.*

Definition 3 *An* implicit *suffix tree for string S is a tree obtained from the suffix tree for S by removing every occurrence of the marker character from the edge labels, then removing every edge with no label, then removing every node with just one outgoing edge.*

In other words, the implicit suffix tree for string S is the one we would obtain without appending the marker character at the end of the string. Recall that every suffix of S is spelled out by a path in the implicit tree, but the path does not necessarily end at a leaf.

Given an alphabet Σ, an input string $S = s_1 \dots s_n \in \Sigma^*$, and a marker $\$ \notin \Sigma$, Ukkonen's algorithm builds the suffix tree for $S\$$ in n *phases*, constructing at phase i the implicit suffix tree $\mathcal{T}_i$ for prefix $S[1..i]$ of S, starting

from the implicit suffix tree $\mathcal{T}_{i-1}$ for $S[1..i-1]$ built at the previous phase. Each phase $i+1$ is in turn divided into $i+1$ *extensions*, one for each of the $i+1$ suffixes of $S[1..i+1]$. In extension j of phase $i+1$ the algorithm first finds the end of the path in the tree that starting from the root spells out substring $S[j..i]$. Then, it extends it by adding character s_{i+1}, unless it already appears there (after the end of the path). So, in phase $i+1$, string $S[1..i+1]$ is first put on the tree, followed by strings $S[2..i+1]$, $S[3..i+1]$, and so on. Extension $i+1$ of phase $i+1$ extends the empty suffix of $S[1..i]$, that is, adds the single character s_{i+1} to the tree. The implicit tree $\mathcal{T}_1$ is just a single edge labeled s_1 connecting the root with leaf 1. The steps of the algorithm can be summarized as follows:

1. Construct tree $\mathcal{T}_1$;
2. For i from 1 to $n-1$ do
3. For j from 1 to $i+1$ do
4. Find the end of the path from the root labeled $S[j..i]$ in the current tree
5. Add character s_{i+1} if needed
6. End for
7. End for

When, at extension j of phase $i+1$, the end of the path corresponding to substring $S[j..i]$ has been located in the tree, the algorithm has to extend it to s_{i+1}. There are three possibilities, and for each one the tree is updated according to a different rule:

Rule 1 In the current tree, path $S[j..i]$ ends at a leaf. To update the tree, character s_{i+1} is appended to the label of the edge entering the leaf.

Rule 2 No path from the end of $S[j..i]$ starts with character s_{i+1}, but at least one other path continues from it (starting with a different character). A new leaf edge is created from the end of $S[j..i]$ and labeled with s_{i+1}. The leaf is labeled with number j. If the path ends within an edge, a new node has to be created at the end of the path, and the edge is broken accordingly, as in the naïve approach (see again Fig. 4.2).

Rule 3 Some path at the end of $S[j..i]$ starts with character s_{i+1}. In this case, nothing has to be done, since suffix $S[j..i+1]$ of $S[1..i+1]$ is already in the (implicit) tree.

Since adding character s_{i+1} takes in any case constant time, the time complexity of the algorithm depends on the time needed to locate, at each phase $i+1$, the $i+1$ substrings $S[j..i]$ in the tree under construction. Naïvely, we could just start from the root of the tree each time, matching the characters of $S[j..i]$ along the edges of the tree, taking $O(i+1-j)$ time at each extension. Thus, the implicit tree $\mathcal{T}_{i+1}$ would be built from $\mathcal{T}_i$ in $O(i^2)$ time, and the construction of the implicit tree $\mathcal{T}_n$ for the whole string would require, all in all, $O(n^3)$ time. Even worse than the naïve $O(n^2)$ approach. But, the cubic time can be reduced to linear with some clever observations and a few smart tricks. And, with the introduction of an additional feature to the tree: the *suffix link*.

Definition 4 *Let $x\alpha$, with $x \in \Sigma$ and $\alpha \in \Sigma^*$, be an arbitrary string spelled out by a path ending at an internal node p of the tree. If there exists another node q with path label α, then node q is the* suffix link *of node p.*

That is, the suffix link q of node p has the same path label except for the first symbol. From now on, we will denote with $L(p)$ the suffix link of node p. In case $\alpha = \epsilon$ (the empty string), then the suffix link is the root of the tree. Although the definition says nothing about it, every internal node of an implicit suffix tree has actually a suffix link from it.

Lemma 1 *If a new internal node p with path label $x\alpha$ is created in extension j of phase $i+1$, then either:*

- *a path labeled α already ends at an internal node of the suffix tree;*
- *a new node with path label α will be created at the next extension.*

Proof: A new internal node p is created in extension j of phase $i+1$ only when extension Rule 2 is applied. That is, path labeled $x\alpha$ ended within an edge, and continued with some character of Σ, say c, different from s_{i+1}. Thus, in extension $j+1$, we will meet a path labeled α in the tree, and it certainly will continue with c (since all the suffixes of $S[j..i]$ are contained in the tree). We will have two cases: the path labeled α continues only with character c, or continues also with some other characters. In the former case, a new node q will have to be created, and $L(p) = q$. Or, in the latter, the path labeled α already ends up at an internal node, exactly $L(p)$. □

Thus, each time a new internal node is created by the algorithm, we will be able to define its suffix link at the next extension. Therefore, in any implicit suffix tree $\mathcal{T}_i$ obtained at the end of phase i for any internal node p with path label $x\alpha$ there is a node $L(p)$ with path label α.

4.1.3 USING SUFFIX LINKS

Suffix links can be used as shortcuts when we move in the tree searching for substrings of S. Suppose that we have to perform extension j of phase $i+1$, that is, we first have to locate the path corresponding to substring $S[j..i]$ in the tree. In the previous extension $(j-1)$ we had located substring $S[j-1..i]$. Starting from the end of the path corresponding to it, we walk backwards at most one node on the path, until we reach either the root or an internal node p. If node p is not the root, then it has a suffix link, according to Lemma 1. Thus, we do not have to traverse more than one edge. Let γ be the edge label of the backwards walk. We move on to node $L(p)$ following the suffix link, then from $L(p)$ we follow the path labeled γ, reaching the end of $S[j..i]$. Note than such path always exists, because substring $S[j-1..i]$ is already in the tree, as well as its suffixes. Finally, we add character s_{i+1} according to one of the extension rules. If in extension $j-1$ a new node had been created, we also define its suffix link as shown in Lemma 1. The end of $S[j-1..i]$ could also have been an internal node p with a suffix link already defined: in this case, we follow immediately the link to node $L(p)$ and just add s_{i+1}. The first extension of any phase i must end at a leaf of $\mathcal{T}_i$, since $S[1..i]$ is the longest string represented in the tree. Thus, it suffices to keep a pointer to the leaf corresponding to the longest string contained in the tree (the same during all the phases), and handle the insertion of $S[1..i+1]$ by applying Rule 1 to the edge entering that leaf.

The introduction of suffix links saves a lot of character matching to the algorithm, since in most of the cases at extension j of phase $i+1$ it does not have to search in the tree for the whole substring $S[j..i]$ but just for a part of it. However, by itself, it does not improve the time bound of $O(n^3)$. But, we still have a couple of aces up our sleeve. Let's see if they help to improve the situation.

