String Data Structures: Prefix Tries and Suffix Trees

DANIEL ANDERSON [1]

Today, huge amounts of the world's data is in the form of text data or data that can be interpreted as textual data. From classic literature, your favourite algorithms and data structures textbook to DNA sequences, text data is everywhere, and it needs to be stored, processed and analysed. We will introduce and study one special data structure for working with text strings, the suffix tree, as well as a related data structure, the prefix trie / retrieval tree data structure which allows for fast storage and lookup of strings.

Summary: Prefix Tries and Suffix Trees

In this lecture, we cover:

- Retrieval Trees / Prefix Tries
- Suffix trees
- Applications of suffix trees

Recommended Resources: Prefix Tries and Suffix Trees

- Weiss, Data Structures and Algorithm Analysis, Section 12.4.2
- `https://visualgo.net/suffixtree` - Visualisation of suffix trees
- `https://youtu.be/NinWEPPrkDQ` - MIT 6.851 lecture on string data structures (advanced)
- `http://www.allisons.org/ll/AlgDS/Tree/Suffix/`

Some String Terminology

Recall that a *string* `S[1..N]` is a sequence of characters, ie. some textual data.

1. A *substring* of `S` is a string `S[i..j]` where $1 \leq i \leq j \leq N$.
2. A *prefix* of `S` is a substring `S[1..j]` where $1 \leq j \leq N$.
3. A *suffix* of `S` is a substring `S[i..N]` where $1 \leq i \leq N$.

A useful observation to make is that a substring of `S` is always a prefix of some suffix of `S`, or equivalently, that a substring is always a suffix of some prefix of `S`.

The Prefix Trie / Retrieval Tree Data Structure

A retrieval tree or prefix trie is a data structure that stores a set of strings arranged in a tree such that words with a shared prefix are contained within the same subtree.

[1] FACULTY OF INFORMATION TECHNOLOGY, MONASH UNIVERSITY: `daniel.anderson@monash.edu`. These notes are based on lecture slides by Arun Konagurthu, the textbook by CLRS, some video lectures from MIT OpenCourseware, and many discussions with students.

Key Ideas: Prefix Trie

- The prefix trie is a (not necessarily binary) tree structure
- Each **edge** (not vertex) of the tree is labelled with a character
- Nodes can be marked as corresponding to the ends of words
- Each string stored in the trie corresponds to some path from the root to a marked node in the tree
- A path from the root to an internal node of the tree corresponds to a common prefix of one or more strings in the trie

Prefix tries can be implemented in a variety of ways, the most important decision is how to index the children of a particular node. There are a lot of strategies, we will briefly think about three of them. We will denote the length of a query string by N, the total length of all words in the trie by M, and the size of the alphabet by Σ.

- Use an array to store a child pointer for each character in the alphabet:

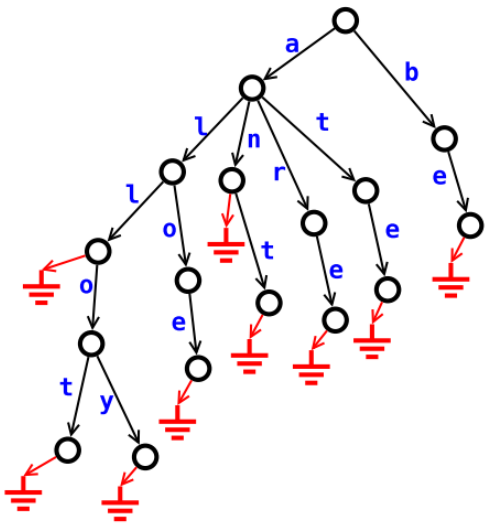
  This method allows us to perform lookup and insert in $O(N)$ time since we can follow each child pointer in constant time. However, the space requirement is quite high, since we are storing a lot of pointers to null children. Specifically, we will use $O(\Sigma M)$ space in the worse case, since every node has a pointer for every character in the alphabet.

- Use a balanced binary search tree for storing child pointers:

  This method uses the minimal amount of space, as we only need to store pointers to children that actually exist. For large alphabets, this is advantageous. However, we lose in lookup time since it now requires up to $O(N \log(\Sigma))$ time per lookup since at each node we must search a BST to find the child pointer. The total space requirement however is minimal, at just $O(M)$.

