CHAPTER 2

STRING AND TEXT ALGORITHMS

2.1 Suffix Tries and Trees

In this section, we will look at suffix tries and trees, and how they can be used to efficiently perform different string operations. We begin by defining some terminologies:

- an *alphabet* consists of all letters that a string can be composed of, e.g. ASCII. It is denoted by Σ.
- a *string* is a word on the alphabet.
- the *length* of a string S is the number of letters it has.
- for a string S, we use $S[i]$ to denote the i-th character in the string.
- for a string S, we use $S[i:j]$ to denote the substring of S from position i to j (both included).
- for strings S and T, their *concatenation* is denoted by ST.
- the *empty string* is denoted by ϵ. It has length 0.
- for a string S, another string U is a *substring* of T if $S = TUV$, for strings T and V.
- for two strings S and T, we say that U is a *common substring* if it is a substring of both S and T.
- for a string S, a *subsequence* of S is composed of characters in S (in the given order), but it need not be a continuous block like a substring. We can delete characters from S to get a subsequence.
- for strings S and T, we say that U is a *common subsequence* if it is a subsequence of both S and T.
- for a string S, a *prefix* is a substring starting from the start.
- for a string S, a *suffix* is a substring ending at the end.
- for an alphabet Σ, the set Σ^* is composed of all strings on Σ, including the empty string ϵ.
- a *leaf node* in a tree has no children.
- a *branch node* in a tree has at least one child.
- a *unary node* in a tree has precisely one child.
- a *binary node* in a tree has precisely two children.

We have previously seen suffix tries used to search hundreds of substrings in a single long text. We can do this using KMP or BM, but these are linear in terms of the long text during each computation. We can use suffix tries to make the algorithm to process the long text just once, and linear in terms of the number of strings we have to search. This is because the trie pre-processes the text and creates an index for it. Because the text is typically much longer than the substrings, this is much faster in practice.

Longest repeated and common substring

Given a string S, we want to compute the longest repeated substring in S. A naive solution for this will loop through the starting position of the first repeat and the second repeat, and check whether there is a match of some length- this takes $O(n^3)$ time. We can use a dynamic programming algorithm to make the algorithm more efficient.

In the algorithm, we store the characters of the string as both rows and columns. If there is a match at a cell, we make the entry 1; otherwise, the entry is 0. The longest common substring corresponds to the greatest sum we can produce when we traverse the long diagonals. The following example illustrates this algorithm to find the longest repeated substring of `ababa`:

	a	b	a	b	a
a	-	0	1	0	1
b	-	-	0	1	0
a	-	-	-	0	1
b	-	-	-	-	0
a	-	-	-	-	-

We only compute the entries on the right side of the diagonal as the values are symmetric. There is always a match at the diagonal- we are not interested in that. In this case, the longest repeated substring is `aba`. Note that the value `a` is shared between the two substrings- this is allowed because the index of the matching `a` is still different.

This algorithm has complexity $O(n^2)$ in both time and space.

We can easily modify this dynamic programming algorithm to compute the longest common substring of two strings. The following example illustrates this.

	a	b	c	a	b
b	0	1	0	0	1
b	0	1	0	0	1
c	0	0	1	0	0
a	1	0	0	1	0
a	1	0	0	1	0

In this case, we need to compute both sides of the diagonal, and the diagonal, since the two strings are assumed to be different. Here, the longest common substring is `bca`.

This algorithm has complexity $O(mn)$ in both time and space, where m and n are the lengths of the two strings.

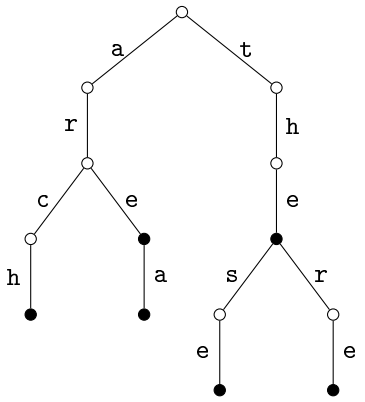
Suffix Tries

We will now improve the time (and space) complexity of these algorithms using suffix tries. For a string S, the suffix tree of S stores all the suffixes in S including the entire string. Using this representation, we can compute the longest repeated substring in $O(n)$ time and longest common substring in $O(m+n)$ time. Moreover, we can search multiple strings (whose sum of length is r) in $O(n+r)$ time.

