Baeza-Yates–Perleberg Filtering Algorithm

A filtering algorithm for approximate string matching searches the text for factors having some property that satisfies the following conditions:

1. Every approximate occurrence of the pattern has this property.
2. Strings having this property are reasonably rare.
3. Text factors having this property can be found quickly.

Each text factor with the property is a potential occurrence, which is then verified for whether it is an actual approximate occurrence.

Filtering algorithms can achieve linear or even sublinear average case time complexity.

The following lemma shows the property used by the Baeza-Yates–Perleberg algorithm and proves that it satisfies the first condition.

Lemma 3.23: Let $P_1P_2 \dots P_{k+1} = P$ be a partitioning of the pattern P into $k+1$ nonempty factors. Any string S with $ed(P,S) \leq k$ contains P_i as a factor for some $i \in [1..k+1]$.

Proof. Each single symbol edit operation can change at most one of the pattern factors P_i. Thus any set of at most k edit operations leaves at least one of the factors untouched. □

The algorithm has two phases:

Filtration: Search the text T for exact occurrences of the pattern factors P_i. Using the Aho–Corasick algorithm this takes $\mathcal{O}(n)$ time for a constant alphabet.

Verification: An area of length $\mathcal{O}(m)$ surrounding each potential occurrence found in the filtration phase is searched using the standard dynamic programming algorithm in $\mathcal{O}(m^2)$ time.

The worst case time complexity is $\mathcal{O}(m^2 n)$, which can be reduced to $\mathcal{O}(mn)$ by combining any overlapping areas to be searched.

Let us analyze the average case time complexity of the verification phase.

- The best pattern partitioning is as even as possible. Then each pattern factor has length at least $r = \lfloor m/(k+1) \rfloor$.
- The expected number of exact occurrences of a random string of length r in a random text of length n is at most n/σ^r.
- The expected total verification time is at most
$$\mathcal{O}\left(\frac{m^2(k+1)n}{\sigma^r}\right) \leq \mathcal{O}\left(\frac{m^3 n}{\sigma^r}\right) \ .$$
This is $\mathcal{O}(n)$ if $r \geq 3 \log_\sigma m$.
- The condition $r \geq 3 \log_\sigma m$ is satisfied when $(k+1) \leq m/(3 \log_\sigma m + 1)$.

Theorem 3.24: The average case time complexity of the Baeza-Yates–Perleberg algorithm is $\mathcal{O}(n)$ when $k \leq m/(3 \log_\sigma m + 1) - 1$.

Many variations of the algorithm have been suggested:

- The filtration can be done with a different multiple exact string matching algorithm:
  - The first algorithm of this type by Wu and Manber used an extension of the Shift-And algorithm.
  - An extension of BDM achieves $\mathcal{O}(nk(\log_\sigma m)/m)$ average case search time. This is sublinear for small enough k.
  - An extension of the Horspool algorithm is very fast in practice for small k and large σ.
- Using a technique called hierarchical verification, the average verification time for a single potential occurrence can be reduced to $\mathcal{O}((m/k)^2)$.

A filtering algorithm by Chang and Marr has average case time complexity $\mathcal{O}(n(k + \log_\sigma m)/m)$, which is optimal.

Summary: Approximate String Matching

We have seen two main types of algorithms for approximate string matching:

- Basic dynamic programming time complexity is $\mathcal{O}(mn)$. The time complexity can be improved to $\mathcal{O}(kn)$ using diagonal monotonicity, and to $\mathcal{O}(n\lceil m/w \rceil)$ using bitparallelism.
- Filtering algorithms can improve average case time complexity and are the fastest in practice when k is not too large.

Other algorithms worth mentioning are those based on automata:

- Algorithms based on bit-parallel simulation of non-deterministic automata were mentioned briefly.
- A deterministic automaton can find occurrences in $\mathcal{O}(n)$ time, but the size of the automaton can be exponential in m and too big to be practical.

Similar techniques can be useful for other variants of edit distance but not always straightforwardly.

