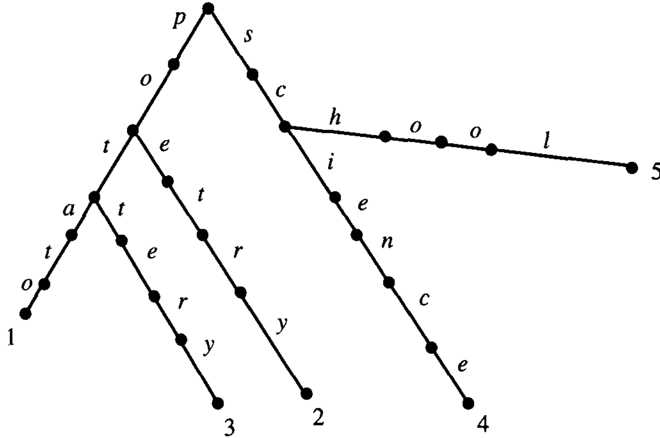

Figure 3.13: Keyword tree $\mathcal{K}$ with five patterns.

from them, whereas the order is just the opposite in the method based on fundamental preprocessing.

3.4. Exact matching with a set of patterns

An immediate and important generalization of the exact matching problem is to find all occurrences in text T of any pattern in a *set* of patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$. This generalization is called the *exact set matching* problem. Let n now denote the total length of all the patterns in $\mathcal{P}$ and m be, as before, the length of T. Then, the exact set matching problem can be solved in time $O(n + zm)$ by separately using any linear-time method for each of the z patterns.

Perhaps surprisingly, the exact set matching problem can be solved faster than, $O(n + zm)$. It can be solved in $O(n + m + k)$ time, where k is the number of occurrences in T of the patterns from $\mathcal{P}$. The first method to achieve this bound is due to Aho and Corasick [9].[2] In this section, we develop the Aho–Corasick method; some of the proofs are left to the reader. An equally efficient, but more robust, method for the exact set matching problem is based on suffix trees and is discussed in Section 7.2.

Definition The *keyword tree* for set $\mathcal{P}$ is a rooted directed tree $\mathcal{K}$ satisfying three conditions: 1. each edge is labeled with exactly one character; 2. any two edges out of the same node have distinct labels; and 3. every pattern P_i in $\mathcal{P}$ maps to some node v of $\mathcal{K}$ such that the characters on the path from the root of $\mathcal{K}$ to v exactly spell out P_i, and every leaf of $\mathcal{K}$ is mapped to by some pattern in $\mathcal{P}$.

For example, Figure 3.13 shows the keyword tree for the set of patterns {*potato, poetry, pottery, science, school*}.

Clearly, every node in the keyword tree corresponds to a prefix of one of the patterns in $\mathcal{P}$, and every prefix of a pattern maps to a distinct node in the tree.

Assuming a fixed-size alphabet, it is easy to construct the keyword tree for $\mathcal{P}$ in $O(n)$ time. Define $\mathcal{K}_i$ to be the (partial) keyword tree that encodes patterns $P_1, \ldots, P_i$ of $\mathcal{P}$.

[2] There is a more recent exposition of the Aho–Corasick method in [8], where the algorithm is used just as an "acceptor", deciding whether or not there is an occurrence in T of at least one pattern from $\mathcal{P}$. Because we will want to explicitly find all occurrences, that version of the algorithm is too limited to use here.

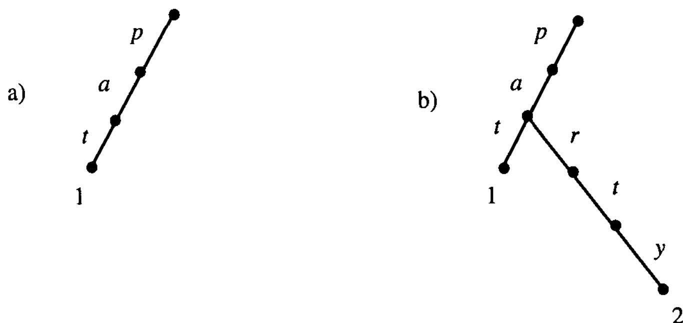

Figure 3.14: Pattern P_1 is the string *pat*. a. The insertion of pattern P_2 when P_2 is *pa*. b. The insertion when P_2 is *party*.

Tree $\mathcal{K}_1$ just consists of a single path of $|P_1|$ edges out of root r. Each edge on this path is labeled with a character of P_1 and when read from the root, these characters spell out P_1. The number 1 is written at the node at the end of this path. To create $\mathcal{K}_2$ from $\mathcal{K}_1$, first find the longest path from root r that matches the characters of P_2 in order. That is, find the longest prefix of P_2 that matches the characters on some path from r. That path either ends by exhausting P_2 or it ends at some node v in the tree where no further match is possible. In the first case, P_2 already occurs in the tree, and so we write the number 2 at the node where the path ends. In the second case, we create a new path out of v, labeled by the remaining (unmatched) characters of P_2, and write number 2 at the end of that path. An example of these two possibilities is shown in Figure 3.14.

In either of the above two cases, $\mathcal{K}_2$ will have at most one branching node (a node with more than one child), and the characters on the two edges out of the branching node will be distinct. We will see that the latter property holds inductively for any tree $\mathcal{K}_i$. That is, at any branching node v in $\mathcal{K}_i$, all edges out of v have distinct labels.

In general, to create $\mathcal{K}_{i+1}$ from $\mathcal{K}_i$, start at the root of $\mathcal{K}_i$ and follow, as far as possible, the (unique) path in $\mathcal{K}_i$ that matches the characters in P_{i+1} in order. This path is unique because, at any branching node v of $\mathcal{K}_i$, the characters on the edges out of v are distinct. If pattern P_{i+1} is exhausted (fully matched), then number the node where the match ends with the number $i + 1$. If a node v is reached where no further match is possible but P_{i+1} is not fully matched, then create a new path out of v labeled with the remaining unmatched part of P_{i+1} and number the endpoint of that path with the number $i + 1$.

