Cairo University
Faculty of Computers and Artificial Intelligence
Computer Science Department

Biological Sequence Analysis

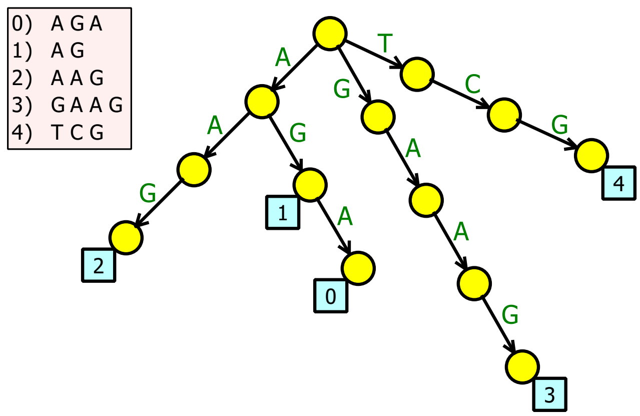
Tries and Suffix Trees

Dr. Amin Allam

[For more details, refer to “Jewels of Stringology” by *Maxime Crochemore* and *Wojciech Rytter*]

1 Tries

The *trie* data structures is a *tree* that stores several *small strings* (*dataset*), and allows to search for (retrieve) a given (*query*) *string* inside the stored *dataset*. The following *trie* stores 5 *strings*:



Insertions and retrievals start from the *root*. Each *edge* is *labelled* with one *character*. *Edges* from a *node* to its *children* must be *labelled* with different *characters*. The ID of a *dataset string* is contained in the *node* such that the *path* from the *root* to that *node* is *labelled* with that *string* (as shown in the squares in the above figure).

Each insertion or retrieval traverses at most exactly m *edges* where m is the *string* length, thus costing $O(m)$ time assuming that $O(1)$ time is needed to traverse from a *node* to its correct *child* according to the *edge label*.

Suppose that the *alphabet size* (number of possible different *characters*) is $|\Sigma|$. A *trie* can be implemented using one of the following methods:

• Each *node* contains an *array* of length $|\Sigma|$, whose i^{th} element holds a *child node pointer* connected by the i^{th} *character* of the *alphabet*. Each *node* requires $O(1)$ time and $O(|\Sigma|)$ space.

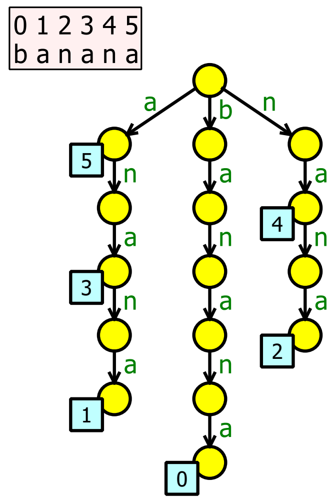
• Each *node* contains a *linked list* where each element contains a *character* and a *child node pointer*. Each *node* requires $O(|\Sigma|)$ time and $O(1)$ space.

• Each *node* contains a *red-black tree* where each element contains a *character* as the *key*, and a *child node pointer*. Each *node* requires $O(log(|\Sigma|))$ time and $O(1)$ space.

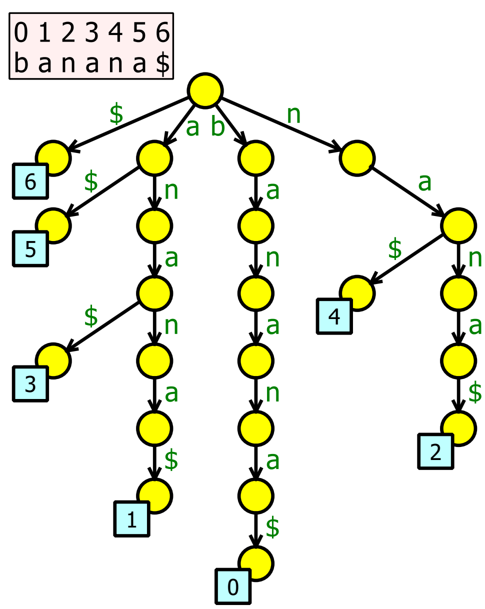
• One *hash table* for the whole *trie*, where each element contains a *character* and two *node pointers*: *parent* and *child*. The *hash function* is a function of the *parent node pointer* and the *character*. Each *node* requires $O(1)$ time and $O(1)$ space, but this method suffers from *cache misses*.