Remark 1 When we have to locate substring γ after following a suffix link $L(p)$, we know that there already exists a path labeled γ starting from $L(p)$, since if a substring $S[i..j]$ is already on the tree, all its suffixes are in it as well. Thus, we just have to follow the edge leaving $L(p)$ whose label starts with the first character of γ. If the length of the edge label (say, g) is shorter than the length of γ, then we do not have to match any character. We directly move to the node at the end of the edge, and repeat the process starting from character $g+1$ of γ, until we reach an edge whose label is longer than the remaining part of γ. Let g' be the length of it. The algorithm has just to skip g' characters down the edge to find the endpoint of $S[j..i]$. Thus, locating the endpoint of γ does not take any matching, and the time needed is no longer proportional to its length, but to the number of edges

that have to be traversed.

This observation leads to a first significant improvement of the time bound. Given a node p we call *node depth* of p the number of nodes on the path from the root to p. We can prove the following:

Lemma 2 *Let $p \to L(p)$ be a suffix link traversed during any extension step of the algorithm. The node depth of p is at most one greater than the node depth of $L(p)$.*

Proof: When suffix link $p \to L(p)$ is traversed, any internal ancestor of p with given path label $x\beta$ has a suffix link to a node whose path label is β. But, $x\beta$ is a prefix of the path to p, and β is a prefix of the path to $L(p)$: thus, all the suffix links of the ancestors of p go to ancestors of $L(p)$. And, if β is nonempty, then the node whose path from the root is labeled β is an internal node. Moreover, since two internal nodes must have different path labels, each ancestor of p has a suffix link to a different ancestor of $L(p)$. Thus, the node depth of $L(p)$ is at least one (for the root) plus the number of internal ancestors of p that have path labels longer than one character. Node p can have only one ancestor node without any corresponding ancestor of $L(p)$, that is, the node whose path label is one character long, and whose suffix link goes to the root. Therefore, p can have node depth at most one greater than $L(p)$. □

So, at this point we are ready to play our first ace:

Theorem 1 Using the method described in Remark 1, any phase of the algorithm takes $O(n)$ time.

Proof: Phase i is made of $i + 1 \leq n$ extensions. In a single extension, the algorithm walks up at by one edge to find an internal node with a suffix link, traverses one suffix link, and walks down some number of nodes. All the operations take constant time, so everything depends on how many edges must be traversed during the downwalk.

The up–walk decreases the current node depth by at most one, and for Lemma 2 traversing the suffix link decreases the node depth by at most another one. Each edge traversed in the down walk increases the node depth by one. The node depth is thus decremented at most $2(i + 1) \leq 2n$ times during all the extensions of phase $i + 1$, and since no node can have node depth greater than n, the total possible increment to the node depth is bounded by $3n$ in the entire phase. It follows that during the down walk at most $3n$ edges are traversed. Using the method described in Remark 1,

the time for each edge traversal is constant, so the total time in a phase is $O(n)$. □

Since the algorithm is made of n phases, the introduction of suffix links and Remark 1 have reduced the overall time to $O(n^2)$. The goal is very close: we just need a few more observations.

Remark 2 In any phase, when extension Rule 3 is applied for the first time in extension j, it will be applied also in *all* the remaining extensions of that phase. The reason is that Rule 3 is applied when path labeled $S[j..i]$ already continues with character s_{i+1}, and the same holds for paths labeled $S[j+1..i]$, $S[j+2..i]$, and so on. Thus, we can be sure that all the suffixes $S[j..i+1]$, $S[j+1..i+1]$, until $S[i+1]$ are already in the tree. So, once in a given extension of a phase we use Rule 3 for the first time, we can directly skip to the next phase, performing the remaining extensions *implicitly*.

Remark 3 When during the execution of the algorithm a new leaf is created with label j, then the leaf will remain a leaf in all the successive trees created by the algorithm. Only the label of the edge entering it will be modified by Rule 1. Leaf 1 is created in phase 1. Successive phases will start with a series of applications of Rules 1 and 2, stopped by the first application of Rule 3. New leaves will be created by Rule 2. Now, let j_i be the last extension performed by Rules 1 or 2 in phase i. We have that $j_i \leq j_{i+1}$. Now, recall that the edge labels are encoded with a pair of integers denoting the initial and final position of the corresponding substrings in string S. In phase $i+1$, when a leaf edge is created, instead of labeling it with $(p, i+1)$, we label it with (p, e), where e is a *global* variable denoting the current phase number. Moreover, all extensions from 1 to j_i will be performed by extension Rule 1. Instead of updating the labels of the edges entering the leaves, we can just increment the e variable by one, thus doing a constant amount of work.

Therefore, when the previous remarks are implemented by the algorithm, in phase $i+1$ all the extensions that have to be performed explicitly are those from $j_i + 1$ until the first extension where Rule 3 is applied. Phase $i+1$ can be thus summarized as follows:

1. Increment e to $i+1$ (Remark 3);
2. Perform explicitly extensions (using suffix links and Remark 1) from $j_i + 1$ to the first extension j^* where Rule 3 is applied (Remark 2);
3. Let $j_{i+1} = j^* - 1$;
4. Move on to phase $i+2$.

Phase $i+2$ will begin from extension j^*, where j^* was the last explicit extension of phase $i+1$. Therefore, two consecutive phases share at most *one* index where an explicit extension is performed. Moreover, phase $i+2$ knows where suffix $S[j^*..i+1]$ ends, so it does not need upwalks, suffix links, or edge traversals to find it in the tree. Now, we can finally prove the following:

Theorem 2 *Using suffix links and Remarks 1, 2, 3, Ukkonen's algorithm builds the implicit suffix trees $\mathcal{T}_1$ through $\mathcal{T}_n$ in $O(n)$ total time.*

Proof: The time for all the implicit extensions is constant, thus $O(n)$ in the entire algorithm. Now, assume that the algorithm is performing the first explicit extension of a phase. Let j_e be the extension number. The value of j_e never decreases between two successive phases, but it can remain the same. Since there are only n phases, and j_e is bounded by n, the algorithm executes in all only $2n$ explicit extensions. By following an argument similar to Lemma 2 and Theorem 1, we have the number of edges traversed is bounded by $6n = O(n)$. □

At the end of phase n, we have built the implicit suffix tree for string S. To convert it to the explicit one, first we append the marker character \$ to S, and perform an additional phase with this character. No suffix is now prefix of another suffix, and the result is an implicit suffix tree where each suffix ends at a different leaf. The last thing left to do is replace the e variable on the edges entering the leaves with n. This can be done with a linear–time traversal of the tree. The result is a true suffix tree for string S.

Theorem 3 *Given a string S on an alphabet Σ and a marker character $\$ \notin \Sigma$, Ukkonen's algorithm builds the suffix tree for $S\$$ in time linear in the length of the string.*

4.1.4 SUFFIX TREES FOR A SET OF STRINGS

In our problem, we are interested in working on sets of strings, rather than a single one. Extending the basic structure to this case is quite easy. The resulting structure is called *generalized suffix tree*.