During the insertion of P_{i+1}, the work done at any node is bounded by a constant, since the alphabet is finite and no two edges out of a node are labeled with the same character. Hence for any i, it takes $O(|P_{i+1}|)$ time to insert pattern P_{i+1} into $\mathcal{K}_i$, and so the time to construct the entire keyword tree is $O(n)$.

3.4.1. Naive use of keyword trees for set matching

Because no two edges out of any node are labeled with the same character, we can use the keyword tree to search for all occurrences in T of patterns from $\mathcal{P}$. To begin, consider how to search for occurrences of patterns in $\mathcal{P}$ that begin at character 1 of T: Follow the unique path in $\mathcal{K}$ that matches a prefix of T as far as possible. If a node is encountered on this path that is numbered by i, then P_i occurs in T starting from position 1. More than one such numbered node can be encountered if some patterns in $\mathcal{P}$ are prefixes of other patterns in $\mathcal{P}$.

In general, to find all patterns that occur in T, start from each position l in T and follow the unique path from r in $\mathcal{K}$ that matches a substring of T starting at character l.

Numbered nodes along that path indicate patterns in $\mathcal{P}$ that start at position l. For a fixed l, the traversal of a path of $\mathcal{K}$ takes time proportional to the minimum of m and n, so by successively incrementing l from 1 to m and traversing $\mathcal{K}$ for each l, the exact set matching problem can be solved in $O(nm)$ time. We will reduce this to $O(n + m + k)$ time below, where k is the number of occurrences.

The dictionary problem

Without any further embellishments, this simple keyword tree algorithm efficiently solves a special case of set matching, called the *dictionary problem*. In the dictionary problem, a set of strings (forming a dictionary) is initially known and preprocessed. Then a sequence of individual strings will be presented; for each one, the task is to find if the presented string is contained in the dictionary. The utility of a keyword tree is clear in this context. The strings in the dictionary are encoded into a keyword tree $\mathcal{K}$, and when an individual string is presented, a walk from the root of $\mathcal{K}$ determines if the string is in the dictionary. In this special case of exact set matching, the problem is to determine if the text T (an individual presented string) completely matches some string in $\mathcal{P}$.

We now return to the general set matching problem of determining which strings in $\mathcal{P}$ are *contained* in text T.

3.4.2. The speedup: generalizing Knuth-Morris-Pratt

The above naive approach to the exact set matching problem is analogous to the naive search we discussed before introducing the Knuth-Morris-Pratt method. Successively incrementing l by one and starting each search from root r is analogous to the naive exact match method for a single pattern, where after every mismatch the pattern is shifted by only one position, and the comparisons are always begun at the left end of the pattern. The Knuth-Morris-Pratt algorithm improves on that naive algorithm by shifting the pattern by more than one position when possible and by never comparing characters to the left of the current character in T. The Aho–Corasick algorithm makes the same kind of improvements, incrementing l by more than one and skipping over initial parts of paths in $\mathcal{K}$, when possible. The key is to generalize the function sp_i (defined on page 27 for a single pattern) to operate on a set of patterns. This generalization is fairly direct, with only one subtlety that occurs if a pattern in $\mathcal{P}$ is a proper substring of another pattern in $\mathcal{P}$. So, it is very helpful to (temporarily) make the following assumption:

Assumption No pattern in $\mathcal{P}$ is a proper substring of any other pattern in $\mathcal{P}$.

3.4.3. Failure functions for the keyword tree

Definition Each node v in $\mathcal{K}$ is *labeled* with the string obtained by concatenating in order the characters on the path from the root of $\mathcal{K}$ to node v. $\mathcal{L}(v)$ is used to denote the label on v. That is, the concatenation of characters on the path from the root to v spells out the string $\mathcal{L}(v)$.

For example, in Figure 3.15 the node pointed to by the arrow is labeled with the string *pott*.

Definition For any node v of $\mathcal{K}$, define $lp(v)$ to be the length of the longest proper suffix of string $\mathcal{L}(v)$ that is a prefix of some pattern in $\mathcal{P}$.

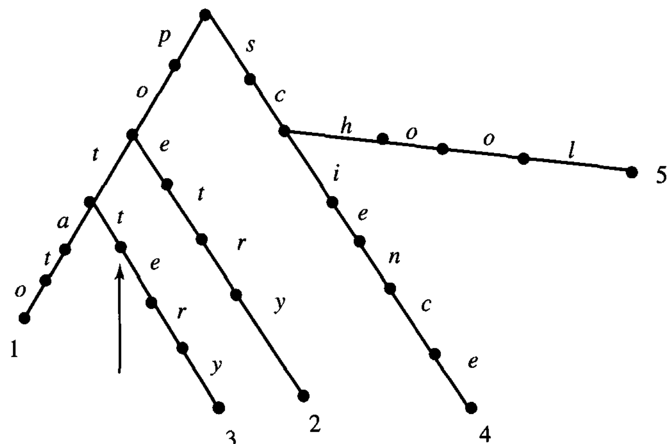

Figure 3.15: Keyword tree to illustrate the label of a node.

