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 3$^m$ transitions that can find all approximate occurrences in $O(n)$ time. However, the space and construction time of the automaton can be too big to be practical.

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

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

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. \qed

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 $r = \lceil m/(k+1) \rceil$.

- The expected number of exact occurrences of a random string of length $r$ in a random text of length $n$ is at most $n/r^k$.

- The expected total verification time is at most $O\left(\frac{m^2(k+1)n}{\sigma^r}\right) \leq O\left(\frac{m^3n}{\sigma^r}\right)$.

This is $O(n)$ if $r \geq 3\log m$.

- The condition $r \geq 3\log m$ is satisfied when $(k+1) \leq m/(3\log m + 1)$.

**Theorem 3.24:** The average case time complexity of the Baeza-Yates-Perleberg algorithm is $O(n)$ when $k \leq m/(3\log m + 1) - 1$.

**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 algorithm has two phases:

**Filtration:** Search the text for exact occurrences of the pattern factors $P_i$. Using the Aho-Corasick algorithm this takes $O(n)$ time for a constant alphabet.

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

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

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 $\Omega(n(k + \log m)/m)$, and there exists a filtering algorithm matching this bound.

**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 $C \subseteq [0..n]$, we write $T_C = \{ T_i | i \in C \}$. 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 $T = \sigma^n$ or $T = \Sigma X$ for any string $X$.

- A basic trie has $\Theta(n^2)$ nodes for most texts, which is too much.
- A compact trie with $O(n)$ nodes and an ordered array with $n+1$ entries have linear size.
- A compact ternary trie has $O(n)$ 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.

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

- 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 $O(|P| + occ)$, where occ is the number of occurrences.

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

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

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 $d = \text{depth}(u)$ and $child(u, T[i+d]) = \perp$ do
   5. $u \leftarrow child(u, T[i+d])$; $d \leftarrow d+1$
   6. while $d < \text{depth}(u)$ and $T[start(u) + d] = T[i+d]$ do $d \leftarrow d+1$
   7. if $d < \text{depth}(u)$ then // (u,d) is in the middle of an edge
      8. $u \leftarrow \text{CreateNode}(u,d)$
   9. $\text{CreateLeaf}(i, u)$
10. $u \leftarrow root$; $d \leftarrow 0$

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

**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, $T[n] = \$ \text{ and } T_i = T[i..n]$ 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:** $T = \text{banana}$.

```
          $\$  
          / 
        /   
      /     
    $a$      $n$
   /     
  /       
$n$ $a$ $n$ $a$
   
```

**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.

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

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

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:

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

**Suffix Links**

The key to efficient suffix tree construction are suffix links:

- $\text{slink}(u)$ 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 $T = \text{banana}$ with internal node suffix links.

```
          $\$  
          / 
        /   
      /     
    $a$      $n$
   /     
  /       
$n$ $a$ $n$ $a$
   
```

```
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, $T[i+1..j]$ and $T[i+1..j]$ are factors of $T$ too, and thus there must be a branching node $v$ representing $T[i+1..j]$. □

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

**Algorithm 4.5:** McCreight

**Input:** text $T[0..n]$ ($T[n] = \$)$

**Output:** suffix tree of $T$: root, child, parent, depth, etc.

1. $v \leftarrow CreateNode(u, d)$
2. $CreateLeaf(i, u)$
3. if $slink(u) = \bot$ then ComputeSlink($u$)
4. $v \leftarrow slink(u)$; $d \leftarrow d - 1$

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 nodes are created during the brute force construction.

**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 $O(n)$ 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 $slink(u)$ 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 $O(n)$.

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

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: $T_i, T_{i-1}, \ldots, T_0$.

- 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 impractical.

- 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 occurrences of a pattern $P$ can be found in time $O(|P| + occ)$.

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 $O(kn)$ 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 $O(n)$ 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.

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.

Generalized Suffix Tree

A generalized suffix tree of two strings $S$ and $T$ is the suffix tree of the string $S \cdot T$, where $\cdot$ is a symbol 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 $S$-leaves, only $T$-leaves, 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.

Matching Statistics

The matching statistics of a string $S[0..n]$ with respect to a string $T$ is an array $MS[S[0..n]]$, where $MS[S]$ is a pair $(i, k)$ such that:
1. $S[i..i+k]$ is the longest prefix of $S$, that is a factor of $T$, and
2. $T[p..p+k] = S[i..i+k]$.

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 $S[i..i+k] = T[j..j+d]$, where $j = \text{start}(S[i])$. If reading $S[i]$ causes a failure transition, then $MS[S[i]] = (d, j)$.

- 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.

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 $T_{n-i}$, which can be computed in constant time. Then $T[i..i+k_i]$ 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 $O(n)$ time.

- The longest even length palindrome can be found similarly in $O(n)$ time. The longest palindrome overall is the longer of the two.