2 Suffix tries

The *suffix trie* data structures is a *trie* that stores all *suffixes* of a given *large string* of length n. A *suffix* of a *string* is a *substring* that ends at the last location $(n-1)$. The *suffix* ID is its *starting location* inside the original *string*. The *suffix trie* allows to search for a given *substring* inside the original *string*. The following *suffix trie* stores all *suffixes* of the *string* `banana`:



The *suffix trie* requires $O(n^2)$ space and construction time, which makes it impractical. To make it practical, *nodes* with one *child* should be removed. Before doing that, a *sentinel* `$` is added to the *original string* to make sure that no *suffix* ends at an *internal node* of the *suffix trie*:

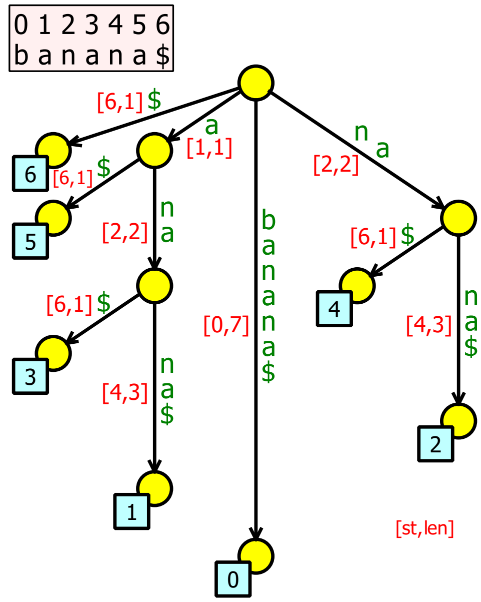


To search for a *substring*, the *suffix trie* is traversed from the *root* to a *node*. IDs associated with all *nodes* in the *subtree* of the reached *node* are reported as locations of that *substring*. For example, searching for `an` or `ana` results {3,1}, while searching for `a` results {5,3,1}.

Now, *one-child nodes* can be safely removed to make a *suffix tree*. Also, since all *suffixes* end at *leaves*, *suffix* IDs can be removed and deduced after *query* traversal by subtracting number of traversed *characters* from n.

3 Suffix trees

A *suffix tree* is a compact *suffix trie* which contains all *suffixes* of an *original string* of length n (including $), does not contain any *one-child node*, and all suffixes end at *leaves*. Consider the following *suffix tree* of the *string* `banana$`:



After *one-child nodes* are removed, some *edges* need to be *labelled* with *substrings*, not with single *characters* as in the *suffix trie*. To avoid *O*(n^2) space, *edges* are *labelled* with the *start location* and the *length* of a *substring* inside the *original string*, instead of *labelling* them with the *substrings* themselves. Thus, the *original string* must be available to conduct *queries*. *Substrings* are shown on *edges* in the above figure only for illustration. The *substring length* can also be removed and deduced by subtracting the smallest *start location* of *children* from the *start location* of *parent*.

Thus, each *node* in the *suffix tree* needs $O(1)$ space, and the number of *leaves* equals to the number of suffixes n. The number of internal nodes is $\leq n-1$ *. Thus, the *suffix tree* needs $O(n)$ space.

$*$ The number of *internal nodes* of a *tree* with no *one-child nodes* $\leq$ number of *leaves* -1.
Proof: Consider a procedure which starts with *leaves* and attempts to construct arbitrary *tree* by picking at least two *nodes* and creating a new *internal node* as their *parent*. After each step, the number of *nodes* with no *parent* decreases by one. The procedure stops when there is exactly one *node* with no *parent* (which is the *root*). The number of steps, as well as the number of created *internal nodes*, cannot exceed the number of *leaves* -1.

To construct a *suffix tree*, we should not create an $O(n^2)$ *suffix trie* then use it to construct the *suffix tree*, because $O(n^2)$ space or time is not available for *large strings*. Ukkonen proposed a practical algorithm to construct the $O(n)$ *suffix tree* using only $O(n)$ space and time.

The time complexity of searching for a *substring* inside the *suffix tree* is $O(m+occ)$ where m is the length of the substring, and occ is the number of occurrences of that *substring* inside the *original string*. That result follows because $O(m)$ is needed as initial traversal, then $O(occ)$ is needed for a depth first search starting from the *internal node* or the place where we stopped.