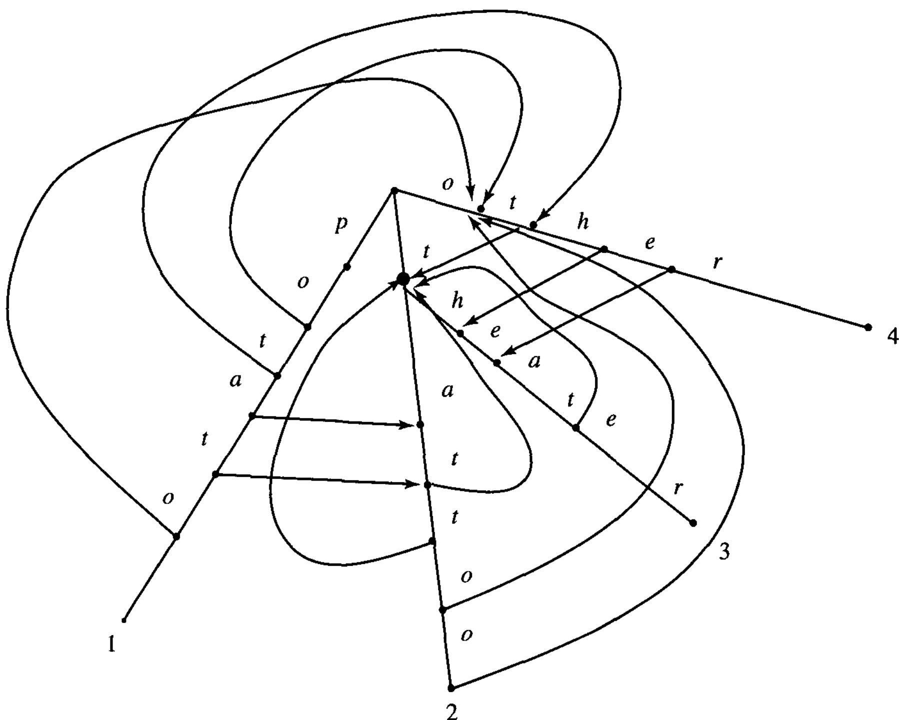

Figure 3.16: Keyword tree showing the failure links.

For example, consider the set of patterns $\mathcal{P}$ = {*potato, tattoo, theater, other*} and its keyword tree shown in Figure 3.16. Let v be the node labeled with the string *potat*. Since *tat* is prefix of *tattoo*, and it is the longest proper suffix of *potat* that is a prefix of any pattern in $\mathcal{P}$, $lp(v) = 3$.

Lemma 3.4.1. *Let α be the $lp(v)$-length suffix of string $\mathcal{L}(v)$. Then there is a unique node in the keyword tree that is labeled by string α.*

PROOF $\mathcal{K}$ encodes all the patterns in $\mathcal{P}$ and, by definition, the $lp(v)$-length suffix of $\mathcal{L}(v)$ is a prefix of some pattern in $\mathcal{P}$. So there must be a path from the root in $\mathcal{K}$ that spells out

string α. By the construction of $\mathcal{T}$ no two paths spell out the same string, so this path is unique and the lemma is proved. □

Definition For a node v of $\mathcal{K}$ let n_v be the unique node in $\mathcal{K}$ labeled with the suffix of $\mathcal{L}(v)$ of length $lp(v)$. When $lp(v) = 0$ then n_v is the root of $\mathcal{K}$.

Definition We call the ordered pair (v, n_v) a *failure link.*

Figure 3.16 shows the keyword tree for $\mathcal{P}$ = {*potato, tatoo, theater, other*}. Failure links are shown as pointers from every node v to node n_v where $lp(v) > 0$. The other failure links point to the root and are not shown.

3.4.4. The failure links speed up the search

Suppose that we know the failure link $v \mapsto n_v$ for each node v in $\mathcal{K}$. (Later we will show how to efficiently find those links.) How do the failure links help speed up the search? The Aho–Corasick algorithm uses the function $v \mapsto n_v$ in a way that directly generalizes the use of the function $i \mapsto sp_i$ in the Knuth-Morris-Pratt algorithm. As before, we use l to indicate the starting position in T of the patterns being searched for. We also use pointer c into T to indicate the "current character" of T to be compared with a character on $\mathcal{K}$. The following algorithm uses the failure links to search for occurrences in T of patterns from $\mathcal{P}$:

Algorithm AC search

```
l := 1;
c := 1;
w := root of K;
repeat
    While there is an edge (w, w') labeled character T(c)
        begin
        if w' is numbered by pattern i then
            report that P_i occurs in T starting at position l;
        w := w' and c := c + 1;
        end;
    w := n_w and l := c − lp(w);
until c > m;
```

To understand the use of the function $v \mapsto n_v$, suppose we have traversed the tree to node v but cannot continue (i.e., character $T(c)$ does not occur on any edge out of v). We know that string $\mathcal{L}(v)$ occurs in T starting at position l and ending at position $c - 1$. By the definition of the function $v \mapsto n_v$, it is guaranteed that string $\mathcal{L}(n_v)$ matches string $T[c - lp(v)..c - 1]$. That is, the algorithm could traverse $\mathcal{K}$ from the root to node n_v and be sure to match all the characters on this path with the characters in T starting from position $c - lp(v)$. So when $lp(v) \geq 0$, l can be increased to $c - lp(v)$, c can be left unchanged, and there is no need to actually make the comparisons on the path from the root to node n_v. Instead, the comparisons should begin at node n_v, comparing character c of T against the characters on the edges out of n_v.

For example, consider the text T = *xxpotattooxx* and the keyword tree shown in Figure 3.16. When $l = 3$, the text matches the string *potat* but mismatches at the next character. At this point $c = 8$, and the failure link from the node v labeled *potat* points

to the node n_v labeled *tat*, and $lp(v) = 3$. So l is incremented to $5 = 8 - 3$, and the next comparison is between character $T(8)$ and character t on the edge below *tat*.

With this algorithm, when no further matches are possible, l may increase by more than one, avoiding the reexamination of characters of T to the left of c, and yet we may be sure that every occurrence of a pattern in $\mathcal{P}$ that *begins* at character $c - lp(v)$ of T will be correctly detected. Of course (just as in Knuth-Morris-Pratt), we have to argue that there are no occurrences of patterns of $\mathcal{P}$ starting strictly between the old l and $c - lp(v)$ in T, and thus l can be incremented to $c - lp(v)$ without missing any occurrences. With the given assumption that no pattern in $\mathcal{P}$ is a proper substring of another one, that argument is almost identical to the proof of Theorem 2.3.2 in the analysis of Knuth-Morris-Pratt, and it is left as an exercise.

