Boyerov-Moorov algoritmus 1977

```
1  i=0;
2  while(i<=n-m){
3     j=m-1;
4     while(j>=0 && P[j]==T[i+j]){
5        j--;
6     }
7     if(j<0){ print i; }
8     i += skip;
9  }
```

Ak skip = 1, triviálny algoritmus. Väčšie (ale bezpečné) skoky:

– pravidlo zlého znaku

– pravidlo dobrého sufixu

Pravidlo zlého znaku

Nech $R[x]$ je index posledného výskytu x v P

alebo -1 ak x sa nevyskytuje v P

Ak $P[j+1..m-1] = T[i+j+1..i+m-1]$ a $P[j] \neq T[i+j] = x$:

Ak $R[x] < j$, skip $= j - R[x]$

inak skip $= 1$.

Horspoolov algoritmus

Použi pravidlo zlého znaku na $x = T[i+m-1]$

$R[x]$ je index posledného výskytu x v $P[0..m-2]$

skip $= m - 1 - R[x]$

$O(mn + \sigma)$ najhorší prípad, $O(n/\sigma + m + \sigma)$ priemerný prípad

Good suffix rule

$P \sim Q$ iff P is a suffix Q or Q is a suffix of P

Skip $\gamma[j]$:

$$\gamma[j] = m - \max\{k \mid 0 \leq k < m \wedge P[j+1..m-1] \sim P[0..k-1]\}$$

Case A: last occurrence of $P[j+1..m-1]$ in $P[0..m-2]$

Case B (if A does not apply):

longest proper suffix of $P[j+1..m]$ which is a prefix of P

the same as: longest proper suffix of P which is a prefix of P

Preprocessing using MP (sp^R: sp for P^R)

$sp[i]$: longest proper suffix of $P[0..i-1]$ which is a prefix of P.

$sp^R[i]$: longest proper prefix of $P[m-i..m-1]$ which is a suffix of P

Good suffix rule preprocessing

```
1  compute sp;
2  compute spr;
3  for (j=-1; j<m; j++) {
4     gamma[j] = m-sp[m];
5  }
6  for (i=m; i>=0; i--) {
7     j = m-spr[i]-1;
8     gamma[j] = i-spr[i];
9  }
```

Rabin-Karp algorithm (1987)

Based on hashing function $H : \Sigma^m \rightarrow \{0, 1, \ldots, p-1\}$

```
1  hp = H(P)        // hash pattern P
2  for(i=0; i<=n-m; i++) {
3     ht = H(T[i..i+m-1]);   // hash window of T
4     if(hp == ht) {          // if hashes the same
5        if(P==T[i..i-m+1]) { // check if occurrence
6           write i;
7        }
8     }
9  }
```

Hashing function

Hashing function $H(S) = S \bmod p$.

String S as a number with σ-based numeral system: $\sum_{j=0}^{|S|-1} \sigma^{|S|-j-1} S[j]$

Computation of $H(S)$ using Horner scheme:

$$S = ((S[1]\sigma + S[2])\sigma + S[3])\sigma \ldots$$

Update of hash value after moving window in T:

$$H_i = (\sigma(H_{i-1} - \sigma^{m-1} T[i-1]) + T[i+m-1]) \bmod p.$$

Compute everything modulo p (avoid large numbers)

Horner scheme computation of hashing function

```
1  int H(S,p,sigma){
2      h = 0;
3      for(i=0; i<|S|; i++) {
4        h = h*sigma mod p;
5        h = (h + S[i]) mod p;
6      }
7      return h;
8  }
```

Running time

$O(n + m(k + f))$

k is the number of occurrences

f is the number of false matches s.t. $h_p = h_t \wedge P \neq T[i \ldots i - m + 1]$

Randomized version

As p use random prime number between 1 and nm^2.

Theorem. Assume $\sigma = 2$ and $nm \geq 29$. For a random prime p between 1 and nm^2 the probability of at least one false match between P and T is at most $2.52/m$ and the expected time of the algorithm is $O(n + mk)$.

Shift-and algoritmus Baeza-Yates, Gonnet 1992

Bit paralellism, good for short patterns

Bit matrix $M[0..m-1, 0..n-1]$:

$M(i,j) = 1 \Leftrightarrow P[0 \ldots i] = T[j-i \ldots j]$

Store columns as binary numbers

Binary number $U(x)$ of length m for each $x \in \Sigma$:

$U(x)[i] = 1 \Leftrightarrow P[i] = x$

To compute column j from $j-1$:

$M(i,j) = (M(i-1, j-1) \vee i = 0) \wedge (P[i] = T[j])$

$M(\cdot, j) = ((M(\cdot, j-1) \ << \ 1) \,|\, 1) \,\&\, U(T[j])$

Occurrence ends at $j \iff M(m-1, j) = 1$

Running time: $O(n + m + \sigma)$ if $m <$ register length

BNDM: Backward non-deterministic DAWG matching

- Check suffixes of $T[i..i+m-1]$ (from shorter to longer)
- For each suffix find all occurrences in P (if any)
- If whole $T[i..i+m-1]$ occurs in P, print
- Select longest suffix α which is a prefix of P
- Move P so that α's align

