- The search time of BDM and BOM is $\mathcal{O}(n(\log_\sigma m)/m)$, which is optimal on *average*. (BNDM is optimal only when $m \leq w$.)
- MP and KMP are optimal in the *worst case*.
- There are also algorithms that are optimal in *both* cases. They are based on similar techniques, but we will not describe them here.

Karp–Rabin

The Karp–Rabin algorithm uses a *hash function* $H : \Sigma^* \to [0..q) \subset \mathbb{N}$ for strings. It computes computes the hash values or *fingerprints* $H(P)$ and $H(T[j..j+m))$ for all $j \in [0..n-m]$.

- If $H(P) \neq H(T[j..j+m))$, then we must have $P \neq T[j..j+m)$.
- If $H(P) = H(T[j..j+m))$, the algorithm compares P and $T[j..j+m)$ in brute force manner. If $P \neq T[j..j+m)$, this is a *false positive*.

A good hash function has two important properties:

- False positives are rare.
- Given $H(a\alpha)$, a and b, where $a, b \in \Sigma$ and $\alpha \in \Sigma^*$, we can quickly compute $H(\alpha b)$. This is a called *rolling* or *sliding window* hash function.

The latter property is essential for fast computation of $H(T[j..j+m))$ for all j.

The hash function used by Karp–Rabin is

$$H(c_0c_1c_2\ldots c_{m-1}) = (c_0r^{m-1} + c_1r^{m-2} + \cdots + c_{m-2}r + c_{m-1}) \bmod q$$

This is a rolling hash function:

$$H(\alpha) = (H(a\alpha) - ar^{m-1}) \bmod q$$
$$H(\alpha b) = (H(\alpha)\cdot r + b) \bmod q$$

which follows from the rules of *modulo arithmetic*:

$$(x+y) \bmod q = ((x \bmod q) + (y \bmod q)) \bmod q$$
$$(xy) \bmod q = ((x \bmod q)(y \bmod q)) \bmod q$$

The parameters q and r have to be chosen with some care to ensure that false positives are rare.

- The original choice is $r = \sigma$ and q is a large *prime*.
- Another possibility is $q = 2^w$, where w is the machine word size, and r is a small prime ($r = 37$ has been suggested). This is faster in practice, because it avoids slow modulo operations.
- The hash function can be *randomized* by choosing q or r randomly. Furthermore, we can change q or r when a false positive occurs.

Algorithm 3.17: Karp-Rabin
Input: text $T = T[0\ldots n)$, pattern $P = P[0\ldots m)$
Output: position of the first occurrence of P in T
(1) Choose q and r; $s \leftarrow r^{m-1} \bmod q$
(2) $hp \leftarrow 0; ht \leftarrow 0$
(3) *for* $i \leftarrow 0$ *to* $m-1$ *do* $hp \leftarrow (hp\cdot r + P[i]) \bmod q$ // $hp = H(P)$
(4) *for* $j \leftarrow 0$ *to* $m-1$ *do* $ht \leftarrow (ht\cdot r + T[j]) \bmod q$
(5) *for* $j \leftarrow 0$ to $n-m-1$ *do*
(6) *if* $hp = ht$ *then if* $P = T[j\ldots j+m)$ *then* return j
(7) $ht \leftarrow ((ht - T[j]\cdot s)\cdot r + T[j+m]) \bmod q$
(8) *if* $hp = ht$ *then if* $P = T[j\ldots j+m)$ *then* return j
(9) return n

On an integer alphabet:

- The worst case time complexity is $\mathcal{O}(mn)$.
- The average case time complexity is $\mathcal{O}(m+n)$.

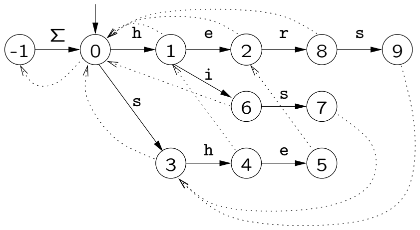
Karp–Rabin is not competitive in practice, but hashing can be a useful technique in other contexts.

Aho–Corasick Algorithm

Given a text T and a set $\mathcal{P} = \{P_1.P_2, \ldots, P_k\}$ of patterns, the *multiple exact string matching* problem asks for the occurrences of all the patterns in the text. The Aho–Corasick algorithm is an extension of the Morris–Pratt algorithm for multiple exact string matching.

Aho–Corasick uses the trie $trie(\mathcal{P})$ as an *automaton* and augments it with a failure function similar to the Morris-Pratt failure function.

Example 3.18: Aho–Corasick automaton for $\mathcal{P} = \{\texttt{he}, \texttt{she}, \texttt{his}, \texttt{hers}\}$.



Algorithm 3.19: Aho–Corasick
Input: text T, pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: all pairs (i, j) such that P_i occurs in T ending at j.
(1) Construct AC automaton
(2) $v \leftarrow root$
(3) *for* $j \leftarrow 0$ *to* $n - 1$ *do*
(4) *while child*$(v, T[j]) = \perp$ *do* $v \leftarrow$ *fail*(v)
(5) $v \leftarrow$ *child*$(v, T[j])$
(6) *for* $i \in$ *patterns*(v) *do* output (i, j)

Let S_v denote the string that node v represents.

- $root$ is the root and *child*() the child function of the trie.
- *fail*$(v) = u$ such that S_u is the *longest proper suffix* of S_v represented by any node.
- *patterns*(v) is the set of pattern indices i such that P_i is a *suffix* of S_v.

At each stage, the algorithm computes the node v such that S_v is the longest suffix of $T[0..j]$ represented by any node.