When $lp(v) = 0$, then l is increased to c and the comparisons begin at the root of $\mathcal{K}$. The only case remaining is when the mismatch occurs at the root. In this case, c must be incremented by 1 and comparisons again begin at the root.

Therefore, the use of function $v \mapsto n_v$ certainly accelerates the naive search for patterns of $\mathcal{P}$. But does it improve the worst-case running time? By the same sort of argument used to analyze the search time (not the preprocessing time) of Knuth-Morris-Pratt (Theorem 2.3.3), it is easily established that the search time for Aho–Corasick is $O(m)$. We leave this as an exercise. However, we have yet to show how to precompute the function $v \mapsto n_v$ in linear time.

3.4.5. Linear preprocessing for the failure function

Recall that for any node v of $\mathcal{K}$, n_v is the unique node in $\mathcal{K}$ labeled with the suffix of $\mathcal{L}(v)$ of length $lp(v)$. The following algorithm finds node n_v for each node v in $\mathcal{K}$, using $O(n)$ total time. Clearly, if v is the root r or v is one character away from r, then $n_v = r$. Suppose, for some k, n_v has been computed for every node that is exactly k or fewer characters (edges) from r. The task now is to compute n_v for a node v that is $k+1$ characters from r. Let v' be the parent of v in $\mathcal{K}$ and let x be the character on the v' to v edge, as shown in Figure 3.17.

We are looking for the node n_v and the (unknown) string $\mathcal{L}(n_v)$ labeling the path to it from the root; we know node $n_{v'}$ because v' is k characters from r. Just as in the explanation

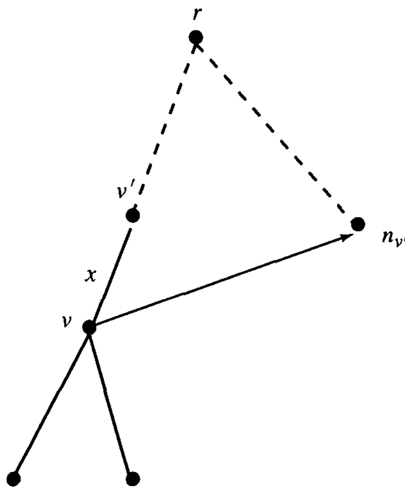

Figure 3.17: Keyword tree used to compute the failure function for node v.

of the classic preprocessing for Knuth-Morris-Pratt, $\mathcal{L}(n_v)$ must be a suffix of $\mathcal{L}(n_{v'})$ (not necessarily proper) followed by character x. So the first thing to check is whether there is an edge $(n_{v'}, w')$ out of node $n_{v'}$ labeled with character x. If that edge does exist, then n_v is node w' and we are done. If there is no such edge out of $n_{v'}$ labeled with character x, then $\mathcal{L}(n_v)$ is a *proper* suffix of $\mathcal{L}(n_{v'})$ followed by x. So we examine $n_{n_{v'}}$ next to see if there is an edge out of it labeled with character x. (Node $n_{n_{v'}}$ is known because $n_{v'}$ is k or fewer edges from the root.) Continuing in this way, with exactly the same justification as in the classic preprocessing for Knuth-Morris-Pratt, we arrive at the following algorithm for computing n_v for a node v:

Algorithm n_v

v' is the parent of v in $\mathcal{K}$;
x is the character on the edge (v', v);
$w := n_{v'}$;
While there is no edge out of w labeled x and $w \neq r$
  do $w := n_w$;
end (while);
If there is an edge (w, w') out of w labeled x then
  $n_v := w'$;
else
  $n_v := r$;

Note the importance of the assumption that n_u is already known for every node u that is k or fewer characters from r.

To find n_v for every node v, repeatedly apply the above algorithm to the nodes in $\mathcal{K}$ in a breadth-first manner starting at the root.

Theorem 3.4.1. *Let n be the total length of all the patterns in $\mathcal{P}$. The total time used by Algorithm n_v when applied to all nodes in $\mathcal{K}$ is $O(n)$.*

PROOF The argument is a direct generalization of the argument used to analyze time in the classic preprocessing for Knuth-Morris-Pratt. Consider a single pattern P in $\mathcal{P}$ of length t and its path in $\mathcal{K}$ for pattern P. We will analyze the time used in the algorithm to find the failure links for the nodes on this path, as if the path shares no nodes with paths for any other pattern in $\mathcal{P}$. That analysis will overcount the actual amount of work done by the algorithm, but it will still establish a linear time bound.

The key is to see how $lp(v)$ varies as the algorithm is executed on each successive node v down the path for P. When v is one edge from the root, then $lp(v)$ is zero. Now let v be an arbitrary node on the path for P and let v' be the parent of v. Clearly, $lp(v) \leq lp(v')+1$, so over all executions of Algorithm n_v for nodes on the path for P, $lp()$ is increased by a total of at most t. Now consider how $lp()$ can decrease. During the computation of n_v for any node v, w starts at $n_{v'}$ and so has initial node depth equal to $lp(v')$. However, during the computation of n_v, the node depth of w decreases every time an assignment to w is made (inside the *while* loop). When n_v is finally set, $lp(v)$ equals the current depth of w, so if w is assigned k times, then $lp(v) \leq lp(v') - k$ and $lp()$ decreases by at least k. Now $lp()$ is never negative, and during all the computations along path P, $lp()$ can be increased by a total of at most t. It follows that over all the computations done for nodes on the path for P, the number of assignments made inside the *while* loop is at most t. The total time used is proportional to the number of assignments inside the loop, and hence all failure links on the path for P are set in $O(t)$ time.

