Trie, and Its Applications

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

Simulation Weblink

https://www.cs.usfca.edu/∼galles/visualization/Trie.html

1 Trie

A trie (a.k.a prefix tree) is essentially a tree that stores a collection of strings $\{S_1, S_2, \ldots, S_k\}$, so as to facilitate fast queries on this collection. The essential idea is start from a tree, which only has the root node. Now insert the first string S_1 into the trie by creating an edge and a node for each character in S_1, such that the node following the i^{th} character is the parent of the node following the $(i+1)^{th}$ character; the first character, obviously, branches off from the root.

To insert any other string S_j, we first try to match S_j in the trie as long as we can (by traversing down the trie starting from the root). Once a mismatch occurs, we create a new edge and a node for the failed character, and make the last matched node the parent of this new node. For the remainder of the string, create an edge and a node for each character, as described previously.

See Figure 1 for illustration.

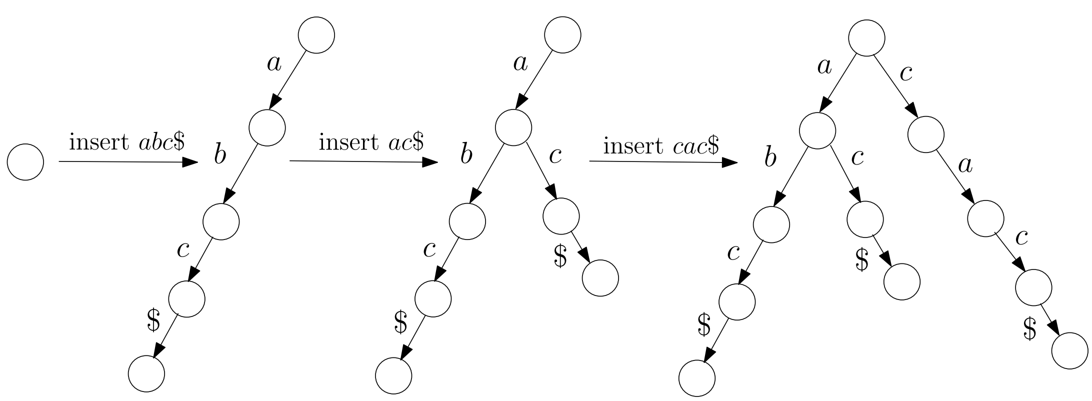

Figure 1: A trie for the strings $\{abc\$, ac\$, cac\$\}$

2 Pattern Matching

Pattern Matching Problem: *Given a text T containing n characters and a pattern P containing p characters, find the starting positions of all sub-strings of T that match exactly with P. If no such position exists, report that no match is found.*

Thus, if $T = banana$ and $P = ana$, we have to report positions 1 and 3. If $P = a$, we have to report positions 1, 3, and 5. If $P = anb$, no match exists.

2.1 A Naive implementation

An obvious approach is to try and match P at every position of T. Although, simple to implement, the worst-case complexity is $O(np)$, which is too slow for most practical purposes.[1]

In most cases, n is much larger compared to p, and also the text T remains static, i.e., it hardly changes. Hence, the main question is whether we can build a data structure, using which we can support pattern matching much faster, ideally, in time proportional to the length p of P and the number of occurrences of P in T. We show that we can use a trie to this end.

2.2 Breaking the Problem

Before we discuss the data structure, let us look at a few definitions:

- A suffix of a string is a substring of the string that ends at the last position.
- A prefix of a string is a substring of the string that starts at the first position.

Main Observation: *A pattern P occurs at a position i of the text T if and only if P is a prefix of the suffix starting at i.*

In other words, if we can find all the suffixes (i.e., their starting positions) which begin with the pattern P, then we are done. The real challenge is to somehow organize suffixes to facilitate this quickly. Here, comes in the usefulness of a trie.

2.3 Suffix Trie

A suffix trie of a text T is a trie of all the suffixes of T. Before creating a suffix trie, we append \$ to T to ensure the following property.

Prefix Free Property: *A collection of strings is prefix-free if no string in the collection is a prefix of another string in the collection.* Thus, $\{abc, bc\}$ is prefix-free, but $\{abc, ab\}$ is not.

For example, if $T = banana$, we first append \$ to get *banana*\$. **Now, create a trie of all the suffixes of** *banana*\$, i.e., create a trie of the following strings (note that they are prefix-free):

- banana\$
- anana\$
- nana\$
- ana\$
- na\$
- a\$
- \$

Now, we augment the trie with some additional information. Observe that due to the prefix-free property of the suffixes, each leaf in the trie corresponds to a unique suffix of T, i.e., the path from root to a leaf spells out a unique suffix. **With each leaf, we store the starting position of the corresponding suffix.** See Figure 2.3 for illustration.

[1] A faster algorithm, known as the KMP algorithm can report all the occurrences in $O(n + p)$ time.

