FIT2004: Supplementary notes for lecture 6

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

- `http://www.allisons.org/ll/AlgDS/Tree/Suffix/`
- Weiss, Data Structures and Algorithm Analysis, Section 12.4.2
- `https://visualgo.net/suffixtree`

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 the lecture slides developed by Arun Konagurthu for FIT2004.

Key Ideas: Prefix Trie

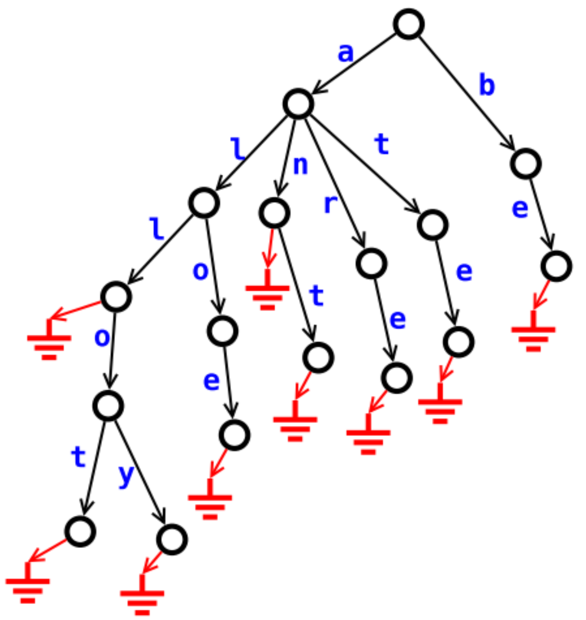
- The prefix trie is a (not necessarily binary) tree structure
- Each **edge** (not vertex) of the tree is labelled with a character
- Nodes can be marked as corresponding to the ends of words
- Each string stored in the trie corresponds to some path from the root to a marked node in the tree
- A path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

Prefix tries can be implemented in a variety of ways, including:

- Have the nodes be implicitly represented by indices, and maintain a collection of arrays listings the indices of each node's children (along with the corresponding character for each child edge.)
- Maintain a collection of explicit node objects, where each node maintains an array of pointers to its child node objects

The outgoing edge list for each node might include a slot for every character in the alphabet (facilitating fast $O(1)$ lookup at the cost of extra space), or might only contain entries for existing children stored as a dynamic array or other lookup structure (hashtable or binary search tree), ie. trading off time to save memory.

Note that these strategies are not specific to prefix tries, but are common when working with trees and graphs in general.



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.

Prefix tries work well particularly for small fixed size alphabets but may perform poorly on arbitrarily large alphabets since the majority of a node's children will be null. Depending on the choice of implementation, this might also result in a large amount of wasted space.

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. 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.

Algorithm: Insertion into a Prefix Trie

```
function INSERT(S[1..M])
    Set node = root
    for each character c in S[1..M] do
        if node has an edge for character c then
            node = node.get_child(c)
        else
            node = node.create_child(c)
        end if
    end for
    if node is the end of a word then
        ▷ The string S was already in the trie
    else
        node.mark_as_end()
    end if
end function
```

Algorithm: Lookup in a Prefix Trie

```
function LOOKUP(S[1..M])
    Set node = root
    for each character c in S[1..M] do
        if node has an edge for character c then
            node = node.get_child(c)
        else
            return false
        end if
    end for
    if node is the end of a word then
        return true
    else
        return false
    end if
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.

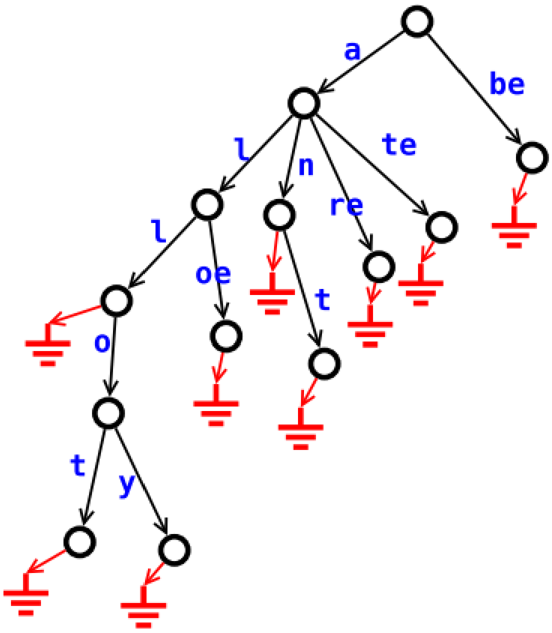
Algorithm: Sort Strings using a Prefix Trie