A trie is a branching tree that is used to store strings in an alphabet. It has the following properties:

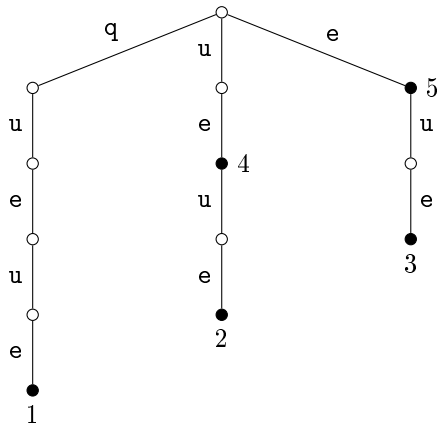
- a trie has a root vertex;
- every edge in the trie corresponds to a character from the alphabet;
- no two children of a branch node have the same edge label;
- every node corresponds to a string in the alphabet, which is given by traversing the tree from the root to the given node;
- every node is marked in a way to recognise whether it corresponds to one of the given words.

An example of the trie on the words- `arch`, `are`, `area`, `the`, `there` and `these` is given below:

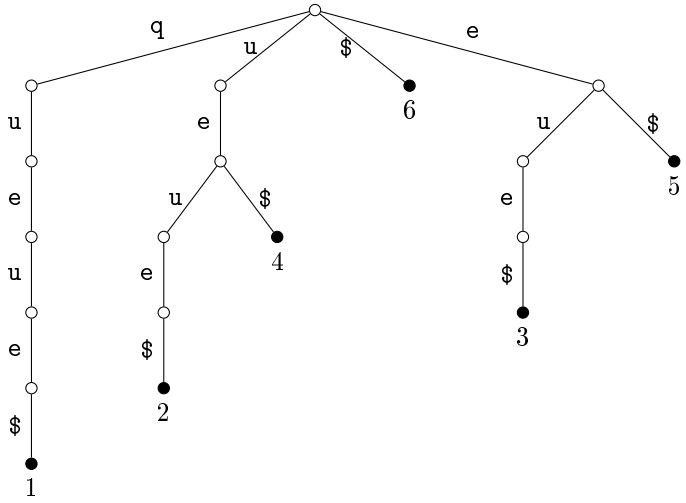


We mark by black any node that corresponds to a string.

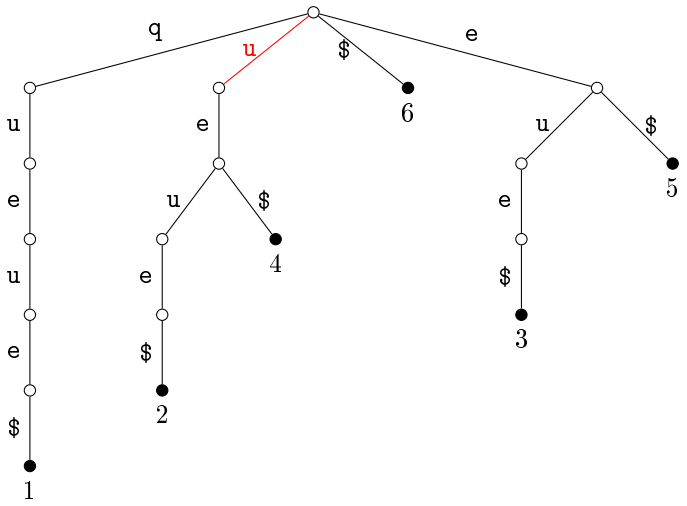
A suffix trie is a trie that stores all the suffixes of a given string. For example, the suffix trie of `queue` is given below.