The easiest way to build a generalized suffix tree for a set of strings $S = \{S^1, \ldots, S^k\}$ is to append a different marker to each string, concatenate all the strings into a single one, and build the suffix tree for the concatenated string. The resulting structure will have one leaf for each suffix of the concatenated string, and can be built in time linear in the sum of all the lengths of the strings of the set. The leaf numbers and edge labels can

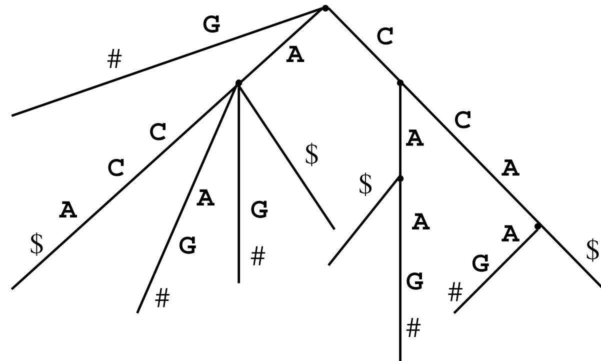

Figure 4.3: Generalized suffix tree for strings `ACCA$` and `CCAAG#`. Characters `$` and `#` are used as markers.

be easily adapted by using two numbers, one identifying a string S^i and the other a position inside S^i.

The main drawback of this method is that the tree will represent substring of the concatenated string that do not occur in the original strings (all those containing one or more markers in positions other than the last one). But, since each end of string marker is distinct and is not in any original string, the label of any path from the root to an internal node must be a substring of one of the original strings. Hence, by reducing the second index of the label on edges entering the leaves all the unwanted suffixes are removed, with no need to change any other part of the tree.

However, this method can be simulated without concatenating the string beforehand. First, we build the suffix tree for S^1. Then, starting from the root of the tree, we match S^2 against a path on the tree until a mismatch occurs. If the first i characters of S^2 match, the tree already encodes all the suffixes of string $S^2[1..i]$. Thus, the first i phases of Ukkonen's algorithm for S^2 have been implicitly executed, and, on the current tree, we can start from phase $i+1$ for string S^2. When S^2 has been fully processed, the tree will encode all the suffixes of S^1 and S^2, and we can move on to S^3, and so on. The time needed is linear in the overall length of the strings of the set (see Fig. 4.3).

4.1.5 MORE ON THE SPACE THEME

As we mentioned earlier, one way to implement the internal nodes of a suffix tree for a string of length n is to incorporate into them an array of $|\Sigma|$ pointers to children nodes, one for each symbol of the string alphabet. Node p has the i–th pointer set to the child node whose edge label starts

with the i–th character of the alphabet. This array allows constant time random access (thus no additional time overhead when we move along the edges of the tree during its construction), and although the space complexity is actually $O(|\Sigma|n)$, it works well in practice on biological sequences, where the alphabet size is usually four or twenty.

The pointer array starts to get impractical when the size of the alphabet grows. In theory, the alphabet size might be even larger than the string length, leading to a quadratic space complexity. In this case, one possible choice would be to use a linked list of pointers for each node, instead of an array, keeping only those pointers that are defined. If we keep the list ordered, accessing the needed pointer now costs additional $O(\log|\Sigma|)$ time, and the construction time is now the minimum between $O(n\log|\Sigma|)$ and $O(n\log n)$. Since there are in all $O(n)$ edges in the tree, we will need $O(n)$ pointers in the nodes, with a reduction on the space complexity to $O(n)$ (hence alphabet independent).

4.1.6 PATTERN MATCHING WITH SUFFIX TREES

We now return to the exact pattern matching problem we mentioned at the beginning of this chapter. We want to know whether a given pattern P of length m appears in string S of length n. First, we build our "index", the suffix tree for S, in $O(n)$ time. Pattern P occurs in S starting at position, say, j, if and only if P occurs as a prefix of suffix $S[j..n]$ of string S. Therefore, we can match the characters of P along the unique path in the tree, until P is exhausted or no more matches are possible. In the latter case, the pattern appears nowhere in S, while in the former every leaf in the subtree below the point of the last match is numbered with a starting location of P in S, and every starting location of P in S numbers such a leaf (see Fig. 4.4).

The matching path is unique because no two edges leaving the same node can have labels starting with the same character. If we implement the tree with an array of pointers, the work at each node takes constant time. Thus, checking whether P appears or not in S takes $O(m)$ time, with an overall complexity of $O(n+m)$. But, if at this point we want to look for another pattern, say P', we can take advantage of the tree that has already been built, and we need just the additional $O(|P'|)$ time for matching the characters of the pattern.

The advantage of first building an index structure for the string is thus self–evident in case we have to search for a large number of patterns. Instead of needing $O(n+|P|)$ time for each pattern P, we need $O(n)$ time only once, and just $O(|P|)$ for each pattern. There is something we have not done yet: traditional methods, in fact, also report where the pattern occurs in $O(n+m)$ time. We can do the same thing with suffix trees, that contain

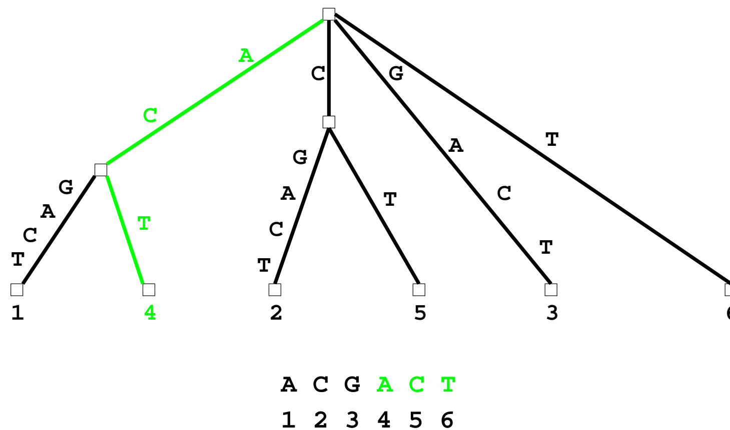

Figure 4.4: Finding pattern ACT in string ACGACT with a suffix tree. First character A is matched, then C, and finally T. The number of the leaf located below the end of the path tells us that the pattern occurs at position 4 in the string.

this information, but at the price of additional time. What we have to do is a traversal of the subtree below the last point where the characters of P were matched, and report the label of the leaves we encounter. Suppose P occurs k times. Since each node has at least two children, the number of leaves encountered is proportional to the number of edges traversed, so the time for the traversal is $O(k)$.

If, instead, we are interested just in *how many times* P appears in S, we can reduce the time complexity again to $O(m)$ with a linear time preprocessing of the tree. The idea is to annotate each internal node p of the tree with the number of leaves of the subtree rooted at p. This can be done with a linear time traversal of the suffix tree at the end of its construction. Thus, to find out how many times P appears in S we just have to read the number associated with the node entered by the edge containing the endpoint in the tree of the path corresponding to P.

The same arguments can be extended to finding the occurrences of a pattern in a set of k strings. In this case, we can annotate each internal node p with a k–bit strings telling us in which strings the substring spelled from the root to p appears in. If the i–th bit of the string is set, the pattern occurs in the i–th string. The bit string of an internal node p is obtained with the logical OR of the bit strings of the children of p. For $|\Sigma|$ children, this takes $O(|\Sigma|k)$ time. Since the tree contains $O(N)$ edges, annotating it with the bit strings takes additional $O(kN)$ time, where N is the overall length of the

input strings.

However, suffix trees and other text indexing structures offer much more significant advantages than simply providing faster access to the patterns contained in a string. They can be used to find repetitions in a single string, and for a number of problems related to string comparison. An overview is given in [36]. We will return to the virtues of suffix trees and similar structures in the next chapter. For the moment, we instead focus on another topic. How much space do suffix trees require in practice? Are there more space–efficient indexing structures?