- Use a hashtable for storing child pointers:

  This method allows us to only store pointers to the children that exist, hence we will only use the minimal $O(M)$ space. It also allows lookup in $O(N)$ expected time, since at each node we perform a hashtable lookup in expected constant time. This method seems to be superior to the first two since it has optimal speed and optimal space usage. However, this implementation is less versatile, and prohibits us from doing some more advanced tricks with the trie. For example, you might want to implement the ability to lookup the alphabetically closest word to a given word. This is possible with the first two approaches, but not (efficiently) with a hashtable since there is no way in a hashtable to quickly lookup the alphabetically nearest key.



An example of a prefix trie containing the strings `be`, `ant`, `alloy`, `ate`, `are`, `aloe`, `an`, `allot`, `all`. The red markers indicate the ends of strings.

In some descriptions of tries, it is common to append a special character to each word to indicate the end. This special character is usually denoted by "$". This ensures that a node is a leaf node if and only if it is the end of a word, which simplifies some problems.

Applications of Tries

Storage and lookup of words

Prefix tries are an effective alternative to standard dictionary data structures such as binary search trees and hashtables when the keys being stored are specifically words from some fixed alphabet. Assuming a constant-size alphabet, building a prefix trie takes time proportional to the total length of all of the words that are going to be inserted, and lookup takes time proportional to the length of the key being searched, which is optimal in both cases.

Algorithm: Insertion into a Prefix Trie

```
1:  function INSERT(S[1..M])
2:      Set node = root
3:      for each character c in S[1..M] do
4:          if node has an edge for character c then
5:              node = node.get_child(c)
6:          else
7:              node = node.create_child(c)
8:          end if
9:      end for
10:     if node is the end of a word then
11:         ▷ The string S was already in the trie
12:     else
13:         node.mark_as_end()
14:     end if
15: end function
```

Algorithm: Lookup in a Prefix Trie

```
1:  function LOOKUP(S[1..M])
2:      Set node = root
3:      for each character c in S[1..M] do
4:          if node has an edge for character c then
5:              node = node.get_child(c)
6:          else
7:              return false
8:          end if
9:      end for
10:     if node is the end of a word then
11:         return true
12:     else
13:         return false
14:     end if
15: end function
```

Note that the lookup algorithm could easily be modified to detect if a word being searched was not necessarily in the trie, but a prefix of a word in the trie by simply returning **true** before checking whether the final node on the path taken was marked as the end of a word.

Sorting a list of strings

We can use a prefix trie to quickly sort a list of strings coming from a fixed size alphabet. Simply insert all of the strings into a prefix trie, and then traverse the trie in lexicographical order.

The complexity of this algorithm is $O(\alpha M)$ where α is the size of the alphabet and M is the total number of characters in all of the input strings. Assuming that all of the input strings are roughly the same length, the complexity of sorting the strings using a typical fast sorting algorithm would instead be

$$N \log(N) \times \left(\frac{M}{N}\right) = M \log(N),$$

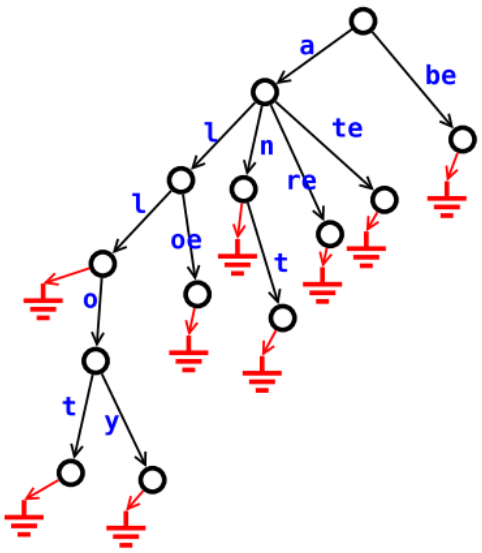
where N is the number of words in the list being sorted. We can therefore see that if the alphabet size is fixed and small, with $\alpha \ll \log(N)$, that the prefix trie method will be asymptotically faster. An interesting observation to make is that sorting strings using a prefix trie is actually exactly the same as using radix sort on the strings! The only difference is that now we are doing a most-significant-digit (MSD) radix sort instead of an LSD radix sort like we covered earlier.

