Data structures for strings*

We consider three generalizations of the string matching problem: searching with a collection of patterns instead of just one, allowing wild-cards in the pattern, and searching for common substrings. We present two data structures: the keyword tree, which solves the first two generalizations, and the suffix tree, which solves the first generalization and the third.

**String matching with a collection of patterns.** We generalize string matching to a collection of patterns $P_1, P_2, \ldots, P_k$. For each pattern, we ask whether or not it occurs as a substring of a given text $T[1..n]$. It is convenient to assume that the collection is prefix-free, that is, no pattern is a prefix of any other pattern, although this is not necessary but leads to complications we would like to avoid. With $m_j$ the length of pattern $P_j$, we let $m = \sum_j m_j$ be the total length of the patterns. Assuming the alphabet has constant size, $c$, we can store the patterns in a tree in which every node has degree at most $c$. Each edge is labeled with a character, and the paths from the root to the leaves spell out the patterns. We refer to this as the **keyword tree** of the patterns, assuming the outgoing edges of a node are sorted according to an ordering of the alphabet, as in Figure 1. It is easy to construct the tree in time $O(m)$. Each node $\mu$ corresponds to a prefix $L(\mu)$ of a pattern. If $\mu$ is a leaf, $L(\mu)$ is an entire pattern and we

![Figure 1: The keyword tree for the patterns “other”, “potato”, “tattoo”, “theater”.](image-url)

*These notes are extracted from the lecture notes on “Design and Analysis of Algorithms”, Fall 2005 by Herbert Edelsbrunner.
store its index $j(\mu)$ at $\mu$. To search with the text, we traverse a longest common prefix for each suffix $T[s + 1..n]$, which takes time $O(nm)$.

**Failure function for keyword trees.** We improve the running time by adapting the idea of a failure function to keyword trees. For each node $\mu$, let $\ell(\mu)$ be the length of the longest suffix of $L(\mu)$ that is a prefix of some pattern. We store a link $Fail[\mu]$ from $\mu$ to the node $\nu$ for which $L(\nu)$ is this prefix. This is illustrated in Figure 2. For convenience, we let $Fail[\mu]$ point to the root if $\ell(\mu) = 0$. The search algorithm maintains a current node $\mu$, which it initializes to the root, $g$, of the tree. It also uses two pointers into the text array: $s$ precedes the starting position of the substring currently matched with the pattern, and $i$ is the position currently tested.

$s = 0; \ i = 1; \ \mu = g;$
repeat
while $\mu$ has child $\nu$ with edge-label $T[i]$ do
if $\nu$ is leaf then print “$P_{j(\nu)} = T[s + 1..i]$”
else $\mu = \nu; \ i = i + 1$
edif
endwhile;
$s = i - \ell(\mu); \ \mu = Fail[\mu]$
until $s > n$.

It is convenient to assume $T[n + 1]$ stores a special end-symbol that avoids patterns are compared with entries beyond the end of the text array. We omit the construction of the lengths $\ell(\mu)$ and the failure links $Fail[\mu]$, which takes time $O(m)$. The running time of the search algorithm is $O(n + m)$ because every step either advances the position in the text and simultaneously increases the depth of the node in the tree, or it reduces the depth, which cannot be done more often than increasing the depth.
Sting matching with wild-cards. We can use keyword trees to solve the string matching problem in which we allow for wild-cards, *, in the pattern. The wild-card is a special character that matches any (single) character in the text. For example, the pattern \( \text{ABRA}******\text{A} \) occurs twice in \( \text{HOCUSPOCUSABRACADABRA} \), starting after positions 10 and 13. Instead of concurrently following different branches for each wild-card, we search for each maximal substring without wild-cards, and we record each match at the text position preceding the first (full) pattern position. We use an integer array \( Q[0..n-1] \) to record the matches. Initially, \( Q \) is all zero. Let \( P_1, P_2, \ldots, P_k \) be the maximal substrings of the pattern that do not contain any wild-cards. For each \( P_j \), we let \( p_j \) be the position preceding \( P_j \) in the pattern. For example, for \( P = \text{ABRA}******\text{A} \) we have \( P_1 = \text{ABRA} \) with \( p_1 = 0 \) and \( P_2 = \text{A} \) with \( p_2 = 10 \). We store the \( P_j \) in a keyword tree with added failure links.

**Step 1.** For each \( 1 \leq j \leq k \), find all starting positions \( s + 1 \) of \( P_j \) in \( T \) and increment \( Q[s - p_j] \) by one, provided \( s \geq p_j \).

**Step 2.** Scan \( Q \) and report every position \( \ell \) with \( Q[\ell] = k \).

In our example, \( P_1 = \text{ABRA} \) increments \( Q \) for indices 10, 13, and 20, and \( P_2 = \text{A} \) increments \( Q \) at indices 0, 3, 6, 8, 10, and 13. There are therefore two occurrences of the pattern in the text, the first at \( T[11..21] = \text{ABRABRACADA} \) and the second at \( T[14..24] = \text{ABRACADABRA} \). The total running time is \( O(n + m) \).