4.1.7 EVERYONE'S LINEAR, BUT SOME ARE MORE LINEAR THAN THE OTHERS

As we have shown, suffix trees can be implemented in space linear in the length of the input string(s). However, in practice, one has to consider the additional overhead needed for the implementation of the structure, that is, its nodes, its edges, and its annotation, if needed. It has been estimated in [17] that a suffix tree for a n character random string requires on the average $1.62 \times n$ nodes, and each node takes 28.25 and 20.25 bytes when implemented with the pointer array and the linked list of pointers, respectively, if implemented as in [68]. On the average, suffix trees require 45.68 and 32.70 bytes per string character, without counting the additional space needed for their annotation with integers or bitstrings.

Research has thus focused on possible ways to reduce the space used by suffix trees. Clearly, one has to trade space for information or time: an example are *suffix arrays* [62], that are very space efficient but require additional time for retrieving the substrings. Basically, given an n character string S, the suffix array $\mathcal{A}$ of S is an array of n elements specifying the lexicographic order of the n suffixes of S (see Fig. 4.5). That is, the suffix at position $\mathcal{A}[1]$ is the lexically smallest suffix, and for any i, the suffix at position $\mathcal{A}[i]$ is lexically smaller than $\mathcal{A}[i+1]$. The suffix array of string S can be implemented just as an array of integers, denoting the starting position in S of the different suffixes. Thus, for a string of length n it requires just n memory words, provided that the words size is at least $\log n$ bits, and takes linear time to be built. The downside now is that searching for a pattern in S takes additional time. By using a binary search technique, a pattern of length m can be found in S in $O(m \log n)$ time. With further $O(n)$ preprocessing of the array [36] the search time can be reduced to $O(m + \log n)$. A cross between suffix trees and arrays is introduced in [51], and named *suffix cactus*. Suffix cacti time and space requirements lie in between those of suffix trees and suffix arrays.

An alternative approach could be earning space by reducing informa-

2: abxac
5: ac
3: bxac
6: c
1: xabxac
4: xac

Figure 4.5: Suffix array for string $xabxac$.

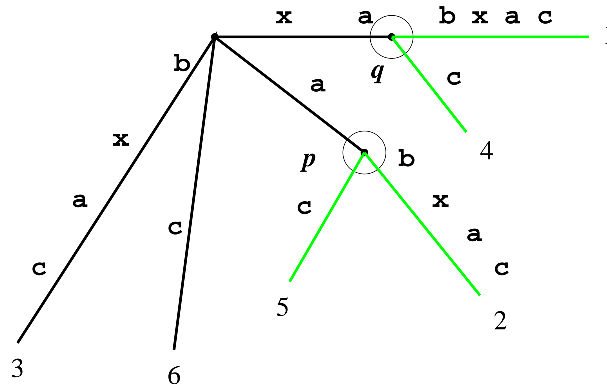

Figure 4.6: Suffix tree for string $xabxac$. The subtrees rooted at internal nodes p and q are isomorphic.

tion. For example, more than half of the nodes in a suffix tree are leaves, used to store the initial position of each suffix in the string. If we merge the leaves into a single *final node* (or *sink*) we obtain a directed acyclic graph with exactly $n - 1$ nodes less than the suffix tree, that still indexes all the substrings occurring in the strings. But this approach can be pushed even further. Consider the suffix tree for string $xabxac$ shown in Fig. 4.6.

The subtree rooted at node p (circled in the figure) is isomorphic to the one rooted at node q, and the edge labels are the same. The only difference is in the leaf numbers. If we are just interested in determining whether or not a pattern P occurs in string S, we could merge all the leaves into a single final node, and merge the subtrees rooted at p and q as well. In other words, all the edges out of p are removed, the edges entering p are redirected to q keeping their original edge labels, and any part of the graph unreachable from the root is removed. The key point is that all the substrings spelled out by paths ending at p and q occur as prefix of the same suffixes. Thus, we can merge p and q into a single node, without fear of deleting from the

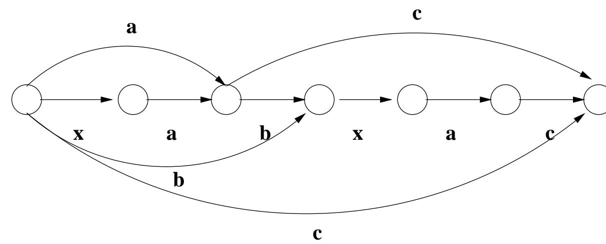

Figure 4.7: Directed Acyclic Word Graph for string $xabxac$.

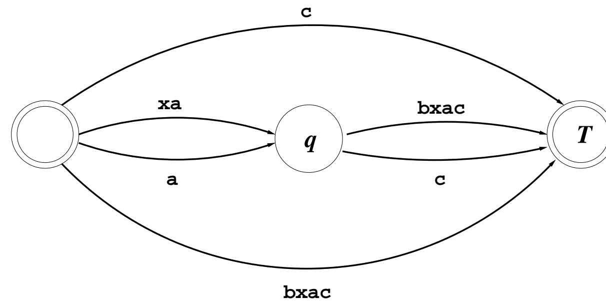

Figure 4.8: Compact Directed Acyclic Word Graph for string $xabxac$.

structure any suffix of the string.

On the other hand, the idea of representing the substrings and suffixes of a string with a graph was first introduced in [15, 16, 24]. The structure, called *Directed Acyclic Word Graph* (DAWG), was presented as the smallest finite state automaton recognizing the suffixes of a string whose transitions (edges) were labeled with single characters of the input string. The DAWG for a string S has at most $2|S| - 1$ nodes (states) and $3|S| - 4$ edges (transitions). Figure 4.7 shows an example of a DAWG for string $xabxac$.

A compact version of DAWG where all the states (nodes) with outdegree one are removed (thus edges are labeled with substrings instead of single characters of the input string) was introduced in [16, 17]. This structure, called *Compact Directed Acyclic Word Graph* (CDAWG) is exactly what we obtain by merging leaves and isomorphic subtrees of a suffix tree (see Fig. 4.8). In [16] also a CDAWG for a set of strings was presented, but both this structure and the one for a single string were built by compaction of a DAWG. Thus, in the case of the structure for multiple strings, if one string had to be added to the set after the CDAWG had been constructed the structure had to be built anew, starting from another DAWG.

The first algorithm for the construction of a CDAWG for a string, with-

out first having to build a suffix tree or a DAWG was presented in [26]. The algorithm processes all the suffixes of the input string from the longest to the shortest, in a fashion similar to McCreight's algorithm for the construction of suffix trees [68]. In this work, we instead present a novel *on line* algorithm for the direct construction of a CDAWG for a string, inspired by Ukkonen's algorithm for suffix trees. Moreover, we show how the algorithm can be extended in order to build the CDAWG for a set of strings, thus avoiding the preliminary construction of a DAWG as in [16]. We presented this algorithm in [45][2]. For a more "code–oriented" version of the algorithm, see also [46] and [47].