Algorithm: Sort Strings using a Prefix Trie

```
1:  function SORT_STRINGS(list[1..N])
2:      for each string S in list do
3:          insert(S)
4:      end for
5:      traverse(root, "")
6:  end function
7:
8:  function TRAVERSE(node, cur_string)
9:      if node is the end of a word then
10:         print(cur_string)
11:     end if
12:     for each character c in the alphabet in order do
13:         if node has an edge with character c then
14:             cur_string.append(c)
15:             traverse(node.get_child(c), cur_string)
16:             cur_string.pop_back()
17:         end if
18:     end for
19: end function
```

Compact Tries

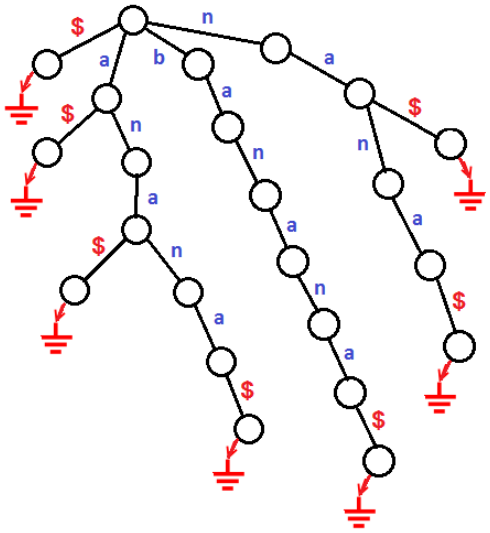
One major drawback to tries that we have observed is that they tend to waste a lot of memory, particularly if they contain long non-branching paths (paths where each node only has one child). In cases like these, we can make prefix tries more efficient by combining the edges along a non-branching path into a single edge. The *Radix Trie* and the *PATRICIA Trie* (not examinable) are examples of compact versions of the prefix trie data structure.



A compact trie containing the strings `be`, `ant`, `alloy`, `ate`, `are`, `aloe`, `an`, `allot`, `all`. The red marks indicate the ends of strings.

Suffix Trees

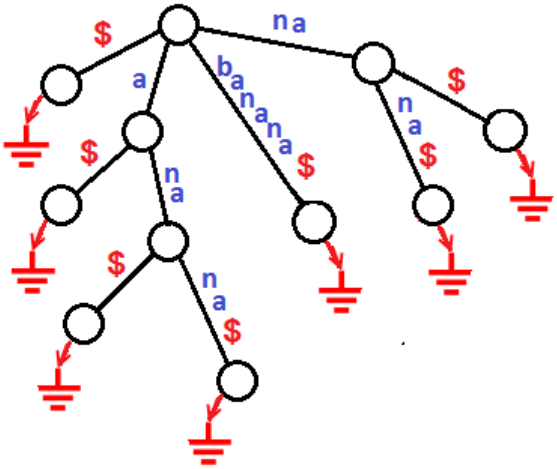
Recall that a suffix of a string is any substring that contains the final character of the string (and in some definitions, the empty suffix containing no characters.) We could take all of the suffixes of a string and insert them into a prefix trie, which would give us a data structure called the *suffix trie* of a string. In applications involving string suffixes, it is often useful to remember explicitly where the end of the string is, so we will use the trick where we append a special character to the end of the string to remind us. We denote this special character by "$".



A suffix trie for the string "banana$".

With the suffix trie, we can easily solve the pattern matching problem for any input string `P[1..M]` in $O(M)$ time, by searching the suffix trie and checking whether `P` is a prefix in the trie (remember that a substring is necessarily a prefix of some suffix, this is why the prefix trie of suffixes = the suffix trie works!)

Of course, we remember that prefix tries are not the most memory efficient data structure, and realise that the suffix trie will consequently take $O(N^2)$ space to store. We can improve on this by instead storing the suffixes in a compact trie. This will reduce the number of nodes in the tree from $O(N^2)$ to just $O(N)$. This compacted version of the suffix trie is called the suffix tree of the string.



A suffix tree for the string "banana$".

