Another way to utilize Lemma 3.15 ($\Delta h_{ij}, \Delta v_{ij} \in \{-1, 0, 1\}$) is to use precomputed tables to process multiple matrix cells at a time.

- There are at most 3^m different columns. Thus there exists a deterministic automaton with 3^m states and $\sigma 3^m$ transitions that can find all approximate occurrences in $\mathcal{O}(n)$ time. However, the space and constructions time of the automaton can be too big to be practical.

- There is a super-alphabet algorithm that processes $\mathcal{O}(\log_\sigma n)$ characters at a time and $\mathcal{O}(\log_\sigma^2 n)$ matrix cells at a time using lookup tables of size $\mathcal{O}(n)$. This gives time complexity $\mathcal{O}(mn/\log_\sigma^2 n)$.

- A practical variant uses smaller lookup tables to compute multiple entries of a column at a time.

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 $P_1 P_2 \ldots P_{k+1} = P$ 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 verification time can be reduced using a technique called hierarchical verification.
- The pattern can be partitioned into fewer than $k+1$ pieces, which are searched allowing a small number of errors.

A lower bound on the average case time complexity is $\Omega(n(k + \log_\sigma m)/m)$, and there exists a filtering algorithm matching this bound.

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.

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 $T_{[0..n]}$ contains $|T_{[0..n]}| = n + 1$ strings of total length $||T_{[0..n]}|| = \Theta(n^2)$. It is also possible that $\Sigma LCP(T_{[0..n]}) = \Theta(n^2)$, 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.
- A compact trie with $\mathcal{O}(n)$ nodes and an ordered array with $n + 1$ entries have linear size.
- A compact ternary trie has $\mathcal{O}(n)$ nodes too. However, the construction algorithms and some other algorithms we will see are not straightforward to adapt for it.

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

Suffix Tree

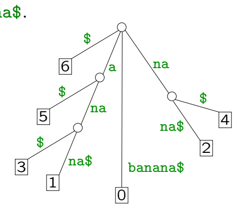
The suffix tree of a text T is the compact trie of the set $T_{[0..n]}$ of all suffixes of T.

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: $T = \mathtt{banana\$}$.



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

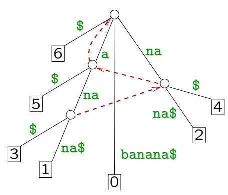
```
CreateLeaf(i, u)        // Create leaf representing suffix T_i
  (1)  create new leaf w
  (2)  start(w) ← i; depth(w) ← n − i + 1
  (3)  child(u, T[i + d]) ← w; parent(w) ← u        // Set u as parent
  (4)  return w
```


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 $T = \mathtt{banana\$}$ 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 $0 \le i < j \le n$, 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}.

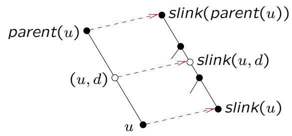
If u is an internal node, then it has two child edges with labels starting with different symbols, say a and b, which means that $T[i..j)a$ and $T[i..j)b$ are both factors of T. Then, $T[i+1..j)a$ and $T[i+1..j)b$ are factors of T too, and thus there must be a branching node v representing $T[i+1..j)$.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

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 *depth*(*slink*(u)) = *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) $v \leftarrow$ *slink*(*parent*(u))
(2) **while** *depth*$(v) < d - 1$ **do**
(3) $\quad v \leftarrow$ *child*$(v, T[$*start*$(u) +$ *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) $d \leftarrow$ *depth*(u)
(2) $v \leftarrow$ *slink*(*parent*(u))
(3) **while** *depth*$(v) < d - 1$ **do**
(4) $\quad v \leftarrow$ *child*$(v, T[$*start*$(u) +$ *depth*$(v) + 1])$
(5) **if** *depth*$(v) > d - 1$ **then** // no node at $(v, d-1)$
(6) $\quad v \leftarrow$ CreateNode$(v, d - 1)$
(7) *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.

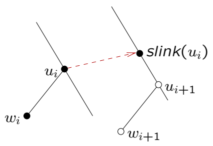
The creation of a new node on line (6) is never needed in a fully constructed suffix tree, but during the brute force algorithm the necessary node may not exist yet:

- 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 might 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) $\quad$ **while** $d =$ *depth*(u) **and** *child*$(u, T[i+d]) \ne \perp$ **do**
(5) $\quad\quad u \leftarrow$ *child*$(u, T[i+d])$; $d \leftarrow d + 1$
(6) $\quad\quad$ **while** $d <$ *depth*(u) **and** $T[$*start*$(u) + d] = T[i+d]$ **do** $d \leftarrow d + 1$
(7) $\quad$ **if** $d <$ *depth*(u) **then** // (u, d) is in the middle of an edge
(8) $\quad\quad u \leftarrow$ CreateNode(u, d)
(9) $\quad$ CreateLeaf(i, u)
(10) $\quad$ **if** *slink*$(u) = \perp$ **then** ComputeSlink(u)
(11) $\quad u \leftarrow$ *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, $d'_{i+1} \ge d'_i - 1$, 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 $\mathcal{O}(n)$.

□

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.
- Practical linear time construction for an integer alphabet is possible via suffix array.

Applications of Suffix Tree

Let us have a glimpse of the numerous applications of suffix trees.