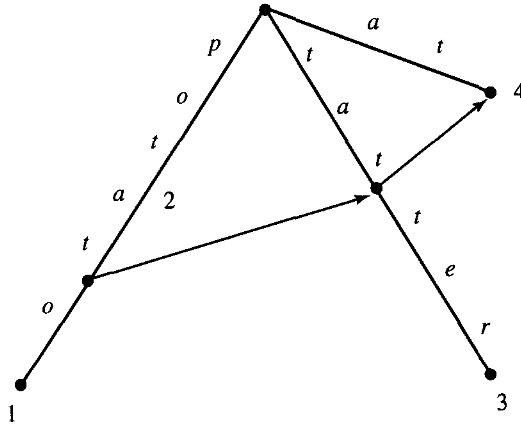

Figure 3.18: Keyword tree showing a directed path from *potat* to *at* through *tat*.

Repeating this analysis for every pattern in $\mathcal{P}$ yields the result that all the failure links are established in time proportional to the sum of the pattern lengths in $\mathcal{P}$ (i.e., in $O(n)$ total time). □

3.4.6. The full Aho–Corasick algorithm: relaxing the substring assumption

Until now we have assumed that no pattern in $\mathcal{P}$ is a substring of another pattern in $\mathcal{P}$. We now relax that assumption. If one pattern is a substring of another, and yet *Algorithm AC search* (page 56) uses the same keyword tree as before, then the algorithm may make l too large. Consider the case when $\mathcal{P} = \{acatt, ca\}$ and $T = acatg$. As given, the algorithm matches T along a path in $\mathcal{K}$ until character g is the current character. That path ends at the node v with $\mathcal{L}(v) = acat$. Now no edges out of v are labeled g, and since no proper suffix of *acat* is a prefix of *acatt* or *ac*, n_v is the root of $\mathcal{K}$. So when the algorithm gets stuck at node v it returns to the root with g as the current character, and it sets l to 5. Then after one additional comparison the current character pointer will be set to $m + 1$ and the algorithm will terminate without finding the occurrence of *ca* in T. This happens because the algorithm shifts (increases l) so as to match the longest *suffix* of $\mathcal{L}(v)$ with a prefix of some pattern in $\mathcal{P}$. Embedded occurrences of patterns in $\mathcal{L}(v)$ that are not suffixes of $\mathcal{L}(v)$ have no influence on how much l increases.

It is easy to repair this problem with the following observations whose proofs we leave to the reader.

Lemma 3.4.2. *Suppose in a keyword tree $\mathcal{K}$ there is a directed path of failure links (possibly empty) from a node v to a node that is numbered with pattern i. Then pattern P_i must occur in T ending at position c (the current character) whenever node v is reached during the search phase of the Aho–Corasick algorithm.*

For example, Figure 3.18 shows the keyword tree for $\mathcal{P} = \{potato, pot, tatter, at\}$ along with some of the failure links. Those links form a directed path from the node v labeled *potat* to the numbered node labeled *at*. If the traversal of $\mathcal{K}$ reaches v then T certainly contains the patterns *tat* and *at* end at the current c.

Conversely,

Lemma 3.4.3. *Suppose a node v has been reached during the algorithm. Then pattern*

P_i occurs in T ending at position c only if v is numbered i or there is a directed path of failure links from v to the node numbered i.

So the full search algorithm is

Algorithm full AC search

```
l := 1;
c := 1;
w := root;
repeat
    While there is an edge (w, w') labeled T(c)
        begin
        if w' is numbered by pattern i or there is
        a directed path of failure links from w' to a node numbered with i
            then report that P_i occurs in T ending at position c;
        w := w' and c := c + 1;
        end;
    w := n_w and l := c - lp(w);
until c > n;
```

Implementation

Lemmas 3.4.2 and 3.4.3 specify at a high level how to find all occurrences of the patterns in the text, but specific implementation details are still needed. The goal is to be able to build the keyword tree, determine function $v \mapsto n_v$, and be able to execute the full AC search algorithm all in $O(m + k)$ time. To do this we add an additional pointer, called the *output link*, to each node of $\mathcal{K}$.

The output link (if there is one) at a node v points to that numbered node (a node associated with the end of a pattern in $\mathcal{P}$) other than v that is reachable from v by the fewest failure links. The output links can be determined in $O(n)$ time during the running of the preprocessing algorithm n_v. When the n_v value is determined, the possible output link from node v is determined as follows: If n_v is a numbered node then the output link from v points to n_v; if n_v is not numbered but has an output link to a node w, then the output link from v points to w; otherwise v has no output link. In this way, an output link points only to a numbered node, and the path of output links from any node v passes through all the numbered nodes reachable from v via a path of failure links. For example, in Figure 3.18 the nodes for *tat* and *potat* will have their output links set to the node for *at*. The work of adding output links adds only constant time per node, so the overall time for algorithm n_v remains $O(n)$.

With the output links, all occurrences in T of patterns of $\mathcal{P}$ can be detected in $O(m+k)$ time. As before, whenever a numbered node is encountered during the full AC search, an occurrence is detected and reported. But additionally, whenever a node v is encountered that has an output link from it, the algorithm must traverse the path of output links from v, reporting an occurrence ending at position c of T for each link in the path. When that path traversal reaches a node with no output link, it returns along the path to node v and continues executing the full AC search algorithm. Since no character comparisons are done during any output link traversal, over both the construction and search phases of the algorithm the number of character comparisons is still bounded by $O(n+m)$. Further, even though the number of traversals of output links can exceed that linear bound, each traversal

of an output link leads to the discovery of a pattern occurrence, so the total time for the algorithm is $O(n+m+k)$, where k is the total number of occurrences. In summary we have,

Theorem 3.4.2. *If $\mathcal{P}$ is a set of patterns with total length n and T is a text of total length m, then one can find all occurrences in T of patterns from $\mathcal{P}$ in $O(n)$ preprocessing time plus $O(m+k)$ search time, where k is the number of occurrences. This is true even without assuming that the patterns in $\mathcal{P}$ are substring free.*

In a later chapter (Section 6.5) we will discuss further implementation issues that affect the practical performance of both the Aho–Corasick method, and suffix tree methods.