4. Suffix Trees and Arrays

Let $T = T[0..n)$ be the text. For $i \in [0..n]$, let T_i denote the suffix $T[i..n)$. Furthermore, for any subset $C \in [0..n]$, we write $T_C = \{T_i \mid i \in C\}$. In particular, $T_{[0..n]}$ is the set of all suffixes of T.

Suffix tree and suffix array are search data structures for the set $T_{[0..n]}$.

- Suffix tree is a compact trie for $T_{[0..n]}$.
- Suffix array is an ordered array for $T_{[0..n]}$.

They support fast exact string matching on T:

- A pattern P has an occurrence starting at position i if and only if P is a prefix of T_i.
- Thus we can find all occurrences of P by a prefix search in $T_{[0..n]}$.

A data structure supporting fast string matching is called a text index.

There are numerous other applications too, as we will see later.

The set $\mathcal{T}_{[0..n]}$ contains $|\mathcal{T}_{[0..n]}| = n + 1$ strings of total length $||\mathcal{T}_{[0..n]}|| = \Theta(n^2)$. It is also possible that $\Sigma LCP(\mathcal{T}_{[0..n]}) = \Theta(n^2)$, for example, when $T = \mathtt{a}^n$ or $T = XX$ for any string X.

- A basic trie has $\Theta(n^2)$ nodes for most texts, which is too much. Even a leaf path compacted trie can have $\Theta(n^2)$ nodes, for example when $T = XX$ for a random string X.

- A compact trie with $\mathcal{O}(n)$ nodes and an ordered array with $n+1$ entries have linear size.

- A compact ternary trie and a string binary search tree have $\mathcal{O}(n)$ nodes too. However, the construction algorithms and some other algorithms we will see are not straightforward to adapt for these data structures.

Even for a compact trie or an ordered array, we need a specialized construction algorithm, because any general construction algorithm would need $\Omega(\Sigma LCP(\mathcal{T}_{[0..n]}))$ time.

Suffix Tree

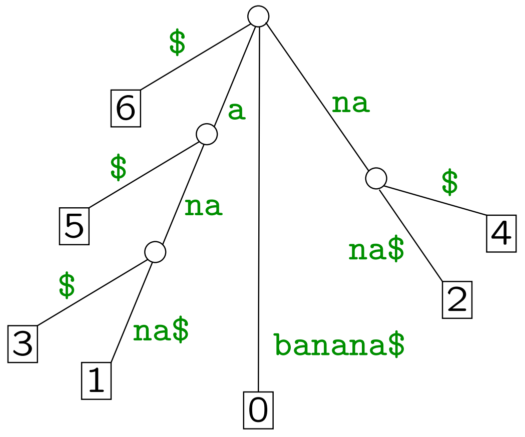
The suffix tree of a text T is the compact trie of the set $T_{[0..n]}$ of all suffixes of T.

We assume that there is an extra character $\$ \notin \Sigma$ at the end of the text. That is, $T[n] = \$$ and $T_i = T[i..n]$ for all $i \in [0..n]$. Then:

- No suffix is a prefix of another suffix, i.e., the set $T_{[0..n]}$ is prefix free.
- All nodes in the suffix tree representing a suffix are leaves.

This simplifies algorithms.

Example 4.1: $T =$ banana$.



As with tries, there are many possibilities for implementing the child operation. We again avoid this complication by assuming that σ is constant. Then the size of the suffix tree is $\mathcal{O}(n)$:

- There are exactly $n+1$ leaves and at most n internal nodes.
- There are at most $2n$ edges. The edge labels are factors of the text and can be represented by pointers to the text.

Given the suffix tree of T, all occurrences of P in T can be found in time $\mathcal{O}(|P| + occ)$, where occ is the number of occurrences.

Brute Force Construction

Let us now look at algorithms for constructing the suffix tree. We start with a brute force algorithm with time complexity $\Theta(\Sigma LCP(T_{[0..n]}))$. Later we will modify this algorithm to obtain a linear time complexity.

The idea is to add suffixes to the trie one at a time starting from the longest suffix. The insertion procedure is essentially the same as we saw in Algorithm 1.2 (insertion into trie) except it has been modified to work on a compact trie instead of a trie.

