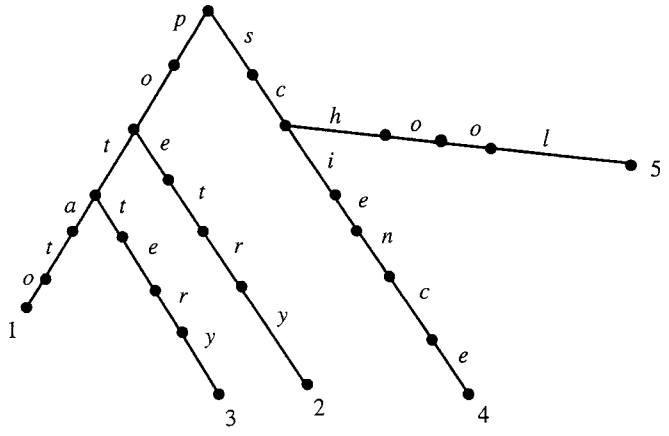

Figure 3.13: Keyword tree $\mathcal{K}$ with five patterns.

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.

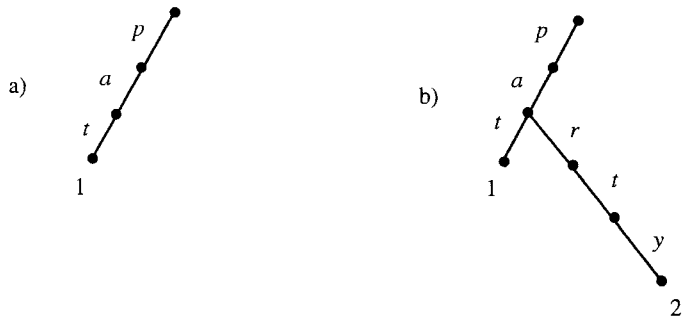

Figure 3.14: Pattern P_1 is the string *pat*. a. The insertion of pattern P_2 when P_2 is *pa*. b. The insertion when P_2 is *party*.

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 $\mathcal{P}$.

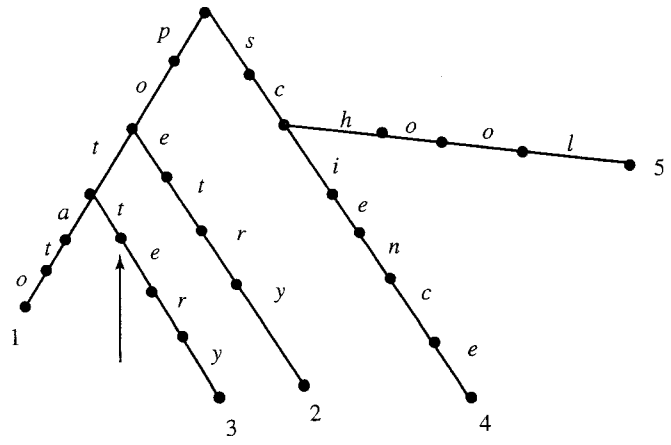

Figure 3.15: Keyword tree to illustrate the label of a node.

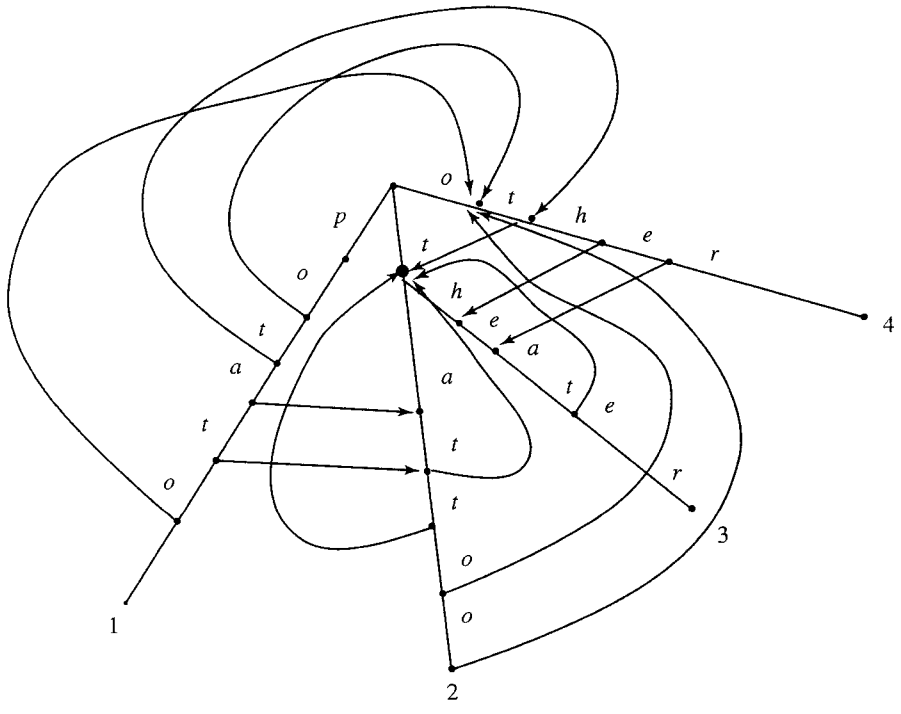

Figure 3.16: Keyword tree showing the failure links.

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 *tatoo*, 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 $T = xxpotatfpoxx$ 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

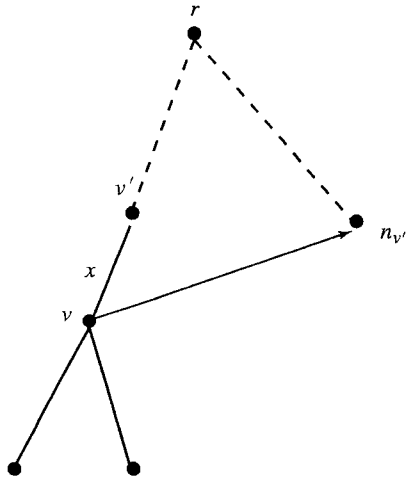

Figure 3.17: Keyword tree used to compute the failure function for node v.

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.

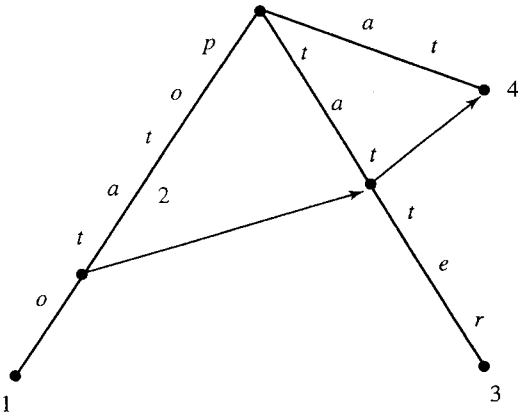

Figure 3.18: Keyword tree showing a directed path from *potat* to *at* through *tat*.

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