3.5. Three applications of exact set matching

3.5.1. Matching against a DNA or protein library of known patterns

There are a number of applications in molecular biology where a relatively stable library of interesting or distinguishing DNA or protein substrings have been constructed. The *Sequence-tagged sites* (*STS*s) and *Expressed sequence tags* (*EST*s) provide our first important illustration.

Sequence-tagged-sites

The concept of a Sequence-tagged-site (STS) is one of the most useful by-products that has come out of the Human Genome Project [111, 234, 399]. Without going into full biological detail, an STS is intuitively a DNA string of length 200–300 nucleotides whose right and left ends, of length 20–30 nucleotides each, occur only once in the entire genome [111, 317]. Thus each STS occurs uniquely in the DNA of interest. Although this definition is not quite correct, it is adequate for our purposes. An early goal of the Human Genome Project was to select and map (locate on the genome) a set of STSs such that any substring in the genome of length 100,000 or more contains at least one of those STSs. A more refined goal is to make a map containing ESTs (expressed sequence tags), which are STSs that come from genes rather than parts of intergene DNA. ESTs are obtained from mRNA and cDNA (see Section 11.8.3 for more detail on cDNA) and typically reflect the protein coding parts of a gene sequence.

With an STS map, one can locate on the map any sufficiently long string of anonymous but sequenced DNA – the problem is just one of finding which STSs are contained in the anonymous DNA. Thus with STSs, map location of anonymous sequenced DNA becomes a string problem, an exact set matching problem. The STSs or the ESTs provide a computer-based set of indices to which new DNA sequences can be referenced. Presently, hundreds of thousands of STSs and tens of thousands of ESTs have been found and placed in computer databases [234]. Note that the total length of all the STSs and ESTs is very large compared to the typical size of an anonymous piece of DNA. Consequently, the keyword tree and the Aho–Corasick method (with a search time proportional to the length of the anonymous DNA) are of direct use in this problem for they allow very rapid identification of STSs or ESTs that occur in newly sequenced DNA.

Of course, there may be some errors in either the STS map or in the newly sequenced DNA causing trouble for this approach (see Section 16.5 for a discussion of STS maps). But in this application, the number of errors should be a small percentage of the length of the STS, and that will allow more sophisticated exact (and inexact) matching methods to succeed. We will describe some of these in Sections 7.8.3, 9.4, and 12.2 of the book.

A related application comes from the "BAC-PAC" proposal [442] for sequencing the human genome (see page 418). In that method, 600,000 strings (patterns) of length 500 would first be obtained and entered into the computer. Thousands of times thereafter, one would look for occurrences of any of these 600,000 patterns in text strings of length 150,000. Note that the total length of the patterns is 300 million characters, which is two-thousand times as large as the typical text to be searched.

3.5.2. Exact matching with wild cards

As an application of exact set matching, we return to the problem of exact matching with a single pattern, but complicate the problem a bit. We modify the exact matching problem by introducing a character ϕ, called a *wild card*, that matches any single character. Given a pattern P containing wild cards, we want to find all occurrences of P in a text T. For example, the pattern $ab\phi\phi c\phi$ occurs twice in the text $xabvccbababcax$. Note that in this version of the problem no wild cards appear in T and that each wild card matches only a single character rather than a substring of unspecified length.

The problem of matching with wild cards should need little motivating, as it is not difficult to think up realistic cases where the pattern contains wild cards. One very important case where simple wild cards occur is in DNA *transcription factors*. A transcription factor is a protein that binds to specific locations in DNA and regulates, either enhancing or suppressing, the transcription of the DNA into RNA. In this way, production of the protein that the DNA codes for is regulated. The study of transcription factors has exploded in the past decade; many transcription factors are now known and can be separated into families characterized by specific substrings containing wild cards. For example, the *Zinc Finger* is a common transcription factor that has the following signature:

$$\text{CYS}\phi\phi\text{CYS}\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\phi\text{HIS}\phi\phi\text{HIS},$$

where CYS is the amino acid cysteine and HIS is the amino acid histidine. Another important transcription factor is the *Leucine Zipper*, which consists of four to seven leucines, each separated by six wild card amino acids.

If the number of permitted wild cards is unbounded, it is not known if the problem can be solved in linear time. However, if the number of wild cards is bounded by a fixed constant (independent of the size of P) then the following method, based on exact set pattern matching, runs in linear time:

Exact matching with wild cards

0. Let C be a vector of length $|T|$ initialized to all zeros.

1. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be the (multi-)set of maximal substrings of P that do not contain any wild cards. Let $l_1, l_2, \ldots, l_k$ be the starting positions in P of each of these substrings.

{For example, if $P = ab\phi\phi c\phi ab\phi\phi$ then $\mathcal{P} = \{ab, c, ab\}$ and $l_1 = 1, l_2 = 5, l_3 = 7$.}

2. Using the Aho–Corasick algorithm (or the suffix tree approach to be discussed later), find for each string P_i in $\mathcal{P}$, all starting positions of P_i in

text T. For each starting location j of P_i in T,
increment the count in cell $j - l_i + 1$ of C by one.

{For example, if the second copy of string ab is found in T
starting at position 18, then cell 12 of C is incremented by one.}

3. Scan vector C for any cell with value k. There is an occurrence of P
in T starting at position p if and only if $C(p) = k$.

