On an integer alphabet, when $m \leq w$:

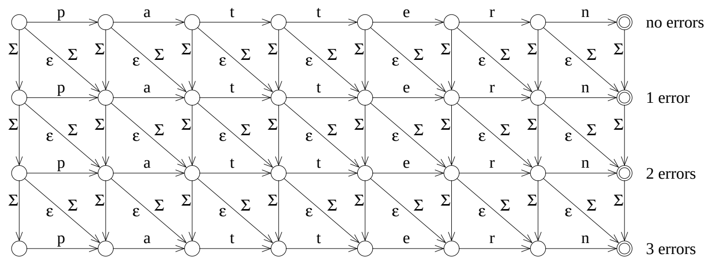
- Pattern preprocessing time is $\mathcal{O}(m + \sigma)$.
- Search time is $\mathcal{O}(n)$.

When $m > w$, we can store each bit vector in $\lceil m/w \rceil$ machine words:

- The worst case search time is $\mathcal{O}(n\lceil m/w \rceil)$.
- Using Ukkonen's cut-off heuristic, it is possible reduce the average case search time to $\mathcal{O}(n\lceil k/w \rceil)$.

There are also algorithms based on bitparallel simulation of a nondeterministic automaton.

Example 4.22: $P = \texttt{pattern}$, $k = 3$



- The algorithm of Wu and Manber uses a bit vector for each row. It can be seen as an extension of Shift-And. The search time complexity is $\mathcal{O}(kn\lceil m/w \rceil)$.
- The algorithm of Baeza-Yates and Navarro uses a bit vector for each diagonal, packed into one long bitvector. The search time complexity is $\mathcal{O}(n\lceil km/w \rceil)$.

Baeza-Yates–Perleberg Filtering Algorithm

A *filtering algorithm* for approximate strings 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*, and it is *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 4.23: Let $P_1P_2 \ldots P_{k+1} = P$ 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. $\square$

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 4.24: 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.

5. 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 a *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 $T_{[0..n]}$.

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 $dp(T_{[0..n]}) = \Theta(n^2)$, for example, when $T = \mathtt{a}^n$ or $T = XX$ for any string X.

- A basic trie with $\mathcal{O}(||T_{[0..n]}||)$ nodes or a trie with compact leaf edges with $\mathcal{O}(dp(T_{[0..n]}))$ nodes could be too large.
- 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 $\Omega(dp(T_{[0..n]}))$ time.

Suffix Tree

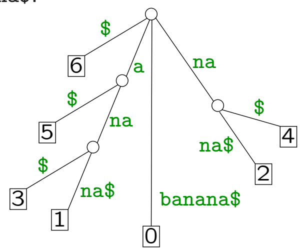
The suffix tree of a text T is the compact trie of the set $T_{[0..n]}$ of all suffixes of T.

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 5.1: $T = \texttt{banana\$}$.



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 $\Theta(dp(T_{[0..n]}))$. This is later modified 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 in Algorithm 2.2 (trie construction) except it has been modified to work on a compact trie instead of a trie.

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 $\bot$ if u has no such child.

parent(u) is the parent of u.

depth(u) is the length of the string S_u represented by 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 *pair* (u, d) with $\mathit{depth}(\mathit{parent}(u)) < d \le \mathit{depth}(u)$ represents a position on the edge $(\mathit{parent}(u), u)$ corresponding to the string $S_{(u,d)} = T[\mathit{start}(u) \ldots \mathit{start}(u) + d)$.

Note that the positions (u, d) correspond to nodes in the uncompact trie.

Algorithm 5.2: Brute force suffix tree construction
Input: text $T[0..n]$ $(T[n] = \$)$
Output: suffix tree of T: $root$, *child*, *parent*, *depth*, *start*

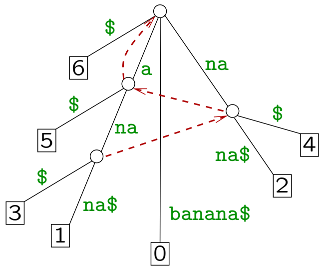
```
create new node root; depth(root) ← 0
u ← root; d ← 0
for i ← 0 to n do        // insert suffix T_i
    while d = depth(u) and child(u, T[i + d]) ≠ ⊥ do
        u ← child(u, T[i + d]); d ← d + 1
        while d < depth(u) and T[start(u) + d] = T[i + d] do d ← d + 1
    if d < depth(u) then        // we are in the middle of an edge
        create new node v
        start(v) ← i; depth(v) ← d
        p ← parent(u)
        child(v, T[start(u) + d]) ← u; parent(u) ← v
        child(p, T[i + depth(p)]) ← v; parent(v) ← p
        u ← v
    create new leaf w        // w represents suffix T_i
    start(w) ← i; depth(w) ← n − i + 1
    child(u, T[i + d]) ← w; parent(w) ← u
    u ← root; d ← 0
```


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



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

Lemma 5.4: 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 $T[i..j)b$ occur somewhere in T. Then, $T[i + 1..j)a$ and $T[i + 1..j)b$ occur in T too, and thus there must be a branching node v representing $T[i + 1..j)$. $\square$

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