Karp–Rabin

The Karp–Rabin hash function (Definition 1.39) was originally developed for solving the exact string matching problem. The idea is to compute 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*.

The text factor fingerprints are computed in a *sliding window* fashion. The fingerprint for $T[j+1..j+1+m) = \alpha T[j+m]$ is computed from the fingerprint for $T[j..j+m) = T[j]\alpha$ in constant time using Lemma 1.40:

$$\begin{aligned} H(T[j+1..j+1+m)) &= (H(T[j]\alpha) - H(T[j]) \cdot r^{m-1}) \cdot r + H(T[j+m])) \bmod q \\ &= (H(T[j..j+m)) - T[j] \cdot r^{m-1}) \cdot r + T[j+m]) \bmod q \ . \end{aligned}$$

A hash function that supports this kind of sliding window computation is known as a *rolling hash function*.

Algorithm 2.17: 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

Choose q and r; $s \leftarrow r^{m-1} \bmod q$
$hp \leftarrow 0; ht \leftarrow 0$
for $i \leftarrow 0$ *to* $m-1$ *do* $hp \leftarrow (hp \cdot r + P[i]) \bmod q$ // $hp = H(P)$
for $j \leftarrow 0$ *to* $m-1$ *do* $ht \leftarrow (ht \cdot r + T[j]) \bmod q$
for $j \leftarrow 0$ to $n-m-1$ *do*
    if $hp = ht$ *then if* $P = T[j\ldots j+m)$ *then* return j
    $ht \leftarrow ((ht - T[j] \cdot s) \cdot r + T[j+m]) \bmod q$
if $hp = ht$ *then if* $P = T[j\ldots j+m)$ *then* return j
return n

On an integer alphabet:

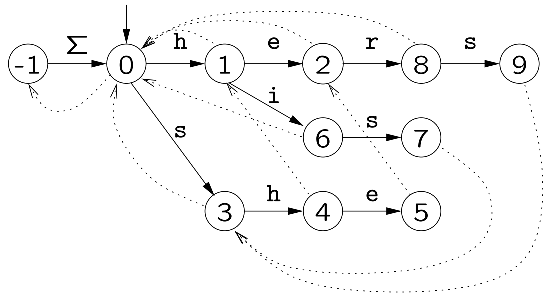
- The worst case time complexity is $\mathcal{O}(mn)$.
- The average case time complexity is $\mathcal{O}(m+n)$.

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 2.18: Aho–Corasick automaton for $\mathcal{P} = \{\mathtt{he}, \mathtt{she}, \mathtt{his}, \mathtt{hers}\}$.



Algorithm 2.19: 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.

Construct AC automaton
$v \leftarrow root$
for $j \leftarrow 0$ *to* $n-1$ *do*
    while child$(v, T[j]) = \perp$ *do* $v \leftarrow$ *fail*(v)
    $v \leftarrow$ *child*$(v, T[j])$
    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 trie node u.
- *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 2.20: Aho–Corasick trie construction
Input: pattern set $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$.
Output: AC trie: $root$, *child*() and *patterns*().

Create new node $root$
for $i \leftarrow 1$ *to* k *do*
  $v \leftarrow root$; $j \leftarrow 0$
  while *child*$(v, P_i[j]) \neq \perp$ *do*
    $v \leftarrow$ *child*$(v, P_i[j])$; $j \leftarrow j + 1$
  while $j < |P_i|$ *do*
    Create new node u
    child$(v, P_i[j]) \leftarrow u$
    $v \leftarrow u$; $j \leftarrow j + 1$
  patterns$(v) \leftarrow \{i\}$

This is the standard trie insertion (Algorithm 1.3) except for the computation of *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 2.21: 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*$^*(v) = \{v,$ *fail*(v), *fail*(*fail*$(v)), \ldots, root\}$.
- Let $u =$ *parent*(v) and *child*$(u, c) = v$. Then $S_v = S_u c$ and a string S is a suffix of S_u iff Sc is suffix of S_v. Thus for any node w
  - If $w \in$ *fail*$^*(v)$, then *parent*(*fail*$(v)) \in$ *fail*$^*(u)$.
  - If $w \in$ *fail*$^*(u)$ and *child*$(w, c) \neq \perp$, then *child*$(w, c) \in$ *fail*$^*(v)$.
- Therefore, *fail*$(v) =$ *child*(w, c), where w is the first node in *fail*$^*(u)$ other than u such that *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 $w_j =$ *fail*(v_j) for all j. Let *depth*(v) be the depth of a node v (*depth*$(root) = 0$).
  - When processing v_j and computing $w_j =$ *fail*(v_j), we have *depth*$(w_j) =$ *depth*$(w_{j-1}) + 1$ before line (10) and *depth*$(w_j) \leq$ *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 in the while-loop when processing the nodes $v_1, v_2, \ldots, v_\ell$ is at most $\ell = |P_i|$, and thus over the whole algorithm at most $||\mathcal{P}||$.

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