```
function SORT_STRINGS(list[1..N])
    for each string S in list do
        insert(S)
    end for
    traverse(root, "")
end function

function TRAVERSE(node, cur_string)
    if node is the end of a word then
        print(cur_string)
    end if
    for each character c in the alphabet in order do
        if node has an edge with character c then
            cur_string.append(c)
            traverse(node.get_child(c), cur_string)
            cur_string.pop_back()
        end if
    end for
end function
```

Compact Tries

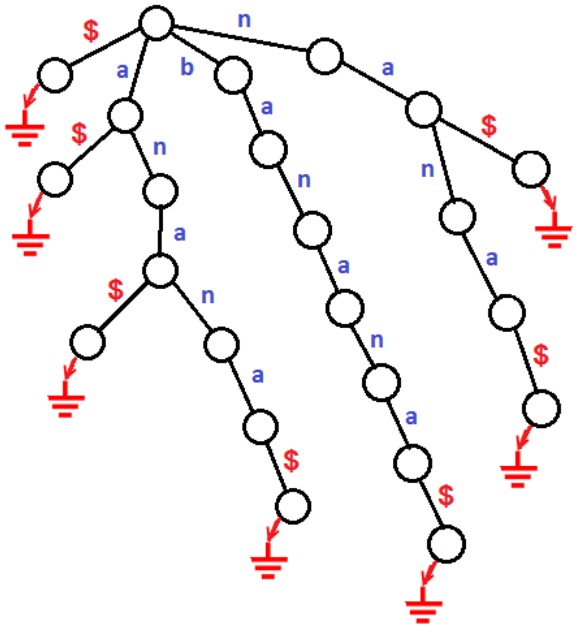
One major drawback to tries that we have observed is that they tend to waste a lot of memory, particularly if they contain long non-branching paths (paths where each node only has one child). In cases like these, we can make prefix tries more efficient by combining the edges along a non-branching path 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

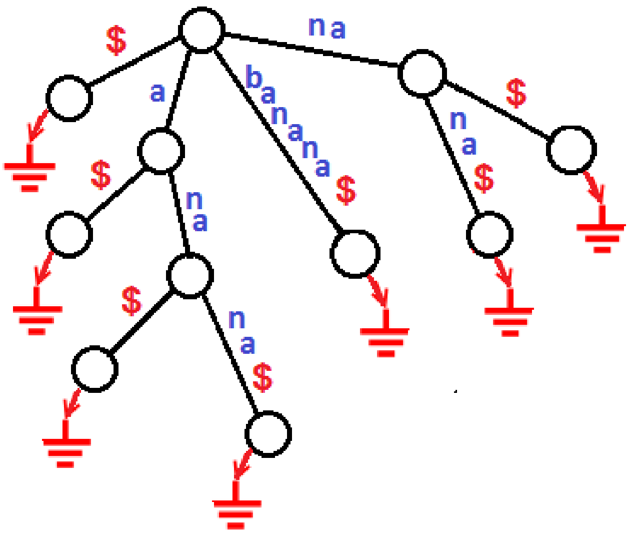
Recall that a suffix of a string is any substring that contains the final character of the string (and in some definitions, the empty suffix containing no characters.) We could take all of the suffixes of a string and insert them into a prefix trie, which would give us a data structure called the *suffix 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 append a special character to the end of the string to remind us. Typically, we denote this special character by `$` (a dollar sign.)



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.

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

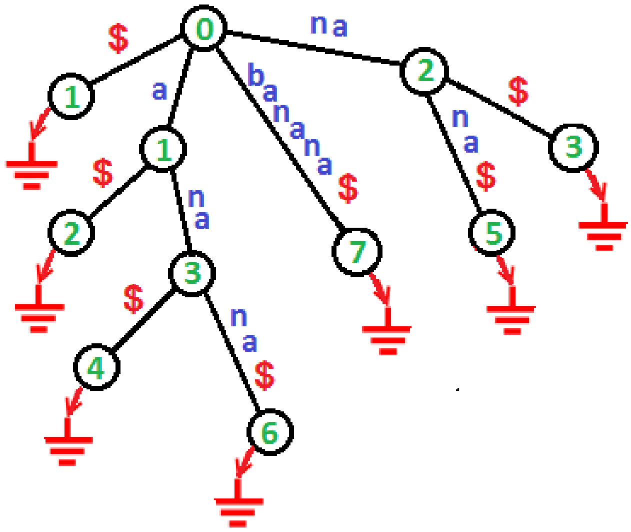
Just as we could use the suffix trie to perform pattern matching in $O(M)$ time, so too can we use the suffix tree for pattern matching in exactly the same way. The implementation is a little more involved since traversing the suffix tree along the edges requires more work (to check that each character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

Disclaimer: These notes are intended to supplement and enrich the content provided by the lectures. They are not intended to be a complete and thorough reference and should not be used as a replacement for attending lectures.