Let S_u denote the string represented by a node u. The suffix tree representation uses four functions:

child(u, c) is the child v of node u such that the label of the edge (u, v) starts with the symbol c, and $\perp$ if u has no such child.

parent(u) is the parent of u.

depth(u) is the length of S_u.

start(u) is the starting position of some occurrence of S_u in T.

Then

- $S_u = T[\mathit{start}(u) \ldots \mathit{start}(u) + \mathit{depth}(u))$.
- $T[\mathit{start}(u) + \mathit{depth}(\mathit{parent}(u)) \ldots \mathit{start}(u) + \mathit{depth}(u))$ is the label of the edge $(\mathit{parent}(u), u)$.

A locus in the suffix tree is a pair (u, d) where $\mathit{depth}(\mathit{parent}(u)) < d \leq \mathit{depth}(u)$. It represents

- the uncompact trie node that would be at depth d along the edge $(\mathit{parent}(u), u)$, and
- the corresponding string $S_{(u,d)} = T[\mathit{start}(u) \ldots \mathit{start}(u) + d)$.

Every factor of T is a prefix of a suffix and thus has a locus along the path from the root to the leaf representing that suffix.

During the construction, we need to create nodes at an existing locus in the middle of an edge, splitting the edge into two edges:

```
CreateNode($u, d$)        // $d < \mathit{depth}(u)$
  (1) $i \leftarrow \mathit{start}(u)$; $p \leftarrow \mathit{parent}(u)$
  (2) create new node $v$
  (3) $\mathit{start}(v) \leftarrow i$; $\mathit{depth}(v) \leftarrow d$
  (4) $\mathit{child}(v, T[i + d]) \leftarrow u$; $\mathit{parent}(u) \leftarrow v$
  (5) $\mathit{child}(p, T[i + \mathit{depth}(p)]) \leftarrow v$; $\mathit{parent}(v) \leftarrow p$
  (6) return $v$
```

Now we are ready to describe the construction algorithm.

Algorithm 4.2: Brute force suffix tree construction
Input: text $T[0..n]$ ($T[n] = \$$)
Output: suffix tree of T: $root$, *child*, *parent*, *depth*, *start*

```
(1)  create new node $root$; $depth(root) \leftarrow 0$
(2)  $u \leftarrow root$; $d \leftarrow 0$        // $(u, d)$ is the active locus
(3)  for $i \leftarrow 0$ to $n$ do       // insert suffix $T_i$
(4)      while $d = depth(u)$ and $child(u, T[i+d]) \neq \perp$ do
(5)          $u \leftarrow child(u, T[i+d])$; $d \leftarrow d+1$
(6)          while $d < depth(u)$ and $T[start(u)+d] = T[i+d]$ do $d \leftarrow d+1$
(7)      if $d < depth(u)$ then       // $(u, d)$ is in the middle of an edge
(8)          $u \leftarrow$ CreateNode$(u, d)$
(9)      CreateLeaf$(i, u)$
(10)     $u \leftarrow root$; $d \leftarrow 0$
```

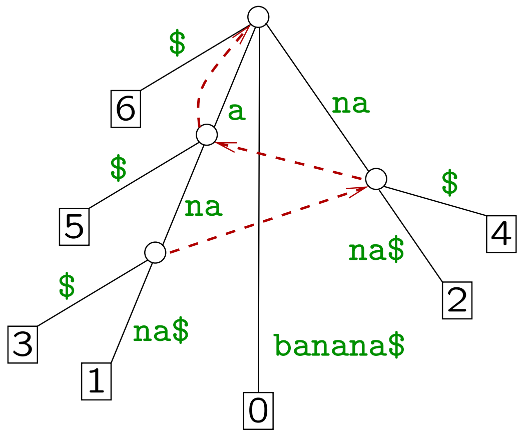
```
CreateLeaf$(i, u)$        // Create leaf representing suffix $T_i$
(1)  create new leaf $w$
(2)  $start(w) \leftarrow i$; $depth(w) \leftarrow n - i + 1$
(3)  $child(u, T[i+d]) \leftarrow w$; $parent(w) \leftarrow u$        // Set $u$ as parent
(4)  return $w$
```

Suffix Links

The key to efficient suffix tree construction are suffix links:

> *slink*(u) is the node v such that S_v is the longest proper suffix of S_u, i.e., if $S_u = T[i..j)$ then $S_v = T[i+1..j)$.