**Suffix trees.** The string matching algorithms we have discussed so far preprocess the pattern, which is the smaller of the two strings, and use the obtained structure to find matches in time proportional to the length of the text. We now turn things around and preprocess the text. Specifically, we take the collection of suffixes, \( T_s = T[s+1..n] \), and construct the keyword tree for \( T_0, T_1, \ldots, T_n \). To avoid complications, we enforce the prefix-free property by appending the special character \( $ \) to each \( T_s \). We thus get a bijection between the leaves of the tree and the suffixes of the text. In each leaf, we record the position preceding the starting position of the corresponding suffix. Finally, we remove each non-branching internal node, merging its two edges into one and concatenating their labels. We thus obtain the suffix tree of \( T \), illustrated in Figure 3. We note that because every internal node has two or more children, the size of the

![Figure 3: The suffix tree of the text mississippi.](image)

...tree is linear in the length of the text. Indeed, there are \( n + 1 \) leaves and therefore at
most $n$ internal nodes and at most $2n$ edges. An edge-label can be an arbitrarily long string, but we can store it using only two integers giving the first and last positions in the text. We also note that any two edges connecting a node with its children are labeled by strings that begin with different characters.

It is straightforward to construct the suffix tree in time $O(n^2)$, simply by adding the suffixes one at a time. It is more difficult but possible to construct it in time $O(n)$. One strategy is to read the text from front to back, and for each new character to expand all suffixed by one and start one new suffix. The details are complicated and omitted.

**String matching with suffix trees.** Given the suffix tree for text $T$, we can determine whether or not the pattern $P$ is a substring of $T$ be traversing a single path.

**Case 1.** We exhaust $P$ and thus find a suffix of $T$ that has $P$ as a prefix.

**Case 2.** We could not exhaust $P$ implying that $P$ is not a substring of $T$.

The time to search in the tree is $O(m)$. In Case 1, we have an internal node whose path from the root spells out $P$, plus possibly a few additional characters at the end, if we used only a portion of the last edge’s label. To find all occurrences of $P$, we can traverse the subtree of this node and report the starting positions of the suffixed stored at the leaves. Since each internal node has two or more children, this takes time linear in the number of occurrences found.

Using the linear-time algorithm for constructing the suffix tree as a preprocessing step, we can solve the string matching problem in time $O(n + m)$, which is the same as for the Knuth-Morris-Pratt algorithm. An advantage of using the suffix tree is that we can search with many patterns, without paying again for the length of the text. The performance is the same as that of the keyword tree, but now we do not have to know the patterns in advance. A disadvantage of using the suffix tree is the extra memory we need for storing the tree.

**Longest common substrings.** A classic problem in string analysis is to find the longest substring common to two given strings, $T_1$ and $T_2$. For example, if $T_1 =$ australi and $T_2 =$ australi, then the longest common substring is i and. To find it, we construct the suffix tree for both texts, representing the suffixes of $T_1$ and of $T_2$, each by a path from the root to a leaf. Each leaf represents a suffix of one string or of both. We mark each internal node $\mu$ with “1” if at least one of the leaves in its subtree represents a suffix of $T_1$. Similarly, we mark $\mu$ with “2” if at least one of the leaves in its subtree represents a suffix of $T_2$. If an internal node has both marks, then its path spells out the prefix of a suffix of $T_1$ as well as a suffix of $T_2$. In other words, it spells out a substring of both. Call the length of this substring the string-depth of $\mu$. To find the longest common substring, we just need to find the internal node with maximum string-depth that has both marks. We summarize the algorithm:

**Step 1.** Construct the suffix tree for $T_1$ and $T_2$.

**Step 2.** Mark internal nodes and determine their string-depths.

**Step 3.** Return the node with maximum string-depth that has marks for both strings.
Letting $n_i$ be the length of $T_i$, Step 1 constructs the suffix tree in time $O(n_1 + n_2)$. The linear-time algorithm mentioned above extends to the case of two (or more) strings, so we can construct the suffix tree of $T_1$ and $T_2$ in time $O(n_1 + n_2)$. The size of the tree is $O(n_1 + n_2)$, and Step 2 marks the internal nodes and computes their string-depths in the same amount of time. Finally, Step 3 find the right node in the same amount of time. In summary, we have an algorithm that determines the longest common substring in time linear in the total length of the two strings.

**Suffix arrays.** Given a text $T[1..n]$, the suffix array, $Pos[0..n]$, records the lexicographic order of the $n+1$ suffixes. Specifically, the suffix $T_{Pos[i]} = T[Pos[i] + 1..n]$ is lexicographically smaller than $T_{Pos[i+1]}$. As an example, consider $T = \text{mississippi}$ and order its suffixes lexicographically:

```
11 : 
10 : i
 7 : ippi
 4 : issippi
 1 : ississippi
 0 : mississippi
 9 : pi
 8 : ppi
 6 : sippi
 3 : sississippi
 5 : ssippi
 2 : ssissippi
```

The suffix array stores 12 integers, one more than the length of the text, which for $T = \text{mississippi}$ are 11, 10, 7, 4, 1, 0, 9, 8, 6, 3, 5, 2. We can construct the suffix array from the suffix tree by in-order traversal, but for that we have to interpret $S$ lexicographically smaller than all other characters in the alphabet. A key property of the array is that it groups suffixes with common prefixes together in contiguous positions. We can therefore use binary search to find all suffixes that contain a given pattern $P$. This takes time $O(m \log n)$, which is not quite as fast as the suffix tree itself, but the array takes less memory. Also, there are methods that can speed up the search to time $O(m + \log n)$, which as fast as the suffix tree unless the size of the text exceeds 2 to the size of the pattern.