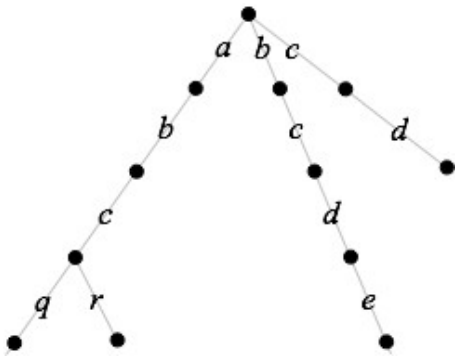
Aho Corasick lecture (+)

Problem: Given z patterns of total size n, find all the places in the text where one or more patterns match.

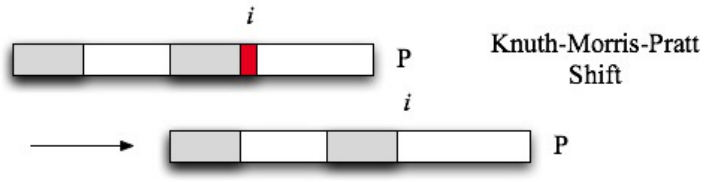
Trivial solutions may be too too expensive. Imagine using a linear time method we've already discussed (e.g. Knuth-Morris-Pratt) that runs in $O(n+m)$, where n is the size of a single pre-processed pattern that we compare to a text of size m. We can just repeat one of those procedures for all z patterns to get a complexity of $O(n+zm)$. What if we had 1,000 patterns and a text size that was 10,000,000 characters? The zm multiplication may be prohibitive in these types of cases, and we can do better.

One immediate way to speed this up is to perform all comparisons simultaneously against a structure that contains *all* the patterns. *Keyword trees* help us do exactly this and can be thought of as a compressed set of patterns that we compare to the text:

The example below puts the set of patterns {*abcq, abcr, bcde, cd*} into a keyword tree.

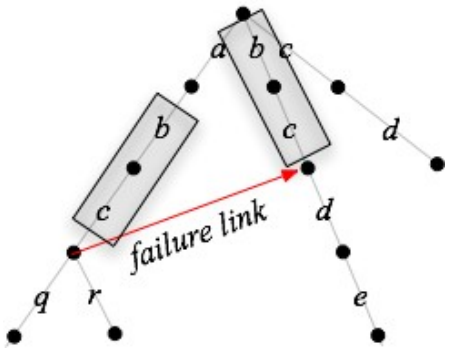


Recall that the Knuth-Morris-Pratt (KMP) algorithm exploits the fact that we can shift our patterns in increments larger than one to reduce our time complexity from quadratic to linear. This method relies on repeated sub-patterns within the pattern. More specifically, while comparing our pattern P to the text, if we notice a mismatch at position i, we shift the pattern as many places as possible so that we can align a maximally matching prefix of P to a suffix of $P[1..i\text{-}1]$. This way we can skip characters we know will not match the pattern while not skipping an occurrence of the pattern:



The Aho-Corasick algorithm is essentially a generalization of KMP to a set of patterns. But things are a little more complicated because instead of shifting a single pattern to the right and jumping to a character we have not yet compared, we shift the entire tree over and we need to jump to an appropriate location in this tree of patterns.

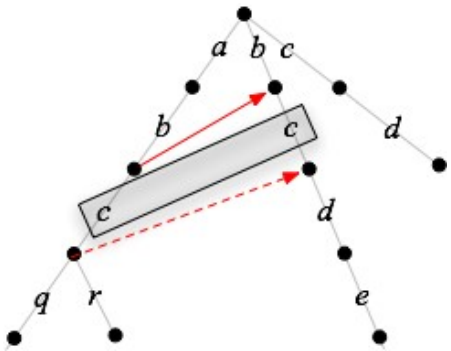
Similarly to KMP, we can encode where we need to jump to, and we point to the right location to the tree via a *failure link*. In our example above, the largest suffix of the node just after ***abc*** that matches a prefix of one of our patterns is the node right after *c* in the pattern ***bc****de*. So in our search for a match, if the next character after ***abc*** (either *q* or *r*) is a mismatch, we can shift our tree to the position in the text that overlaps with ***bc*** but now we can search for an occurrence of the pattern that starts with ***bc***. The failure link points to exactly the location we need to not have to re-compare ***bc***:



The length of the box defines how much we can skip, which is the same length that we use in KMP, there's just a bunch of fancy tree stuff to do.