4.2 COMPACT DIRECTED ACYCLIC WORD GRAPHS

We first give a more formal definition of the Compact Directed Acyclic Word Graph for a string. Given a string S, let $Suf(S)$ be the set of suffixes of S. We define the syntactic congruence on Σ^* associated with $Suf(S)$ and denoted by $\equiv_{Suf(S)}$, for any $u, v \in \Sigma^*$, as:

$$u \equiv_{Suf(S)} v \iff u^{-1}Suf(S) = v^{-1}Suf(S)$$

That is, u and v are congruent if and only if they occur as prefixes of the same suffixes of S. In other words, the occurrences of u and v must end at the same positions in the string. Hence, if u and v occur in the string, one must be a suffix of the other. As in [16, 26], we will call *classes of factors* the congruence classes of the relation $\equiv_{Suf(S)}$. According to the definition, all the strings in Σ^* not occurring as a substring of S belong to the same class. The class composed by all the strings that are not substrings of S is called the *degenerate* class. The longest string in a non–degenerate class of factors is called the *representative* of the class.

Given a non–degenerate class of factors C of $\equiv_{Suf(S)}$, and its representative u, if there are at least two characters $a, b \in \Sigma$ such that ua and ub are substrings of S, then C is a *strict class of factors* of $\equiv_{Suf(S)}$.

From now on, we will say that two strings are *strictly congruent* if they belong to the same strict class of factors. An example was shown in Fig 4.8 for string $xabxac$. Strings xa and a are strictly congruent, since there exist two different characters (b and c) that continue the representative of the

[2]The work was submitted to the CPM 2001 conference. In due time, we received the referees' answers, telling us that the paper was original and interesting, and deserved to be presented at the conference. However, they pointed out, there was a group of japanese researchers that had submitted exactly the *same* algorithm to the *same* conference, at the *same* time. The papers were thus merged into a joint contribution.

class xa. Strings $xabx$ and abx are congruent, but not strictly, since they are continued only by character a.

We are now ready to give a formal definition of a CDAWG.

Definition 5 *The* Compact Directed Acyclic Word Graph *(CDAWG) of a string S is a directed acyclic graph, where:*

1. *two distinct nodes are marked as initial and final;*
2. *edges are labeled with non empty substrings of S;*
3. *labels of two edges leaving the same node cannot begin with the same character;*
4. *every suffix of S corresponds to a path on the graph starting from the initial node and ending at a node, such that the concatenation of the edge labels on the path exactly spells the suffix. From now on, we will call a node corresponding to a suffix of S* terminal node*;*
5. *substrings spelled by paths ending at the same non–terminal node of the graph belong to the same strict class of factors.*

The CDAWG of a string S has at most $|S| + 1$ nodes and $2|S| - 2$ edges [16, 26].

We will denote with (p, α, q) the edge $p \to q$ of the graph labeled with substring α. According to the definition of strict class of factors, all the nodes of the graph have outdegree greater than one, except for the initial node (that may have outdegree one, as for string *aaaaa*), the final node (that does not have outgoing edges), and the terminal nodes, that may have outdegree one. On the other hand, in the DAWG of a string S, where non–terminal nodes may have outdegree one, nodes correspond to the non degenerate classes of factors of S, but not necessarily strict (see Fig. 4.7).

As we did with suffix trees, we can define an *implicit* CDAWG:

Definition 6 *The* implicit *CDAWG of a string S is a CDAWG where nodes with outdegree one (except the initial node) are removed, and edges are merged accordingly.*

In the implicit CDAWG of a string S, the suffixes of S are spelled out by paths in the graph starting at the initial node, but not necessarily ending at a node. An example is shown in Fig. 4.9. For every node p, let $length_S(p)$ be the length of the longest substring spelled by a path from the initial node to p. Edges belonging to the spanning tree of the longest paths from the initial node are called *solid edges*. In other words, an edge (p, α, q) is

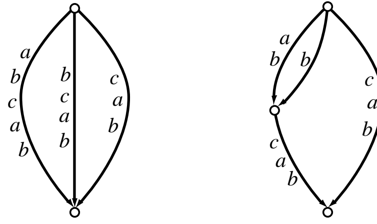

Figure 4.9: Implicit CDAWG and CDAWG for string $abcab$.

solid iff $length_S(q) = length_S(p) + |\alpha|$. Finally, we assume that the label of each edge is implemented with a pair of integers denoting the starting and ending point in the string of the substring corresponding to the label, and every node is annotated with the length of the longest path from the initial node.

4.2.1 ON LINE CONSTRUCTION OF CDAWGS

Given an alphabet Σ, let $S = s_1 \dots s_n$ be a string on Σ. As in Ukkonen's algorithm for suffix trees, our algorithm is divided in n phases, building at each phase i the implicit CDAWG $\mathcal{G}_i$ for each prefix $S[1..i]$ of S. More in detail, the implicit CDAWG $\mathcal{G}_{i+1}$ for $S[1..i+1]$ is constructed starting from graph $\mathcal{G}_i$ for $S[1..i]$. Again, each phase $i+1$ is divided in $i+1$ extensions, one for each of the $i+1$ suffixes of $S[1..i+1]$. In extension j of phase $i+1$, the algorithm finds the end of the path from the initial node labeled with substring $S[j..i]$, and extends it by adding character s_{i+1} to the path, unless it is already there. Therefore, in phase $i+1$, substring $S[1..i+1]$ is first put on the graph, followed by $S[2..i+1]$, $S[3..i+1]$, and so on. Extension $i+1$ of phase $i+1$ adds the single character s_{i+1} after the initial node. The initial graph $\mathcal{G}_1$ has one initial node I and one final node F, connected by an edge labeled by character s_1. The algorithm can be sketched as follows:

1. Construct graph $\mathcal{G}_1$;
2. For i from 1 to $n-1$ do
3.   For j from 1 to $i+1$ do
4.   Find the end of the path from I labeled $S[j..i]$ in the current graph
5.   Add character s_{i+1} if needed
6.   End for

7. End for

At extension j of phase $i+1$, once the path labeled $S[j..i]$ has been located, the CDAWG can be updated according to three different rules, similar to the ones of Ukkonen's algorithm for suffix trees:

Rule 1 In the current graph, the path labeled with $S[j..i]$ ends in F. To update the graph, character s_{i+1} is appended to the label of the edge entering F.

Rule 2 The path spelling $S[j..i]$ does not continue with s_{i+1}, but continues with at least one character c. If the path ends at a node p, we create a new edge (p, s_{i+1}, F). Otherwise, we create a new node q at the end of the path, splitting the edge in two at the point where the path ends. Then, we create a new edge (q, s_{i+1}, F).

Rule 3 Some path at the end of $S[j..i]$ continues with s_{i+1}. In this case, substring $S[j..i+1]$ is already in the current graph: we do nothing (hence the implicit graph).

These rules, however, do not guarantee that at the end of the phase we correctly constructed a CDAWG. In fact, the algorithm must also check whether a substring strictly congruent to another one has been encountered, or, conversely, whether a substring has to be removed from a strict class of factors, so that at the end of phase $i+1$ paths ending at the same node will correspond to strict classes of factors of $S[1..i+1]$, and vice versa. Here we sketch how the algorithm has to be modified.

Detecting strictly congruent factors

Two substrings α and β belong to the same class C iff they are prefixes of the same suffixes, and there are at least two characters $a, b \in \Sigma$ such that αa, αb, βa, and βb occur in s. Moreover, α must be a suffix of β, or vice versa. We suppose w.l.o.g. that $\alpha = c\beta$, with $c \in \Sigma$. We also assume that α and β have occurred just once, and that substrings αa and βa had been put in the graph in some previous phase (in two consecutive extensions), and in the current extension we have to insert αb.