It is important to note that if we store all of the edge labels as strings then the memory used by the suffix tree will still be $O(N^2)$. In order to mitigate this and bring the memory required down to $O(N)$, we instead simply refer to each label via its position as a substring in the original string. For example, the substring "na" in "banana$" would simply be represented as $[3,2]$, meaning that it is a substring beginning at position 3 in the string and having length 2 within the string "banana$".

Building a suffix tree

The naive approach

The simplest way to build a suffix tree is to build the suffix trie in $O(N^2)$ and then compress it into a suffix tree. This is simple to implement but is unfortunately useless in practice due to the poor time complexity.

Converting a suffix array into a suffix tree (NOT EXAMINABLE)

In the next lecture, we will learn about another data structure, the suffix array. Interestingly, suffix arrays and suffix trees actually encode precisely the same information. That is, given the suffix array of a particular string and the string itself, you can determine the suffix tree, and vice versa. The idea is relatively straight forward and involves first computing the longest common prefix (LCP) array from the suffix array.

Given the LCP array, one can easily determine the first level of the suffix tree by observing that the subtrees of the root node are precisely the suffixes with no common prefix, ie. they correspond to sub-arrays of the LCP array beginning at entries of zero in the LCP array. Knowing the contents of the subtrees, one can seek the minimum value of the LCP array for each subtree and deduce that this is precisely the length of the parent edge of this subtree (since it is the length of prefix that all suffixes in this subtree have in common). Subtracting this quantity from the LCP array for each sub-tree then allows this process to be repeated recursively until the entire suffix tree has been built.

If implemented carefully, this entire process can be done in $O(N)$ time. Combined with the linear time construction for suffix arrays and the LCP array, this yields a linear time algorithm for producing a suffix tree.

Ukkonen's algorithm (NOT EXAMINABLE)

A very elegant algorithm for constructing suffix trees was given by Ukkonen, who produced an algorithm that was not only linear time but also **online,** meaning that you can continue to add characters to the string while updating the suffix tree without having to start over from scratch. In its essence, Ukkonen's algorithm works by extending the length of each leaf edge of the suffix tree by one for each new character inserted, and appropriately splitting existing edges into two whenever a common prefix diverges. Those who are interested should read *On-line construction of suffix trees*, Ukkonen, E. and this stackoverflow post `http://stackoverflow.com/questions/9452701/ukkonens-suffix-tree-algorithm-in-plain-english` which describes the algorithm in a very accessible way.

Applications of Suffix Trees

Pattern matching

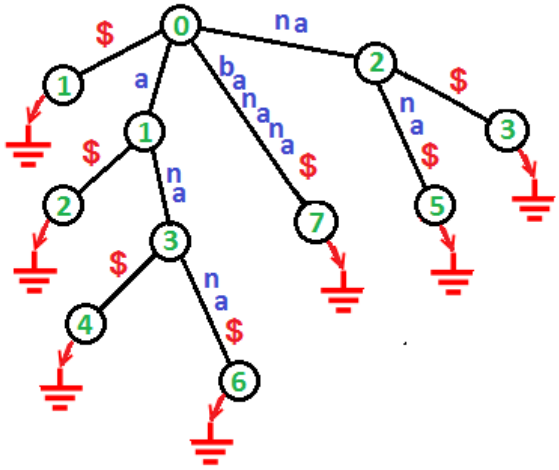
Just as we could use the suffix trie to perform pattern matching in $O(M)$ time, so too can we use the suffix tree for pattern matching in exactly the same way. The implementation is a little more involved since traversing the suffix tree along the edges requires more work (to check that each character along the edge is a match), but the idea and complexity are the same.

The longest repeated substring problem

Problem Statement: Longest Repeated Substring

Given a string `S[1..N]`, find the longest substring of `S` that occurs at least two times.

To solve this problem, we recall that within a prefix trie and equivalently a suffix tree, internal nodes correspond precisely to common prefixes, and hence in the case of suffix trees, substrings that occur multiple times. Finding the longest repeated substring is therefore simply a matter of traversing the suffix tree and looking for the deepest internal node (non-leaf node).



A suffix tree for the string “banana$” where each node is labelled with its depth (distance from the root node as measured by the number of characters on each edge). The deepest internal node has depth 3, which corresponds to the substring ”ana.”