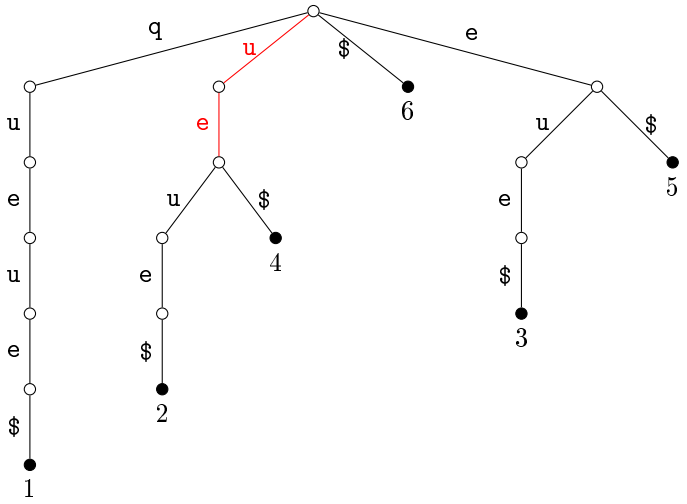
For a suffix trie, we also want each suffix to end at a leaf node. This is not the case in the trie above- the suffixes `ue` and `e` end at branch nodes. This is because there is a suffix (`ue`) that is a prefix of another one (`ueue`). To fix this issue, we add a unique termination symbol `$` that is not present in the alphabet. This ensures that no suffix is a prefix of another suffix. In this case, the suffix trie for `queue` is the following:



We will now use the suffix trie above to find `ue`. We use the suffix trie to descend from the root to the branch corresponding to the character `u`.



We then descend from this branch to the one corresponding to the character `e`.



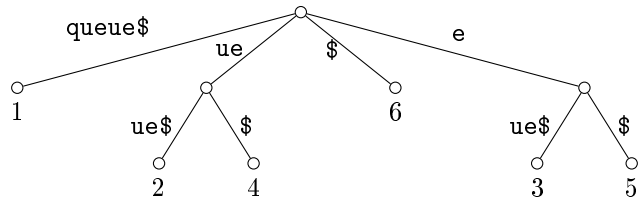
We have now completely matched the substring. This means that the substring occurs at indices 2 and 4. Using the suffix trie, we were able to find all the occurrences in linear time with respect to the substring (assuming that the alphabet has constant size). If we had tried to match `ux` instead, we would not have been able to find a branch for the second letter. That implies that there is no match in the string.

We can use the suffix trie to also find the longest repeated substring. This corresponds to finding the branch node with the greatest string depth. A branch node is a prefix of at least 2 suffixes, and so represents a repeated substring. For the string `queue`, the repeated substrings, as seen in the trie, are `e` and `ue`. Hence, the longest repeated substring is `ue`, with length 2, with starting indices 2 and 4.

To build a suffix trie, we repeated insert all the suffixes to the trie. To insert the suffix i, it takes $O(n-i)$ time. So, overall, it takes $O(n^2)$ time and space. This is not tractable for long texts. To improve this, we make use of suffix trees.

Suffix Trees

Suffix trees are very similar to suffix tries, but we have compressed unary branch nodes. So, the suffix tree for `queue` is the following:



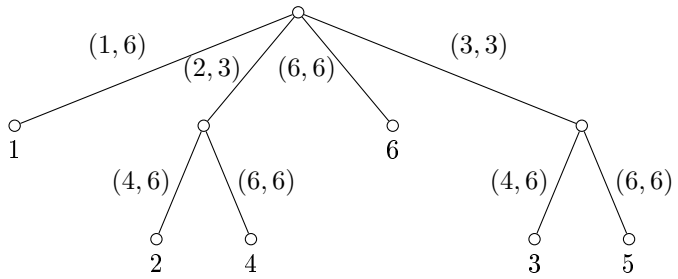
This is a much more efficient representation since we do not need to store a node for each letter. In this case, the number of nodes is $O(n)$, whereas for the suffix trie, it was $O(n^2)$.