Correctness and complexity of the method

Correctness Clearly, there is an occurrence of P in T starting at position p if and only if, for each i, subpattern $P_i \in \mathcal{P}$ occurs at position $j = p + l_i - 1$ of T. The above method uses this idea in reverse. If pattern $P_i \in \mathcal{P}$ is found to occur starting at position j of T, and pattern P_i starts at position l_i in P, then this provides one "witness" that P occurs at T starting at position $p = j - l_i + 1$. Hence P occurs in T starting at p if and only if similar witnesses for position p are found for each of the k strings in $\mathcal{P}$. The algorithm counts, at position p, the number of witnesses that observe an occurrence of P beginning at p. This correctly determines whether P occurs starting at p because each string in $\mathcal{P}$ can cause at most one increment to cell p of C.

Complexity The time used by the Aho–Corasick algorithm to build the keyword tree for $\mathcal{P}$ is $O(n)$. The time to search for occurrences in T of patterns from $\mathcal{P}$ is $O(m + z)$, where $|T| = m$ and z is the number of occurrences. We treat each pattern in $\mathcal{P}$ as being distinct even if there are multiple copies of it in $\mathcal{P}$. Then whenever an occurrence of a pattern from $\mathcal{P}$ is found in T, exactly one cell in C is incremented; furthermore, a cell can be incremented to at most k. Hence z must be bounded by km, and the algorithm runs in $O(km)$ time . Although the number of character comparisons used is just $O(m)$, km need not be $O(m)$ and hence the number of times C is incremented may grow faster than $O(m)$, leading to a nonlinear $O(km)$ time bound. But if k is assumed to be bounded (independent of $|P|$), then the method does run in linear time. In summary,

Theorem 3.5.1. *If the number of wild cards in pattern P is bounded by a constant, then the exact matching problem with wild cards in the Pattern can be solved in $O(n + m)$ time.*

Later, in Sections 9.3, we will return to the problem of wild cards when they occur in either the pattern, text, or both.

3.5.3. Two-dimensional exact matching

A second classic application of exact set matching occurs in a generalization of string matching to two-dimensional exact matching. Suppose we have a *rectangular* digitized picture T, where each point is given a number indicating its color and brightness. We are also given a smaller rectangular picture P, which also is digitized, and we want to find all occurrences (possibly overlapping) of the smaller picture in the larger one. We assume that the bottom edges of the two rectangles are parallel to each other. This is a two-dimensional generalization of the exact string matching problem.

Admittedly, this problem is somewhat contrived. Unlike the one-dimensional exact matching problem, which truly arises in numerous practical applications, compelling applications of two-dimensional exact matching are hard to find. Two-dimensional matching that is inexact, allowing some errors, is a more realistic problem, but its solution requires

more complex techniques of the type we will examine in Part III of the book. So for now, we view two-dimensional exact matching as an illustration of how exact set matching can be used in more complex settings and as an introduction to more realistic two-dimensional problems. The method presented follows the basic approach given in [44] and [66]. Since then, many additional methods have been presented since that improve on those papers in various ways. However, because the problem as stated is somewhat unrealistic, we will not discuss the newer, more complex, methods. For a sophisticated treatment of two-dimensional matching see [22] and [169].

Let m be the total number of points in T, let n be the number of points in P, and let n' be the number of rows in P. Just as in exact string matching, we want to find the smaller picture in the larger one in $O(n + m)$ time, where $O(nm)$ is the time for the obvious approach. Assume for now that each of the rows of P are distinct; later we will relax this assumption.

The method is divided into two phases. In the first phase, search for all occurrences of each of the rows of P among the rows of T. To do this, add an end of row marker (some character not in the alphabet) to each row of T and concatenate these rows together to form a single text string T' of length $O(m)$. Then, treating each row of P as a separate pattern, use the Aho–Corasick algorithm to search for all occurrences in T' of any row of P. Since P is rectangular, all rows have the same width, and so no row is a proper substring of another and we can use the simpler version of Aho–Corasick discussed in Section 3.4.2. Hence the first phase identifies all occurrences of complete rows of P in complete rows of T and takes $O(n + m)$ time.

Whenever an occurrence of row i of P is found starting at position (p, q) of T, write the number i in position (p, q) of another array M with the same dimensions as T. Because each row of P is assumed to be distinct and because P is rectangular, at most one number will be written in any cell of M.

In the second phase, scan each *column* of M, looking for an occurrence of the string $1, 2, \ldots, n'$ in consecutive cells in a single column. For example, if this string is found in column 6, starting at row 12 and ending at row $n' + 12$, then P occurs in T when its upper left corner is at position (6,12). Phase two can be implemented in $O(n' + m) = O(n + m)$ time by applying any linear-time exact matching algorithm to each column of M.

This gives an $O(n + m)$ time solution to the two-dimensional exact set matching problem. Note the similarity between this solution and the solution to the exact matching problem with wild cards discussed in the previous section. A distinction will be discussed in the exercises.

Now suppose that the rows of P are not all distinct. Then, first find all identical rows and give them a common label (this is easily done during the construction of the keyword tree for the row patterns). For example, if rows 3, 6, and 10 are the same then we might give them all the label of 3. We do a similar thing for any other rows that are identical. Then, in phase one, only look for occurrences of row 3, and not rows 6 and 10. This ensures that a cell of M will have at most one number written in it during phase 1. In phase 2, don't look for the string $1, 2, 3, \ldots, n'$ in the columns of M, but rather for a string where 3 replaces 6 and 10, etc. It is easy to verify that this approach is correct and that it takes just $O(n + m)$ time. In summary,

Theorem 3.5.2. *If T and P are rectangular pictures with m and n cells, respectively, then all exact occurrences of P in T can be found in $O(n + m)$ time, improving upon the naive method, which takes $O(nm)$ time.*

3.6. Regular expression pattern matching

A *regular expression* is a way to specify a set of related strings, sometimes referred to as a *pattern*.[3] Many important sets of substrings (patterns) found in biosequences, particularly in proteins, can be specified as regular expressions, and several databases have been constructed to hold such patterns. The PROSITE database, developed by Amos Bairoch [41, 42], is the major regular expression database for significant patterns in proteins (see Section 15.8 for more on PROSITE).