Exact String Matching

As already mentioned earlier, given the suffix tree of the text, all occ occurrences of a pattern P can be found in time $\mathcal{O}(|P| + occ)$.

Even if we take into account the time for constructing the suffix tree, this is asymptotically as fast as Knuth–Morris–Pratt for a single pattern and Aho–Corasick for multiple patterns.

However, the primary use of suffix trees is in indexed string matching, where we can afford to spend a lot of time in preprocessing the text, but must then answer queries very quickly.

Approximate String Matching

Several approximate string matching algorithms achieving $\mathcal{O}(kn)$ worst case time complexity are based on suffix trees.

Filtering algorithms that reduce approximate string matching to exact string matching such as partitioning the pattern into $k+1$ factors, can use suffix trees in the filtering phase.

Another approach is to generate all strings in the k-neighborhood of the pattern, i.e., all strings within edit distance k from the pattern and search for them in the suffix tree.

The best practical algorithms for indexed approximate string matching are hybrids of the last two approaches. For example, partition the pattern into $\ell \leq k+1$ factors and find approximate occurrences of the factors with edit distance $\lfloor k/\ell \rfloor$ using the neighborhood method in the filtering phase.

Text Statistics

Suffix tree is useful for computing all kinds of statistics on the text. For example:

- Every locus in the suffix tree represents a factor of the text and, vice versa, every factor is represented by some locus. Thus the number of distinct factors in the text is exactly the number of distinct locuses, which can be computed by a traversal of the suffix tree in $\mathcal{O}(n)$ time even though the resulting value is typically $\Theta(n^2)$.

- The longest repeating factor of the text is the longest string that occurs at least twice in the text. It is represented by the deepest internal node in the suffix tree.

Generalized Suffix Tree

A generalized suffix tree of two strings S and T is the suffix tree of the string $S£T\$$, where £ and \$ are symbols that do not occur elsewhere in S and T.

Each leaf is marked as an S-leaf or a T-leaf according to the starting position of the suffix it represents. Using a depth first traversal, we determine for each internal node if its subtree contains only S-leafs, only T-leafs, or both. The deepest node that contains both represents the longest common factor of S and T. It can be computed in linear time.

The generalized suffix tree can also be defined for more than two strings.

AC Automaton for the Set of Suffixes

As already mentioned, a suffix tree with suffix links is essentially an Aho–Corasick automaton for the set of all suffixes.

- We saw that it is possible to follow suffix link / failure transition from any locus, not just from suffix tree nodes.

- Following such an implicit suffix link may take more than a constant time, but the total time during the scanning of a string with the automaton is linear in the length of the string. This can be shown with a similar argument as in the construction algorithm.

Thus suffix tree is asymptotically as fast to operate as the AC automaton, but needs much less space.

Matching Statistics

The matching statistics of a string $S[0..n)$ with respect to a string T is an array $MS[0..n)$, where $MS[i]$ is a pair (ℓ_i, p_i) such that

1. $S[i..i+\ell_i)$ is the longest prefix of S_i that is a factor of T, and
2. $T[p_i..p_i+\ell_i) = S[i..i+\ell_i)$.

Matching statistics can be computed by using the suffix tree of T as an AC-automaton and scanning S with it.

- If before reading $S[i]$ we are at the locus (v, d) in the automaton, then $S[i-d..i) = T[j..j+d)$, where $j = start(v)$. If reading $S[i]$ causes a failure transition, then $MS[i-d] = (d, j)$.
- Following the failure transition decrements d and thus increments $i-d$ by one. Following a normal transition/edge, increments both i and d by one, and thus $i-d$ stays the same. Thus all entries are computed.

From the matching statistics, we can easily compute the longest common factor of S and T. Because we need the suffix tree only for T, this saves space compared to a generalized suffix tree.

Matching statistics are also used in some approximate string matching algorithms.

LCA Preprocessing

The lowest common ancestor (LCA) of two nodes u and v is the deepest node that is an ancestor of both u and v. Any tree can be preprocessed in linear time so that the LCA of any two nodes can be computed in constant time. The details are omitted here.

- Let w_i and w_j be the leaves of the suffix tree of T that represent the suffixes T_i and T_j. The lowest common ancestor of w_i and w_j represents the longest common prefix of T_i and T_j. Thus
$$lcp(T_i, T_j) = depth(LCA(w_i, w_j)),$$
which can be computed in constant time using the suffix tree with LCA preprocessing.

- The longest common prefix of two suffixes S_i and T_j from two different strings S and T is called the longest common extension. Using the generalized suffix tree with LCA preprocessing, the longest common extension for any pair of suffixes can be computed in constant time.

Some $\mathcal{O}(kn)$ worst case time approximate string matching algorithms use longest common extension data structures.

Longest Palindrome

A palindrome is a string that is its own reverse. For example, `saippuakauppias` is a palindrome.

We can use the LCA preprocessed generalized suffix tree of a string T and its reverse T^R to find the longest palindrome in T in linear time.

- Let k_i be the length of the longest common extension of T_{i+1} and T^R_{n-i}, which can be computed in constant time. Then $T[i-k_i..i+k_i]$ is the longest odd length palindrome with the middle at i.
- We can find the longest odd length palindrome by computing k_i for all $i \in [0..n)$ in $\mathcal{O}(n)$ time.
- The longest even length palindrome can be found similarly in $\mathcal{O}(n)$ time. The longest palindrome overall is the longer of the two.