Example 4.3: The suffix tree of $T =$ `banana$` with internal node suffix links.



Suffix links are well defined for all nodes except the root.

Lemma 4.4: If the suffix tree of T has a node u representing $T[i..j)$ for any $0 \leq i < j \leq n$, then it has a node v representing $T[i+1..j)$.

Proof. If u is the leaf representing the suffix T_i, then v is the leaf representing the suffix T_{i+1}.

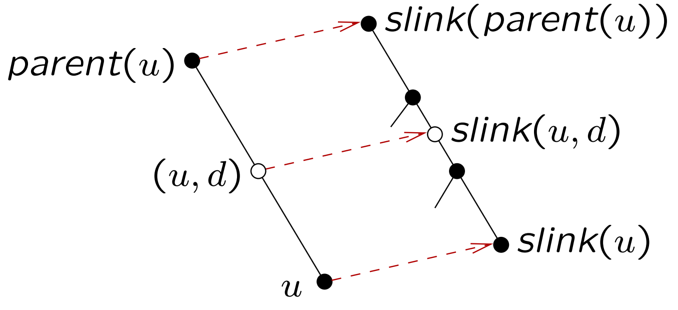
If u is an internal node, then it has two child edges with labels starting with different symbols, say a and b, which means that $T[i..j)a$ and $T[i..j)b$ are both factors of T. Then, $T[i+1..j)a$ and $T[i+1..j)b$ are factors of T too, and thus there must be a branching node v representing $T[i+1..j)$. □

Usually, suffix links are needed only for internal nodes. For root, we define *slink*$(root) = root$.

Suffix links are the same as Aho–Corasick failure links but Lemma 4.4 ensures that $\mathit{depth}(\mathit{slink}(u)) = \mathit{depth}(u) - 1$. This is not the case for an arbitrary trie or a compact trie.

Suffix links are stored for compact trie nodes only, but we can define and compute them for any locus (u, d):

```
slink(u, d)
  (1)  $v \leftarrow \mathit{slink}(\mathit{parent}(u))$
  (2)  while $\mathit{depth}(v) < d - 1$ do
  (3)      $v \leftarrow \mathit{child}(v, T[\mathit{start}(u) + \mathit{depth}(v) + 1])$
  (4)  return $(v, d - 1)$
```



The same idea can be used for computing the suffix links during or after the brute force construction.

```
ComputeSlink($u$)
  (1) $d \leftarrow \textit{depth}(u)$
  (2) $v \leftarrow \textit{slink}(\textit{parent}(u))$
  (3) while $\textit{depth}(v) < d - 1$ do
  (4)       $v \leftarrow \textit{child}(v, T[\textit{start}(u) + \textit{depth}(v) + 1])$
  (5) if $\textit{depth}(v) > d - 1$ then        // no node at $(v, d - 1)$
  (6)       $v \leftarrow \text{CreateNode}(v, d - 1)$
  (7) $\textit{slink}(u) \leftarrow v$
```

The procedure CreateNode$(v, d - 1)$ sets *slink*$(v) = \perp$.

The algorithm uses the suffix link of the parent, which must have been computed before. Otherwise the order of computation does not matter.