In this section, we examine the problem of finding substrings of a text string that match one of the strings specified by a given regular expression. These matches are computed in the Unix utility *grep*, and several special programs have been developed to find matches to regular expressions in biological sequences [279, 416, 422].

It is helpful to start first with an example of a simple regular expression. A formal definition of a regular expression is given later. The following PROSITE expression specifies a set of substrings, some of which appear in a particular family of granin proteins:

[ED]-[EN]-L-[SAN]-x-x-[DE]-x-E-L.

Every string specified by this regular expression has ten positions, which are separated by a dash. Each capital letter specifies a single amino acid and a group of amino acids enclosed by brackets indicates that exactly one of those amino acids must be chosen. A small x indicates that any one of the twenty amino acids from the protein alphabet can be chosen for that position. This regular expression describes 192,000 amino acid strings, but only a few of these actually appear in any known proteins. For example, ENLSSEDEEL is specified by the regular expression and is found in human granin proteins.

3.6.1. Formal definitions

We now give a formal, recursive definition for a regular expression formed from an alphabet Σ. For simplicity, and contrary to the PROSITE example, assume that alphabet Σ does not contain any symbol from the following list: $*$, $+$, (,), ϵ.

Definition A single character from Σ is a regular expression. The symbol ϵ is a regular expression. A regular expression followed by another regular expression is a regular expression. Two regular expressions separated by the symbol "+" form a regular expression. A regular expression enclosed in parentheses is a regular expression. A regular expression enclosed in parentheses and followed by the symbol "*" is a regular expression. The symbol * is called the Kleene closure.

These recursive rules are simple to follow, but may need some explanation. The symbol ϵ represents the empty string (i.e., the string of length zero). If R is a parenthesized regular expression, then R^* means that the expression R can be repeated any number of times (including zero times). The inclusion of parentheses as part of a regular expression (outside of Σ) is not standard, but is closer to the way that regular expressions are actually specified in many applications. Note that the example given above in PROSITE format does not conform to the present definition but can easily be converted to do so.

As an example, let Σ be the alphabet of lower case English characters. Then $R = (a + c + t)ykk(p + q)^*\ vdt(l + z + \epsilon)(pq)$ is a regular expression over Σ, and $S =$

[3] Note that in the context of regular expressions, the meaning of the word "pattern" is different from its previous and general meaning in this book.

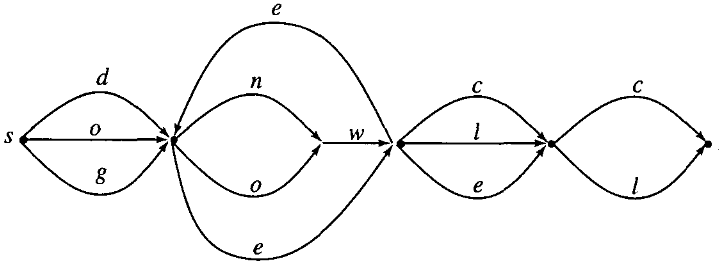

Figure 3.19: Directed graph for the regular expression $(d + o + g)((n + o)w)^*(c + l + \epsilon)(c + l)$.

aykkpqppvdtpq is a string specified by R. To specify S, the subexpression $(p+q)$ of R was repeated four times, and the empty string ϵ was the choice specified by the subexpression $(l + z + \epsilon)$.

It is very useful to represent a regular expression R by a directed graph $G(R)$ (usually called a nondeterministic, finite state automaton). An example is shown in Figure 3.19. The graph has a start node s and a termination node t, and each edge is labeled with a single symbol from $\Sigma \cup \epsilon$. Each s to t path in $G(R)$ specifies a string by concatenating the characters of Σ that label the edges of the path. The set of strings specified by all such paths is exactly the set of strings specified by the regular expression R. The rules for constructing $G(R)$ from R are simple and are left as an exercise. It is easy to show that if a regular expression R has n symbols, then $G(R)$ can be constructed using at most $2n$ edges. The details are left as an exercise and can be found in [10] and [8].

Definition A substring T' of string T *matches* the regular expression R if there is an s to t path in $G(R)$ that specifies T'.

Searching for matches

To search for a *substring* in T that matches the regular expression R, we first consider the simpler problem of determining whether some (unspecified) *prefix* of T matches R. Let $N(0)$ be the set of nodes consisting of node s plus all nodes of $G(R)$ that are reachable from node s by traversing edges labeled ϵ. In general, a node v is in set $N(i)$, for $i > 0$, if v can be reached from some node in $N(i-1)$ by traversing an edge labeled $T(i)$ followed by zero or more edges labeled ϵ. This gives a constructive rule for finding set $N(i)$ from set $N(i-1)$ and character $T(i)$. It easily follows by induction on i that a node v is in $N(i)$ if and only if there is path in $G(R)$ from s that ends at v and generates the string $T[1..i]$. Therefore, prefix $T[1..i]$ matches R if and only if $N(i)$ contains node t.

Given the above discussion, to find all prefixes of T that match R, compute the sets $N(i)$ for i from 0 to m, the length of T. If $G(R)$ contains e edges, then the time for this algorithm is $O(me)$, where m is the length of the text string T. The reason is that each iteration i [finding $N(i)$ from $N(i-1)$ and character $T(i)$] can be implemented to run in $O(e)$ time (see Exercise 29).

To search for a *nonprefix* substring of T that matches R, simply search for a prefix of T that matches the regular expression Σ^*R. Σ^* represents any number of repetitions (including zero) of any character in Σ. With this detail, we now have the following:

Theorem 3.6.1. *If T is of length m, and the regular expression R contains n symbols, then it is possible to determine whether T contains a substring matching R in $O(nm)$ time.*