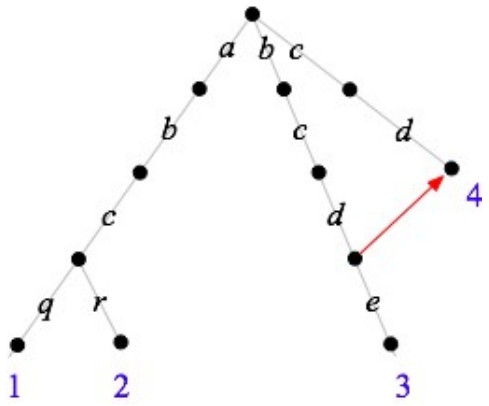
Given these failure links, we don't have to compare against characters we've already dealt with and we end up finding our patterns on the order of the size of the text, *m*. But it would be desirable to have an overall linear processing algorithm and it's not clear yet that we can compute the failure links in linear time.

But we can! The procedure is actually surprisingly similar to KMP. Basically, we use the *parent's* failure link as our previous value (which we assume we have computed). Suppose we wanted to compute the failure link shown above based on failure link after node *b*. Since the character right after *b* is *c* for substring *abc*, then we can look to see if the node for the parent's failure link also contains *c* as a child. If it does, then we simply set the current failure link to the node right after that child:



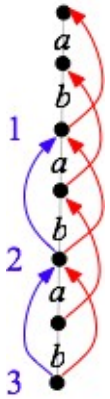
Of course the *c*'s may not mach, so in the case the characters do not match, like in KMP, we traverse the failure link of the parent's failure link, and so on, until we do get a match. The proof for the fact that this is entire algorithm runs in linear time (even with the parent recursion) is like KMP as well. The basic idea for both KMP and Aho-Corasick is that there is a single variable in the algorithm that corresponds to the length of matching substrings. *Over the entire algorithm's lifetime*, this variable can increase by at most around *n* values and as it's decrementing in the parent recursion, it can never go below 0, thus the total amount it can decrease is also at most around *n* values.

One issue we haven't addressed yet is that if one of our patterns is a proper substring of another pattern, then we need to identify it as well. The failure links help us in this task:



The basic idea is that if we traverse the failure links to a leaf node that corresponds to a pattern (in this case pattern 4), then we report an occurrence of a sub-pattern as well.

Following these failure links can be expensive, so an optimization is to store what are called *output* links. Let's consider the pattern set {*ab, abab, ababab*}:



The links in blue represent these output links. We still have to recurse through those, but every time we do we have a match. The text discusses how you can build these in links a straightforward way without affecting the time-complexity during preprocessing. As discussed, we could have a memory tradeoff and store the matching sub-patterns during pre-processing, but this could be quite expensive.

So for pre-processing, we have $O(n)$ time. Running the algorithm $O(m)$ time in addition to traversing the output links, which always result in matches. If the total number of matches is k, then the total running time is $O(n+m+k)$. Note that this is data-dependent, so if we have text that has a crazy-number of matches based on our pattern set, then the performance could be terrible.

This method can be applied in a fairly straightforward way to match wildcards. Suppose our wildcard pattern is: *ab*abc*abcd*, the basic idea is to take all substrings that aren't wildcards and record where they need to occur in the pattern. For instance, the patterns *ab, abc, abcd* are located at indices 1,4,8. We now feed all these patterns into our favorite exact set matching algorithm at the moment: Aho-Corasick. For some character *i* in the text, to get a wildcard match, we just need Aho-Corasick to report occurrences of *all* three patterns at their appropriate indices, and we have a match!

Problem: Find the longest common substring of two strings.

This is our primer into suffix trees! The idea is to take all possible suffixes of one string and create a keyword tree out of it. For instance, the take a *banana*$ has suffixes:

banana$
anana$
nana$
ana$
na$
a$
$

This tree helps us solve the problem, and we can construct the tree in *linear* time! (Supposedly this is a rare case where Knuth was proven wrong ...)