Running time if $m <$ register length

Worst-case: $O(nm + \sigma)$

Average-case: $O(n\frac{\log_\sigma m}{m} + m + \sigma)$

```
1   i=0; while(i<=n-m){
2      j = m-1; last_j = m-1;
3      B = (1<<m)-1;   //m times 1
4      while(B>0 && j>=0){
5         B = (B >> 1) & U[T[i+j]];
6         //B[k]=1 <=> T[i+j..i+m-1]=P[k..k+m-j-1]
7         if(B & 1){   // occurrence of prefix in P?
8            if(j==0) print i;   // occurrence of whole P?
9            else last_j = j;   // move last found prefix
10        }
11        j--;
12     }
13     i += last_j;   //skip to the last prefix
14  }
```

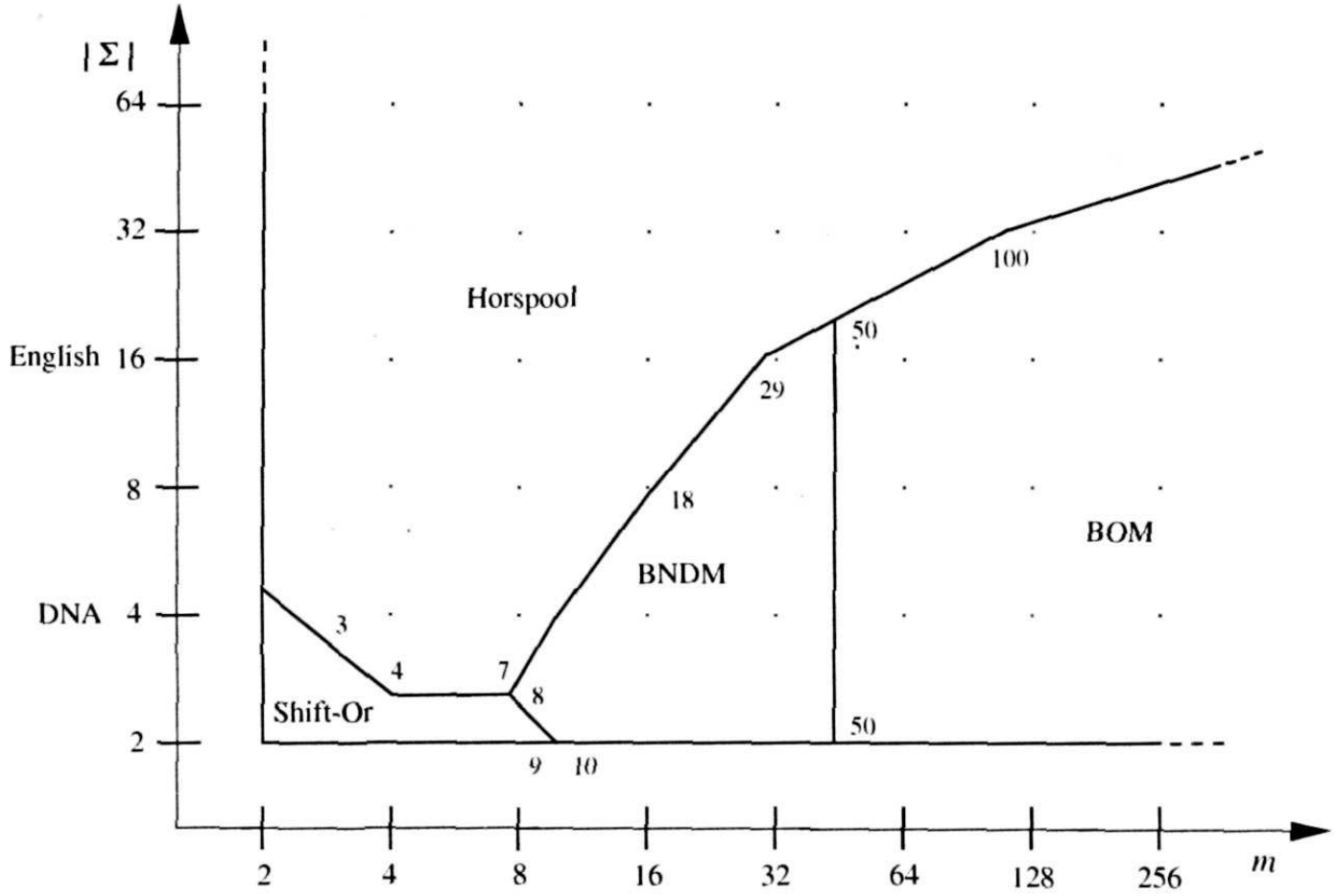
Overview of string matching algorithms

Algorithm	Worst case	Average case	Comp.	Note
Trivial	$O(nm)$	$O(n+m)$	Y	simple
DFA	$O(n+\sigma m)$	$O(n+\sigma m)$	N	real-time
(K)MP	$O(n+m)$	$O(n+m)$	Y	
Boyer Moore	$O(nm), O(n+m)$	$O(\frac{n}{\sigma}+m)$	N	
Rabin-Karp	$O(nm)$	$O(n+mk)$	N	randomized
Shift-and	$O(n+m+\sigma)$	$O(n+m+\sigma)$	N	m-bit register
BNDM	$O(nm+\sigma)$	$O(n\frac{\log_\sigma m}{m}+m+\sigma)$	N	m-bit register

Multiple patterns: Aho Corasick $O((n+m)\log\sigma+k)$

2D patterns: Baker Bird $O((n+m)\log\sigma)$

Fastest algorithm for various m and σ



Navarro, Raffinot (2002) Flexible Pattern Matching in Strings.

Random texts, 32-bit numbers