The path spelling α ends along an edge, and the next character on the edge is a. A new node p is created at the end of the path, and a new edge (p, b, F) is created from p. At the following extension, we have to locate β in the graph. If β has occurred only once (together with α), it now belongs to the same strict class of factors, and we end in the middle of a *non–solid* edge that continues with a. In this case, we *redirect* the edge to p, labeling it with the part of the label that was contained in the path of β (see Fig. 4.10).

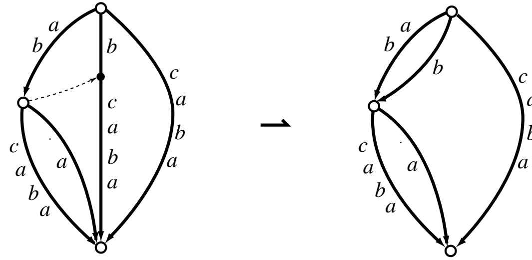

Figure 4.10: Implicit CDAWG for string $abcaba$ before (left) and after redirection of an edge, at phase 6, extension 5. Node 1, labeled ab, was created at the previous extension, after the insertion of a at the end of the path labeled ab. Now, path corresponding to b is found ending in the middle of non–solid edge $(I, bcaba, F)$, that is redirected to node 1 and becomes $(I, 1, b)$.

Since there can be more than two consecutive substrings to be assigned to the same class, it is possible that we again end along non–solid edges in the following extensions. In this case, we redirect the non–solid edges to p as well, until we reach an extension where we end at a node or along a solid edge. Otherwise, if β had previously occurred also by itself, either the path corresponding to β ends at a node (β has been followed by characters different from a), or the edge we end on is solid (β had been followed only by a). In the former case, if there is not an edge labeled b leaving the node we create a new edge labeled b to the final node. In the latter case, we create a new node and connect it to the final node. Then, there may be again non–solid edges that have to be redirected into the newly created node.

SPLITTING A STRICT CLASS OF FACTORS

Conversely, a substring that has been assigned to a strict class of factors has to be removed from the class if it does not occur as a suffix of the representative of the class when a new character s_{i+1} is added to the string. Let α and β, $\alpha = c\beta$, be the two substrings assigned to the same class in the previous example. Now, suppose that in phase $i + 1$ we have to insert β in the graph. In this case, s_{i+1} is the last character of β, and we find it at the end of the edge entering node p, that is non–solid, since β is not the representative of the class. Now we have two cases: s_{i+1} was found at the end of an edge that entered node p also at the previous extension, or we ended up somewhere else. In the former case, we had also inserted α at the previous extension of the same phase, therefore β still belongs to the

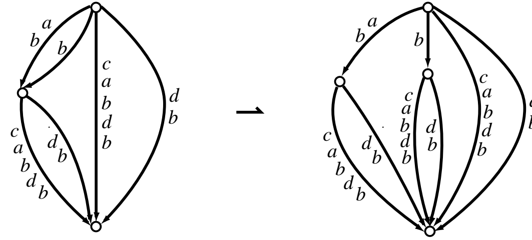

Figure 4.11: CDAWG for string *abcabdb* at phase 7, extension 7. Character b is found at the end of the non–solid edge $(I, b, 1)$. At extension 6, the path spelling *db* ended at the final node. Thus, b has to be removed from the class associated with node 1, that is cloned into node 2. Edge $(I, b, 1)$ becomes $(I, b, 2)$.

same class. In the latter, we have detected an occurrence of β not preceded by α, that is, not as a suffix of α, and we have to remove it from the class. To reflect this in the graph, we *clone* the node p into a new node q (that has the same outgoing edges), and redirect the non–solid edge to q keeping the same label. The redirected edge becomes solid. An example is shown in Fig. 4.11. If also some suffixes of β had been previously assigned to the same class as β, in the following extensions we will again find s_{i+1} at the end of a non–solid edge entering p. These edges are redirected to q. It can be proved that it suffices to check only the last edge on each path to ensure that a class has to be split. No cloning takes place if a character is found at the end of an edge entering the final node.

The two observations outlined above can be implemented in the algorithm by modifying Rules 2 and 3 accordingly. It is worth mentioning that both redirection of edges to a newly created class and node cloning can take place during the same phase. An example is shown in Fig. 4.12.

4.2.2 THE RETURN OF SUFFIX LINKS

Naïvely, locating the end of $S[j..i]$ in extension j of phase $i + 1$ would take $O(i - j)$ time by walking from the initial node and matching the characters of $S[j..i]$ along the edges of the graph. This would lead to an overall $O(n^3)$ time complexity for the construction of the whole graph. We will now reduce it to $O(n)$, as we did with suffix trees, by re–introducing *suffix links* and with some remarks.

Definition 7 *Let p be a node of the graph, different from the initial or final node. Let β be the representative of the class associated with p. The* suffix link *of p,*

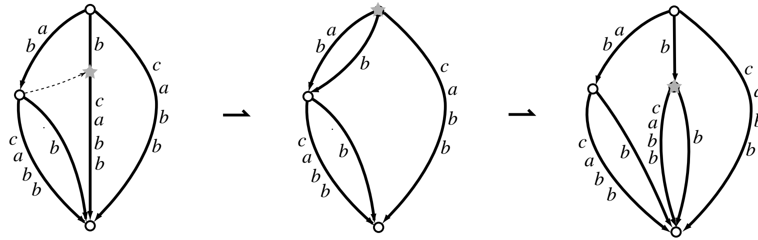

Figure 4.12: From left to right, CDAWG for string $abcabb$ at phase 6, extensions 5, 6, and 7. Character b is put in the graph after string ab, and the path spelling b is found in the middle on non–solid edge $(I, bcabb, F)$ (left) that is redirected to node labeled ab (center). Then, at extension 7 (that adds b after the empty string) b is found at the end of a non–solid edge. The node entered by the edge is cloned (right).

denoted by $L(p)$, is the node q whose representative γ is the longest suffix of β whose path does not end at p.

The suffix link of a node p can be implemented with a pointer from p to $L(p)$. If γ is empty, then $L(p)$ is the initial node. Suffix links are not defined for the initial and the final node. Although the definition does not guarantee that every node in the graph has a suffix link, we can prove the following:

Lemma 3 *Any node created during phase $i+1$ will have a suffix link from it by the end of the phase.*

Proof: In extension j of phase $i+1$ a new node p can be created at the end of a path spelling a substring α by application of Rule 4.2.1 or by cloning. In the former case, $L(p)$ will be the first node to be created or encountered at the end of the path corresponding to a suffix of $S[j..i]$ (possibly after edge redirections). Such a node always exists, since the last extension locates the empty suffix at the initial node. In the latter case, when a node p is cloned into node q with path spelling α, substring α is the longest suffix of the representative of p that does not belong to its class. Thus, $L(p)$ is set to q. Suffix link $L(q)$ is left undefined until one of the suffixes of α ends at a node other than p (that again could be I). $\square$

During any phase, the only node of the graph other than the initial and the final without a suffix link from it is the last created one. Let us suppose that the algorithm has completed extension j of phase $i+1$. Suffix links are used to speed up the search for the remaining suffixes of $S[j..i]$. Starting