The creation of a new node on line (6) never happens in a fully constructed suffix tree, but during the brute force algorithm the necessary node may not exist yet:

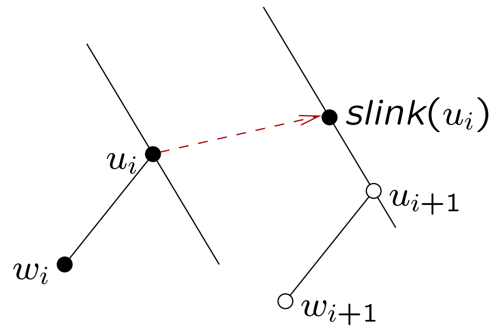
- If a new internal node u_i was created during the insertion of the suffix T_i, there exists an earlier suffix T_j, $j < i$ that branches at u_i into a different direction than T_i.
- Then *slink*(u_i) represents a prefix of T_{j+1} and thus exists at least as a locus on the path labelled T_{j+1}. However, it may be that it does not become a branching node until the insertion of T_{i+1}.
- In such a case, ComputeSlink(u_i) creates *slink*(u_i) a moment before it would otherwise be created by the brute force construction.

McCreight's Algorithm

McCreight's suffix tree construction is a simple modification of the brute force algorithm that computes the suffix links during the construction and uses them as short cuts:

- Consider the situation, where we have just added a leaf w_i representing the suffix T_i as a child to a node u_i. The next step is to add w_{i+1} as a child to a node u_{i+1}.

- The brute force algorithm finds u_{i+1} by traversing from the root. McCreight's algorithm takes a short cut to *slink*(u_i).



- This is safe because *slink*(u_i) represents a prefix of T_{i+1}.

Algorithm 4.5: McCreight
Input: text $T[0..n]$ $(T[n] = \$)$
Output: suffix tree of T: $root$, *child*, *parent*, *depth*, *start*, *slink*

```
 (1) create new node $root$; $depth(root) \leftarrow 0$; $slink(root) \leftarrow root$
 (2) $u \leftarrow root$; $d \leftarrow 0$        // $(u,d)$ is the active locus
 (3) for $i \leftarrow 0$ to $n$ do       // insert suffix $T_i$
 (4)       while $d = depth(u)$ and $child(u, T[i+d]) \neq \bot$ do
 (5)             $u \leftarrow child(u, T[i+d])$; $d \leftarrow d+1$
 (6)             while $d < depth(u)$ and $T[start(u)+d] = T[i+d]$ do $d \leftarrow d+1$
 (7)       if  $d < depth(u)$ then        // $(u,d)$ is in the middle of an edge
 (8)             $u \leftarrow$ CreateNode$(u,d)$
 (9)       CreateLeaf$(i,u)$
(10)       if $slink(u) = \bot$ then ComputeSlink$(u)$
(11)       $u \leftarrow slink(u)$; $d \leftarrow d-1$
```

Theorem 4.6: Let T be a string of length n over an alphabet of constant size. McCreight's algorithm computes the suffix tree of T in $\mathcal{O}(n)$ time.

Proof. Insertion of a suffix T_i takes constant time except in two points:

- The while loops on lines (4)–(6) traverse from the node *slink*(u_i) to u_{i+1}. Every round in these loops increments d. The only place where d decreases is on line (11) and even then by one. Since d can never exceed n, the total time on lines (4)–(6) is $\mathcal{O}(n)$.

- The while loop on lines (3)–(4) during a call to ComputeSlink(u_i) traverses from the node *slink*(*parent*(u_i)) to *slink*(u_i). Let d'_i be the depth of *parent*(u_i). Clearly, $d'_{i+1} \geq d'_i - 1$, and every round in the while loop during ComputeSlink(u_i) increases d'_{i+1}. Since d'_i can never be larger than n, the total time in the loop on lines (3)–(4) in ComputeSlink is $\mathcal{O}(n)$.

$\square$

There are other linear time algorithms for suffix tree construction:

- Weiner's algorithm was the first. It inserts the suffixes into the tree in the opposite order: $T_n, T_{n-1}, \ldots, T_0$.

- Ukkonen's algorithm constructs suffix tree first for $T[0..1)$ then for $T[0..2)$, etc.. The algorithm is structured differently, but performs essentially the same tree traversal as McCreight's algorithm.

- All of the above are linear time only for constant alphabet size. Farach's algorithm achieves linear time for an integer alphabet of polynomial size. The algorithm is complicated and unpractical.