The second important trick is to add the so-called failure link. This idea is very similar to KMP. For now, we assume no pattern is a substring of another pattern. This issue will be fixed later. Recall in KMP, when performing compare and shift, shift by the size of longest suffix of the currently matched part of P that is also a proper prefix of it. Recall this shift will never miss any hits. The failure link concept is the same: for each node v in the tree, let \( L(v) \) be the spelled out string (i.e. prefix of some pattern). \( L(p(v)) \) is the length of longest proper suffix of \( L(v) \) that is also a prefix of some pattern in \( P \). Note: there can be multiple patterns in \( P \) that satisfy the prefix-suffix condition but by using the keyword tree node we resolve this multiplicity issue by a single link. Also we require the longest one (and this ensure we will not miss any true hits). As an example, you should try to add failure links for the set of pattern \( P = \{ \text{potato, tattoo, theater, other} \} \).

It is easy to say there is only a single node in keyword tree that is labeled by \( L(p(v)) \) length suffix. Why? If there is one, of course it is unique: the suffix is fixed anyway; also there must be one since it is a prefix of some \( P \) and we must be able to find it when tracing to that \( P \). How are failure links used? Simple: we keep matching; when there is a mismatch (that is, \( T \) can no longer match the keyword tree), then we follow the failure link and continue. Why does this work? Just like KMP: moving by the length of largest prefix-suffix match ensures no hits will be missed. Time analysis: we never re-examine a char in \( T \) and thus the total time is \( O(m) \): shift can be done by no more than \( m \) times and number of comparisons is at \( m \) times.

Next, how do we construct failure links? We do this by setting failure links to nodes that are closer to root. \( \text{Definition: for a node } v \text{ in the tree, } n_v \text{ is the unique node in the tree labeled with } L(v) \text{ of length } L(p(v)) \). This is called the failure link. Initially, \( n_r = r \) for root \( r \) of keyword tree. Also \( n_v = r \) if \( |L(v)| = 1 \). That is, at the root \( L(p(r)) = 0 \), and also when \( L(v) \) has a single symbol then there is no non-trivial proper prefix. Now for a node \( v \) that is \( k+1 \) chars from root with parent \( v' \). \( \text{Note } v' \text{ failure link (to node } n_{v'} \text{) is set (since its distance from root is } k \). Thus, if \( n_{v'} \) has a branch labeled with \( x \) (\( x \) is the character on the edge from \( v' \) to \( v \): i.e. the symbol not in \( v' \)), then clearly that node connected by \( x \) from \( n_{v'} \) is the failure link. You should pause and make sure you see why this is the case. But what if there is not such a branch with \( x \) out of \( n_{v'} \)? This means there is no prefix of the same \( n_{v'} \) plus \( x \). But note that \( L(n_v) \) is a proper suffix of \( L(n_{v'}) \) followed by \( x \). Why? \( L(n_v) \) minus \( x \) can not gets longer than \( L(n_{v'}) \) since that is the longest prefix (without \( x \)). So we just trace the failure link continuous to \( n_{v'} \). And continue until we find a node which has a branch labeled with \( x \). If in the end, there is no such branch labeled with \( x \) found, then the failure link is set back to the root. It is not hard to see this procedure is correct. But how long does it take? \( \text{Claim: the overall time is } O(n) \text{ for all patterns with total length } n. \text{ This seems to be not the case: you are tracing multiple links for each node and that could take long time. The trick is give more detailed analysis.} \)

We consider a single pattern of length \( t \). And we consider each node along the root to this single pattern \( P \) and set \( L(p(v)) \) and failure link \( n_v \) along the way. I will show the time for this pattern is \( O(t) \) and thus the overall time for all patterns is \( O(n) \). The key is to consider how \( L(p(v)) \) (the length of failure link part) changes. Clearly each time \( L(p(v)) \leq L(p(v')) + 1 \). That is, adding one char will at most increase failure link length by 1 (almost trivial). So the increase of \( L(p(v)) \) is at most \( t \). Now, decrease may happen when you trace downwards by multiple links. However, this
can not happen too much since \( lp(v) \geq 0 \). Thus, the total decrease is at most \( O(t) \). Since each tracing of failure links reduces \( lp(v) \) by at least 1, we know we can only trace at most \( O(t) \) links throughout the algorithm.

One last issue: what if there are keyword that is some substring of the another keyword? If this happens, then keyword may just bypass them since the failure links only consider prefix of keywords and if some keyword appears somewhere in the middle it will not notice it. You should pause here and think why this is the case. Remedy: create so-called output links. Output link of a node \( v \) points to patterns that is reachable from \( v \) through failure links. Key: if we reach a node \( v \) during pattern matching, then all the keywords pointed by the failure links must appear in it. Why? Any reachable keywords by failure links must be a suffix of \( L(v) \). Then we can modify the algorithm by also looking for output links and output them if there are output links of \( v \) when we reach \( v \); if such output links are found, we then report their occurrence. Therefore, the overall running time of Aho-Corasick algorithm is \( O(n + m + k) \), where \( k \) is the number of occurrence.

2 Application of AK

The pattern matching with wild-cards problem. Sometimes \( P \) contains unknown (wild-cards). For now, assume each wildcard stands for a single letter (i.e. only mismatch but not indels). Ex: \( P = ab**c* \), and \( T = xabvcchababeax \). Then \( P \) appear twice in \( T \). Now we assume the number of wildcards is constant and then we can apply Aho-Corasick set matching.

Here is the approach:

1. break \( P \) into pieces \( P_1, P_2, \ldots P_k \) s.t. each \( P_i \) has no *. Denote the start position of \( P_i \) in \( P \) as \( l_i \). Example: \( P = ab**c*ab** \), then \( P_1 = ab, P_2 = c, P_3 = ab, l_1 = 1, l_2 = 5, l_3 = 7 \).

2. Using Aho-Corasick find, for each \( P_i \), all starting position of \( P_i \) in \( T \). For each such start position \( j \) of \( P_i \) in \( T \), increase the count \( C \) at \( j - l_i + 1 \) by one (i.e. \( C[j - l_i + 1] \leftarrow C[j - l_i + 1] + 1 \)). Here \( C \) is initialized to all 0. This step means there is a support from \( P_i \) that \( P \) starts at \( j - l_i + 1 \).

3. now it is clear: there is an occurrence of \( P \) at \( p \) iff \( C(p) = k \).

Correctness: almost trivial. You should pause and ensure you see it. Time: \( O(n) \) build keyword tree of \( P \). Time to search for \( P_i \) is \( O(m + z) \), \( m \) is the text length and \( z \) is the number of occurrence of \( P_i \). Let \( k \) be the number of segments no bigger than the number of *. Need to bound \( z \). Claim: \( z \leq km \). Why? Each segment can match at most \( m \); there are \( z \) of them. Here treat each \( P_i \) distinct (because each starts at a specific location), thus only one position of \( C \) is incremented. Also, a cell can be incremented by at most \( k \) times. So the total number of matches is \( km \). So total time is \( O(km) \). If \( k \) is constant, this is linear time \( O(m) \).

A final remark: what if the more general case where can be many * (i.e. \( k \) is constant? I believe there is no linear time algorithm is known for this problem. The best algorithm (dated back to 2002) runs in \( O(m\log n) \) time.