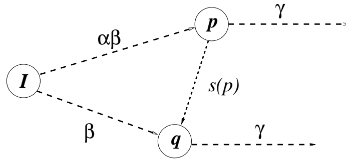

Figure 4.13: A suffix link. Node p corresponds to class $\alpha\beta$, node q corresponds to β. Paths labeled with suffixes of $\alpha\beta$ longer than β end at p. If at some extension j character s_{i+1} is added after $\alpha\beta\gamma$, then extensions from $j+1$ to $j+|\alpha|$ are implicitly performed as well.

from the end of $S[j..i]$ in the graph, we walk backwards along the corresponding path to either the initial node or a node p that has a suffix link. This requires traversing at most one edge. Let γ be the concatenation of the edge labels of the path from p to $S[j..i]$. If p is not the initial node, we move to node $L(p)$ and follow from it the path spelling γ. Otherwise, we search for $S[j+1..i]$ starting from I. Finally we add s_{i+1} according to one of the extension rules, redirecting an edge or cloning a node if needed. Notice that, if node p is the end of l different paths, the position reached after searching from γ from $L(p)$ will be the end of path $S[j+l..i]$, that is, extensions from $j+1$ to $j+l-1$ have been implicitly performed at extension j.

A path spelling γ starting from $L(p)$ always exists, since all the suffixes of $S[j..i]$ are already in the graph. Thus, to find the path spelling γ the algorithm just matches the first characters on the edges encountered. To obtain a linear time algorithm, we need just two more "tricks".

Remark 4 When during any extension Rule 3 is applied, that is, a given substring $S[j..i+1]$ is already on the graph, then the same rule will apply to all further extensions, since all the suffixes of $S[j..i+1]$ are already in the graph as well. Therefore, once Rule 3 is applied (and no node has to be cloned or edges redirected), we can stop and move on to the next phase, since all the strings to be inserted are already in the graph and no adjustment is needed for the classes.

Remark 5 If a new edge is created entering the final node during extension j of any phase i, then Rule 1 will always apply at extension j in any successive phase. That is, new characters will always be appended at the end of the last edge in the path associated with $S[j..i]$, that will enter the final node. Thus, when a new edge is created entering the final node with label $S[j..i+1]$, we label it with integers h and e ($j \leq h \leq i+1$), where e denotes the current phase, that is, the current end position in the string. If

we implement e with a global variable, and set it to $i + 1$ at the beginning of each phase $i+1$, we perform implicitly all the extensions that would end up at the final node.

Every phase i starts with a series of applications of Rules 1 and 2, that put s_i at the end of an edge entering the final node; when Rule 3 is applied for the first time, it will be also applied to all further extensions. Now, let j_i be the first extension where Rule 3 is applied with cloning in phase i, and j_i^* the first extension where it is applied without edge redirection to the cloned node. Extensions $j_i + 1$ to $j_i^* - 1$ will redirect edges to the last node created. Extensions from $j_i^* + 1$ to i need not to be performed, since in each of them we would not do anything. In phase $i + 1$, all extensions from 1 to $j_i - 1$ will apply Rule 1, therefore they are implicitly performed by setting the counter e to $i+1$. Thus, we can start phase $i+1$ directly from extension $j_i^* - 1$, until we find an extension where Rule 3 is applied without cloning or edge redirection. This can be done by starting phase $i + 1$ from the position in the graph of the last suffix of $S[1..i]$ that had to be redirected to the cloned node. This took place at extension $j_i^* - 1$. The first extension in phase $i + 1$ will have to look for s_{i+1} exactly at the endpoint of the last extension of phase i. This will also implicitly perform all extensions from j_i to $j_i^* - 1$. Of course, if in phase i Rule 3 is first applied without cloning we can move on to phase $i + 1$ as well.

The algorithm does not need to know which extension it is currently performing. That is, it starts phase $i + 1$ from the endpoint of phase i, adding s_{i+1}. Then it starts moving in the graph by using suffix links, and adding s_{i+1} at the end of each path. If the backward walk ends at I, and $\gamma = \gamma_1 \dots \gamma_k$ is the label of the path traversed, then it looks for the path labeled $\gamma_2 \dots \gamma_k$. Phase $i+1$ ends when the algorithm applies for the first time Rule 3 without node cloning or edge redirection. Moreover, whenever we find s_{i+1} at the end of a non–solid edge, we no longer have to check what happened at the previous extension, and just clone the node. In fact, if the representative of the class had been met during one of the previous extensions, we would have stopped the phase at that point, without reaching the current extension.

At the end of phase n, we have constructed the implicit CDAWG for string s. In order to obtain the actual CDAWG, we perform an additional extension phase $n+1$, extending the string to a dummy symbol \$ that does not belong to the string alphabet. Anyway, we do not increment the phase counter e to $n + 1$, so as to avoid appending \$ to edges entering the final node. Moreover, whenever a new node p has to be created, we do not add edge $(p, \$, F)$ to the graph. Nodes created in this phase will thus have outdegree one, and will correspond to terminal nodes of the CDAWG.

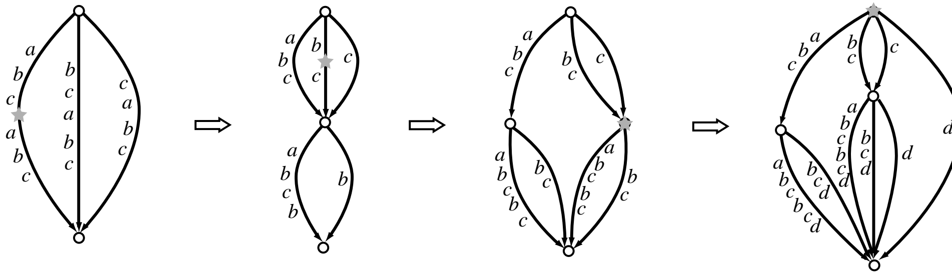

Figure 4.14: From left to right, construction of the CDAWG for string *abcabcbcd*: at the end of phase 6 (implicit CDAWG for string *abcabc*); at the end of phase 7 (*abcabcb*, where *abc*, *bc*, and *c* belong to the same strict class of factors); at the end of phase 8 (*abcabcbc*, where *bc* and *c* have been removed from the class with representative *abc*); the final structure. Dotted points indicate the position in the graph reached at the end of the last extension of each phase.