Now, we define the suffix tree for some string S. We first append \$ to the string and construct all the suffixes that satisfy the following properties:

- each edge has a string label;
- all branch nodes have at most 2 children;
- no two children of a branch node have two edges that started with the same character;
- there is a bijection between the leaf nodes and the suffixes, i.e. we can descend from the root to a leaf to recover the suffix. The entire label (from root to a node) is called its *path label*.

In a suffix tree, there are $n+1$ suffixes, so $n+1$ leaf nodes. Also, the number of branch nodes is at most n. To see this, assume that we have x edges, n leaf nodes and b branches. Then, $x = b + n - 1$- we have an edge going to each branch node and leaf node, except for the root. Moreover, since every branch node has at least 2 children, $x \geq 2b$. Hence, $b \leq n - 1$. Hence, there are $O(n)$ nodes in total. This is an improvement from suffix tries, which have $O(n^2)$ nodes.

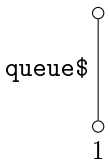
However, the suffix tree still has $O(n^2)$ space complexity- we need linear space to store the entire suffix. To mitigate this, we just store the index of the start and the end of the substring in the text. So, the suffix tree for `queue` is the following:



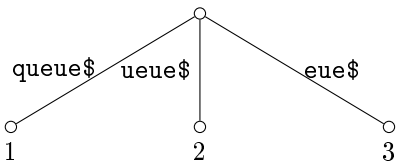
Each label now takes $O(1)$ space, so the total complexity is $O(n)$ space. We call this representation the *suffix tree with short edge labels*, and the one above as the *suffix tree with full edge labels*.

One way to construct the suffix tree with short edge labels is to create the suffix trie, use it to construct the suffix tree with full edge labels by collapsing unary branches, and then replace the full edge labels with short edge labels. This is not the most efficient approach since we do need $O(n^2)$ space during the algorithm.

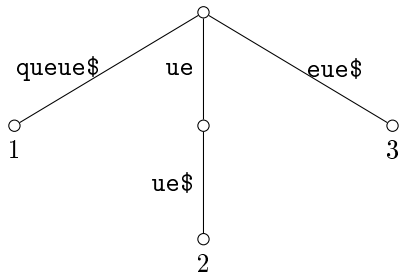
A more efficient approach would involve directly constructing the suffix tree with short edge labels. We illustrate this with an example, by constructing the suffix tree with full edge labels for `queue`. We will use full edge labels for illustration purposes; the algorithm would instead make use of the short edge labels. We start by inserting the suffix `queue$`.



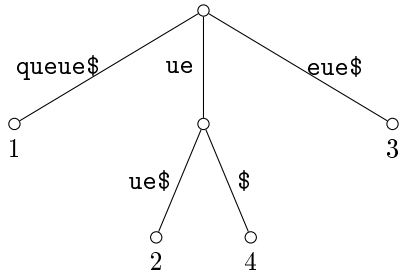
We insert the suffix by branching from the root. We can do the same thing when inserting the next two suffixes- `ueue$` and `eue$`.



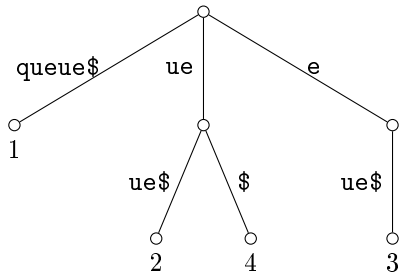
At this point, the next suffix is `ue$`. We already have a node starting with `ue`, but this is not a complete match (i.e. the branch has a longer label). So, we perform an *edge-split* first to ensure we can branch at the right position.



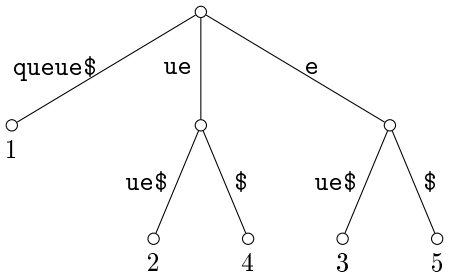
We can now insert `ue$` as a child of the branch `ue`.