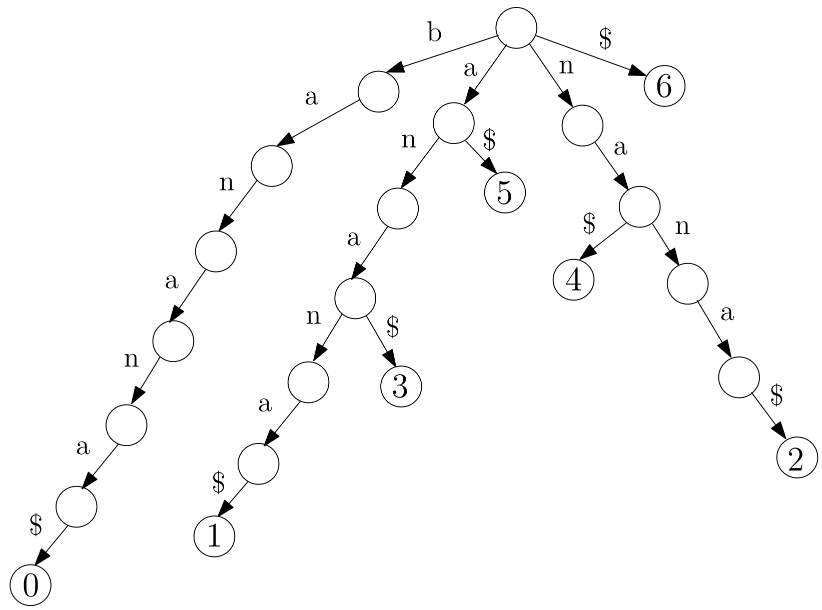

Figure 2: Suffix trie for $T = banana\$$

2.4 Finding All Occurrences of a pattern P

To report all occurrences of P, we traverse from the root of the suffix trie by using the characters of P. We may run into one of the following scenarios:

- We have consumed the pattern P and reached a node u. This simply means that all the suffixes that begin with P lie in the subtree of u. By our construction, we can find the starting positions of all these suffixes simply by checking each leaf in the subtree of u. Recall that the starting positions of these suffixes are exactly the occurrences of P in T.
- We are at a node (possibly the root), and we were not able to match the next character of the pattern P. This means there does not exist a suffix, which is prefixed by P; hence, there does not exist any position where P occurs in T.

See Figure 2.4 for illustration.

3 Complexity (not required)

Note that in the worst case, i.e., when each string begins with a different character, we create a node for each character of each string in the collection. Hence, the trie will have $(L + 1)$ nodes, where L is the total length of all the strings. Therefore, the space complexity is $O(L)$.

In a suffix trie, this translates to a space complexity of $O(n^2)$, where n is the length of T; the worst case is realized when each character of T is unique.[2]

We can report all the occurrences in $O(p + occ)$ time, where p is the length of P and occ is the number of occurrences of P in T. Moreover, this is the best that one can hope for, as in the worst

[2] The suffixes are of length $1, 2, 3, \ldots, n$. Hence, the total length is $1 + 2 + 3 + \cdots + n = \dfrac{n(n+1)}{2} = O(n^2)$.

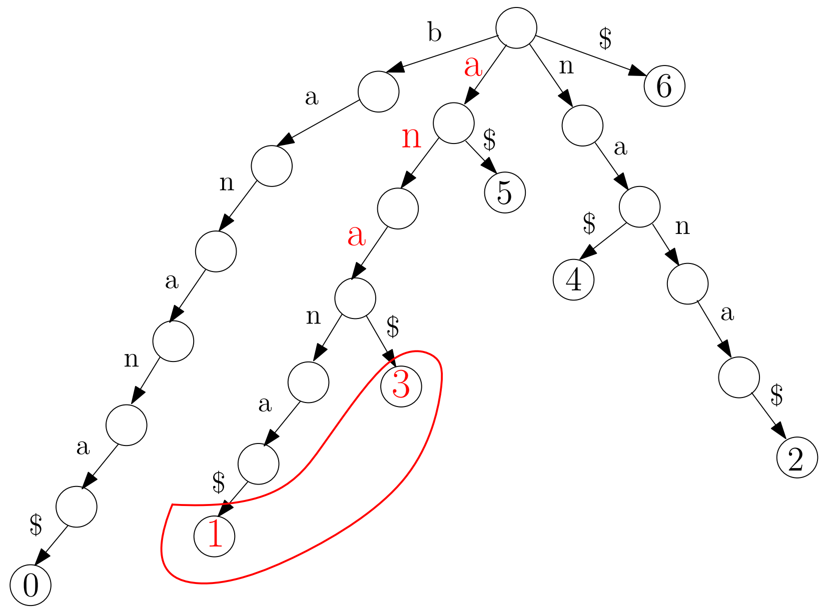

Figure 3: Searching for $P = ana$ in the suffix trie for $T = banana\$$

case, we need to read P completely and need at least $O(1)$ time to report each occurrence.[3]

Although the query time is fast, the space complexity, unfortunately, for a suffix trie is too large. As a side remark, suffix trie can be modified to yield a more space-efficient data structure called the *suffix tree*. This occupies space $O(n)$ in the worst-case (a substantial improvement from the $O(n^2)$ space of suffix trie), whereas, it still matches the $O(p + occ)$ query time of the suffix trie.

4 Other Applications (not required)

Following are a couple of other applications of trie:

- **Dictionary Matching** (using Aho-Corasick Automata): Given a collection of patterns and a text, report all those positions in the text, where at least one pattern occurs.
- **Spell Check in a Word Processor**: The idea is to create a trie of all the strings in the English Dictionary. Now, whenever you type in a word on the processor, run the word through the trie to detect whether it has a correct spelling, or not.

[3] The query time is really achieved by augmenting the suffix trie with additional information. Each node in the trie is equipped with constant time hashing (a.k.a *perfect hashing*,) such that given any character, in $O(1)$ time, we can jump to its appropriate child or detect there is none.

Additionally, we store an array over the leaves in the suffix trie. Basically, the i^{th} index in the array corresponds to the $(i+1)^{th}$ leftmost leaf, i.e., array index 0 corresponds to the leftmost leaf, array index 1 corresponds to the second leftmost leaf, and so on. Each cell in the array stores the starting position of the suffix of the corresponding leaf. Also, with each node v in the suffix trie, we store the range (i.e., the starting index and the ending index) of the array that corresponds to the leaves under v.

We first traverse the suffix trie in $O(p)$ time to find the node u, such that all the suffixes under u are prefixed by P. Now use the array-range stored in u to find the starting positions of these suffixes in $O(1)$ time per suffix.