Notice that, whenever a path $S[j..n]$ ends along an edge, we always create a new node and mark it as terminal, while cloning of nodes and redirection of edges work as in the previous phases. When a path $S[j..n]$ ends at a node, we mark the node as terminal. At the end of the additional phase, the implicit CDAWG has been transformed into the actual CDAWG for string s. An example of the on–line construction of a CDAWG is shown in Fig. 4.14. With arguments analogous to Ukkonen's algorithm for suffix trees, we can prove the following:

Theorem 4 *Given a string $S = s_1 \dots s_n$ over a finite alphabet Σ, the on line algorithm implemented with suffix links and implicit extensions builds the CDAWG for S in $O(n)$ time and $O(n|\Sigma|)$ space if the nodes of the graph are implemented with a pointer array, or in $O(n|\Sigma|)$ time and $O(n)$ space with linked lists of pointers.*

Proof: The operations performed in any explicit extension (creation or cloning of nodes, edge redirections), that is, extensions that are not performed implicitly by incrementing the e counter, take constant time. Let j_i^* be the last explicit extension performed at phase i, and j_{i+1} the first explicit extension performed at phase $i+1$. In the worst case, we have $j_{i+1} = j_i^* - 1$. Moreover, for each i, $j_i \leq j_{i+1}$. Thus, at most $3n$ explicit extensions are performed by the algorithm. At any extension j of phase i, to locate the endpoint of $S[j..i]$ the algorithm walks back at most one edge from the endpoint of $S[j-1..i]$, follows a suffix link, and than traverses some edges

checking the first symbol on each edge. If the graph is implemented with a pointer array in each node, traversing an edge takes constant time. Else, it takes $O(|\Sigma|)$ time. The only thing unaccounted for is the overall number of edges traversed. For every node p of the graph, let the *node depth* of p be the number of nodes on the path from the root to p labeled with the representative of the class associated with p. As in Ukkonen's algorithm for suffix trees, the node depth during all the explicit extensions is reduced at most by $O(n)$ edges, and since the maximum node–depth is n, the maximum number of edges traversed is bounded by $O(n)$. □

4.3 THE CDAWG FOR A SET OF STRINGS

The basic idea of the CDAWG for a set of strings $\hat{S} = \{S^1, \ldots, S^k\}$ is the same of the single string structure. Now, the nodes of the structure correspond to patterns that occur as prefix of the same suffixes in every string of the set. In other words, given $Suf(\hat{S})$ (the set of the suffixes of the k strings), the nodes of the CDAWG correspond to strict classes of factors for $\equiv_{Suf(\hat{S})}$.The only difference is that now we have k final nodes $F_1 \ldots F_k$, one for each string, and we want all the suffixes of S^i to end at the corresponding final node F_i. This result can be obtained by appending a different termination symbol, not belonging to the string alphabet, to each string of the set. More formally:

Definition 8 *The CDAWG for a set of strings $\hat{S} = S^1 \ldots S^k$ is a directed acyclic graph where:*

1. *a node is marked as initial and k distinct nodes $F_1 \ldots F_k$ are marked as final;*
2. *edges are labeled with non empty substrings of at least one of the strings;*
3. *labels of two edges leaving the same node cannot begin with the same character;*
4. *for every string S^i in the set, all the suffixes of S^i are spelled by paths starting at the initial node and ending at final node F_i;*
5. *paths ending at non final nodes correspond to strict classes of factors of the congruence relation $\equiv_{Suf(\hat{S})}$.*

The CDAWG for a set of strings can be constructed with the algorithm presented in the previous section. First, we build the CDAWG for string S^1 (with the termination symbol) and final node F_1. Notice that, since the termination symbol does not occur elsewhere in S^1, the resulting structure is

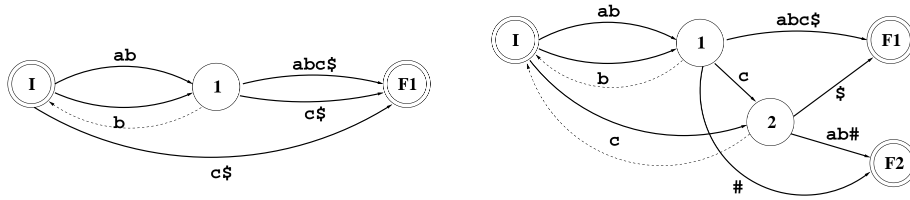

Figure 4.15: CDAWG for strings $ababc\$$ and $abcab\#$, after the insertion of $ababc\$$ (left) and $abcab\#$ (right). Characters $ and # are used as terminations. Dotted edges show suffix links.

a CDAWG, with no need to perform the additional phase. Then, string S^2 is added to the graph, but in this case with final node F_2. The same will apply to every other string in the set. Node cloning and edge redirection rules ensure the correctness of the resulting structure. It can be proved that the algorithm takes $O(N)$ time to construct the structure, implemented with pointer arrays, where $N = ||\hat{S}|| = \sum_{i=1}^{k} |S^i|$. This structure (with marginal differences) was first described in [16], where it was built by reducing a DAWG. Therefore, adding a new string to the set required the construction of a new DAWG from scratch. The algorithm presented here, instead, permits to add strings directly to the compact structure (see Fig. 4.15). As in [16] we can give an upper bound on the size of the structure.

Theorem 5 (Blumer et.al., [16]) *The CDAWG for a set of strings $S^1 \dots S^k$, has at most $N + k$ nodes, where $N = \sum_{i=1}^{k} |S^i|$.*

Proof: Consider the suffix tree for string $\hat{S} = S^1 t_1 S^2 t_2 \dots S^k t_k$, that is, the concatenation of the k strings augmented with the termination symbols. The tree has $N + k$ leaves and at most N internal nodes. Its minimization into a CDAWG merges all leaves into the unique final state F, and possibly merges other nodes. Thus, the CDAWG has at most $N + 1$ nodes. The CDAWG for the set of strings can be derived from the CDAWG by redirecting edges entering the final node to the final node corresponding to the first termination symbol encountered on their label. At the end, we have k final nodes instead of one. The overall number of nodes is therefore at most $N + k$. □

4.4 HOW MUCH SPACE DO WE SAVE?

As we have seen, the theoretical bounds show that a CDAWG for a string has in the worst case half the nodes of the corresponding suffix tree. This result comes directly from the fact that all the leaves of the tree are merged into a single final node.

As shown in [17], the theoretical average number of states for a CDAWG of a random n character string is $0.54 \times n$. An efficient way to implement it is described in [26]. For each node, the target nodes of outgoing edges can be implemented with a pointer array of $|\Sigma|$ integers. Usually, four bytes per integer are sufficient. Moreover, for each edge we need to store the value of the starting and final position in the input string of its label. Finally, we can indicate whether an edge is solid or not with a one–bit flag (0.125 bytes), with an average memory requirement of 22.40 bytes per input character.

One might ask why saving memory in the implementation of text–indexing structures is such an issue. After all, RAM is becoming cheaper and cheaper, and it's not unusual, nowadays, to find desktop PCs equipped with 1 Gigabyte of RAM, or even more. Interestingly enough, in the first paper that presented the CDAWG ([16], dated 1987) the authors stated that

> The drawback of the suffix tree is the space it occupies. Although significantly smaller than the suffix tree in many cases, the compact DAWG for a large set of text, say 1 Megabyte, still requires considerable storage space, depending on the implementation (...) Unless these problems can be overcome, the application of the CDAWG will be limited to smaller, intensively searched or analyzed text collections (...)

Sic transit gloria mundi. In the biological applications of text indexing structures, the input strings are usually *huge*. See again Table 2.1 for some figures. The smallest genomes, the bacterial ones, range from 500,000 characters to more than five million. To have an idea, consider that Dante's *Divina Commedia* is composed by about 500,000 characters, including spaces. If we want to build a suffix tree for a relatively small bacterial genome, say, of 2 Mbytes, the implementation of the tree alone requires about 60 Mbytes of memory, without all the additional data structures needed, for example, to annotate the tree, or traverse it. Thus, the advantage of using more space–efficient text indexing structures is self evident: the more space you save for the structure, the longer is the genome sequence you can analyze. If we manage to halve the memory requirement, for example by using a CDAWG instead of a suffix tree, then our algorithms are able to treat sequences twice as long.