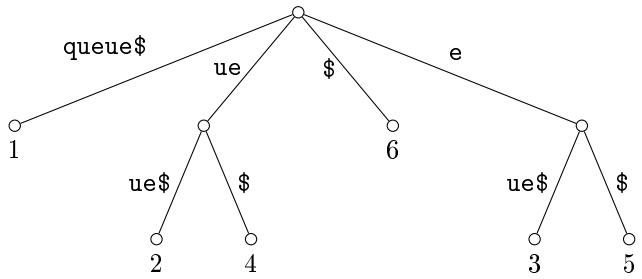
We have to do the same thing when inserting `e$`- we edge-split `eue$`.



We can now insert `e$` to the tree.



Next, we insert `$`.



The suffix tree is now complete.

This algorithm takes $O(n)$ space complexity, but has worst-case complexity $O(n^2)$, e.g. if we have to edge-split every time. The average case is $O(n \log n)$. There are more efficient algorithms that require $O(n)$ time complexity.

We will now use the suffix tree for different string algorithms. If we have a string S and a substring T to search in the string, we construct a tree on S, and then match each letter in T with the corresponding node. If we cannot match T completely, then the string is not present. If it is present, we can return the index of the match by using the branch node we are at. This algorithm is very similar to the one for suffix tries we saw above. The time taken for this algorithm is linear with respect to the substring. We can also find all occurrences using this approach.

Now, we try to find the longest repeated substring in a text S. We can construct the suffix tree for S in $O(n)$ time. We then traverse the tree to find the branch node with the greatest string depth. A branch node corresponds to a common substring, and the string depth keeps track of the length of the common substring. We can traverse in $O(n)$ time, so the entire algorithm is $O(n)$.

Next, we can find a longest common substring of texts S and T. To do so, we construct the suffix tree for $S\#T\$$, where $\#$ and $\$$ are symbols not in Σ. This is called the *generalised suffix tree*. We then traverse the tree in a similar way as the longest repeated substring, but we are looking for a common branch node with the greatest string depth. A common branch is one that has leaf nodes corresponding to suffix starting in both S and T.

Now, we will look at a method to identify common branch nodes. For each branch node v, we keep track of two values- whether it has a leaf node with suffix value corresponding to a string in S ($b_1(v)$), and whether it has a leaf node with suffix value corresponding to a string in T ($b_2(v)$). Then, v is a branch node if and only if $b_1(v) \wedge b_2(v)$. We can compute the $b_i(v)$ values as follows:

- if v is a leaf node, then we can check the value of the suffix to compute $b_i(v)$- $b_1(v)$ is `true` if $1 \leq j \leq m$ and $b_2(v)$ is `true` if $m+2 \leq j \leq m+n+1$, where m and n are the lengths of S and T respectively;
- instead, if v is a branch node, then $b_i(v) = b_i(w_1) \wedge b_i(w_2) \wedge \cdots \wedge b_i(w_n)$, where $w_1, w_2, \ldots, w_n$ are the descendant leaf nodes of v.

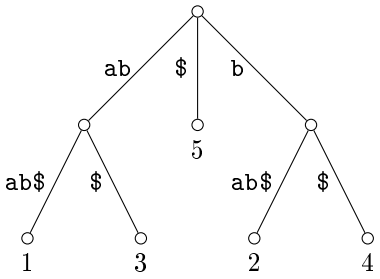
So, the algorithm for longest common substring is the following:

- build the generalised suffix tree;

- calculate the values $b_i(v)$ for every node v;
- find the common branch node with maximum string depth.

The last 2 steps can be combined. Each step takes $O(m+n)$ time, so the algorithm overall takes $O(m+n)$ time.

We will now illustrate that the character # is required in order to correctly output the result. So, assume that the two strings are `a` and `bab`. Then, the suffix tree for $ST\$$ is the following:



Here, the common branch node with the greatest depth is `ab`. However, it is not a common substring of `a` and `bab`.