Algorithm 3.20: Aho–Corasick trie construction
Input: pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: AC trie: $root$, *child*() and *patterns*().
(1) Create new node $root$
(2) *for* $i \leftarrow 1$ *to* k *do*
(3) $v \leftarrow root$; $j \leftarrow 0$
(4) *while child*$(v, P_i[j]) \neq \perp$ *do*
(5) $v \leftarrow$ *child*$(v, P_i[j])$; $j \leftarrow j + 1$
(6) *while* $j < |P_i|$ *do*
(7) Create new node u
(8) *child*$(v, P_i[j]) \leftarrow u$
(9) $v \leftarrow u$; $j \leftarrow j + 1$
(10) *patterns*$(v) \leftarrow \{i\}$

This is the standard trie construction (Algorithm 2.2) except *rep*() has been replaced by *patterns*():

- The creation of a new node v initializes *patterns*(v) to $\emptyset$.
- At the end, $i \in$ *patterns*(v) iff v represents P_i.

Algorithm 3.21: Aho–Corasick automaton construction
Input: AC trie: $root$, *child*() and *patterns*()
Output: AC automaton: *fail*() and updated AC trie
(1) Create new node $fallback$
(2) *for* $c \in \Sigma$ *do child*$(fallback, c) \leftarrow root$
(3) *fail*$(root) \leftarrow fallback$
(4) $queue \leftarrow \{root\}$
(5) *while* $queue \neq \emptyset$ *do*
(6) $u \leftarrow$ popfront$(queue)$
(7) *for* $c \in \Sigma$ such that *child*$(u, c) \neq \perp$ *do*
(8) $v \leftarrow$ *child*(u, c)
(9) $w \leftarrow$ *fail*(u)
(10) *while child*$(w, c) = \perp$ *do* $w \leftarrow$ *fail*(w)
(11) $fail(v) \leftarrow$ *child*(w, c)
(12) *patterns*$(v) \leftarrow$ *patterns*$(v) \cup$ *patterns*(*fail*(v))
(13) pushback$(queue, v)$

The algorithm does a *breath first traversal* of the trie. This ensures that correct values of *fail*() and *patterns*() are already computed when needed.

fail(v) is correctly computed on lines (8)–(11):

- The nodes that represent suffixes of S_v that are exactly *fail*$^*(v) = \{v, \textit{fail}(v), \textit{fail}(\textit{fail}(v)), \ldots, root\}$.
- Let $u = \textit{parent}(v)$ and $\textit{child}(u, c) = v$. Then S is a suffix of S_u iff Sc is suffix of S_v. Thus
  - If $w \in \textit{fail}^*(v)$, then $\textit{parent}(\textit{fail}(v)) \in \textit{fail}^*(u)$.
  - If $w \in \textit{fail}^*(u)$ and $\textit{child}(w, c) \neq \perp$, then $\textit{child}(w, c) \in \textit{fail}^*(v)$.
- Therefore, $\textit{fail}(v) = \textit{child}(w, c)$, where w is the first node in *fail*$^*(u)$ other than u such that $\textit{child}(w, c) \neq \perp$.

patterns(v) is correctly computed on line (12):

$$\begin{aligned}\textit{patterns}(v) &= \{i \mid P_i \text{ is a suffix of } S_v\} \\ &= \{i \mid P_i = S_w \text{ and } w \in \textit{fail}^*(v)\} \\ &= \{i \mid P_i = S_v\} \cup \textit{patterns}(\textit{fail}(v))\end{aligned}$$

Assuming σ is constant:

- The search time is $\mathcal{O}(n)$.
- The space complexity is $\mathcal{O}(m)$, where $m = ||\mathcal{P}||$.
  - Implementation of *patterns*() requires care (exercise).
- The preprocessing time is $\mathcal{O}(m)$, where $m = ||\mathcal{P}||$.
  - The only non-trivial issue is the while-loop on line (10).
  - Let $root, v_1, v_2, \ldots, v_\ell$ be the nodes on the path from root to a node representing a pattern P_i. Let $w_j = \textit{fail}(v_j)$ for all j. Let *depth*(v) be the depth of a node v ($\textit{depth}(root) = 0$).
  - When computing w_j, $\textit{depth}(w_j) = \textit{depth}(w_{j-1}) + 1$ before line (10) and $\textit{depth}(w_j) \leq \textit{depth}(w_{j-1}) + 1 - t_j$ after line (10), where t_j is the number of rounds in the while-loop.
  - Thus, the total number of rounds is at most $\ell = |P_i|$ when computing $w_1, w_2, \ldots, w_\ell$, and at most $||\mathcal{P}||$ over the whole algorithm.

The analysis when σ is not constant is left as an exercise.

4. Approximate String Matching

Often in applications we want to search a text for something that is *similar* to the pattern but not necessarily exactly the same.

To formalize this problem, we have to specify what does "similar" mean. This can be done by defining a *similarity* or a *distance measure*.

A natural and popular distance measure for strings is the *edit distance*, also known as the *Levenshtein distance*.

Edit distance

The *edit distance* $ed(A, B)$ of two strings A and B is the minimum number of edit operations needed to change A into B. The allowed edit operations are:

S *Substitution* of a single character with another character.

I *Insertion* of a single character.

D *Deletion* of a single character.

Example 4.1: Let $A =$ `Lewensteinn` and $B =$ `Levenshtein`. Then $ed(A, B) = 3$.

The set of edit operations can be described

with an *edit sequence*: `NNSNNNINNNND`
or with an *alignment*:

```
Lewens-teinn
Levenshtein-
```

In the edit sequence, N means No edit.