2-INF-237 Vybrané partie z dátových štruktúr

2-INF-237 Selected Topics in Data Structures

- Instructor: Broňa Brejová
- E-mail: brejova@fmph.uniba.sk
- Office: M163

- Course webpage: http://compbio.fmph.uniba.sk/vyuka/vpds/

String matching (vyhľadávanie vzorky v texte)

Given: pattern (vzorka) P of length m, text T of length n.

Goal: Find all positions $\{i_1, i_2, \ldots\}$ such that $T[i_j..i_j + m - 1] = P$.

Example:

Input: P =ma, T =Ema ma mamu

Output: 1, 4, 7

Input: P =”a ma”, T =Ema ma mamu

Output: 2, 5

Trivial algorithm

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

String matching

- Trivial algorithm $O(nm)$

  Worst case $T = a^n$, $P = a^m$

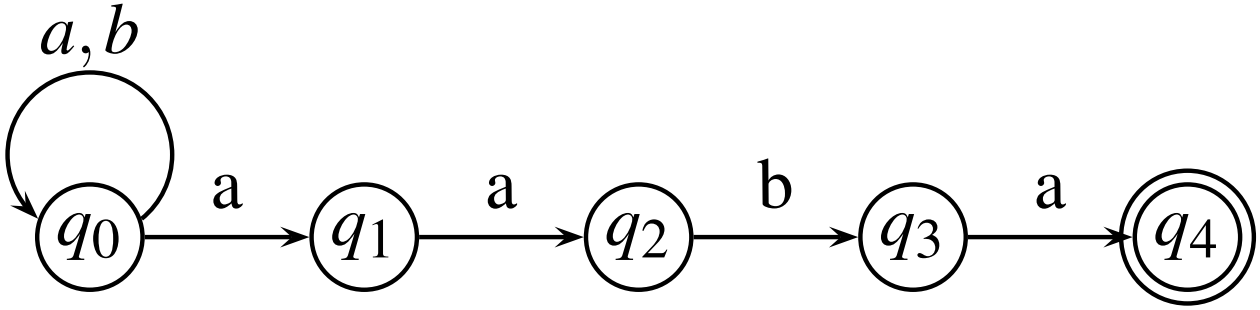
  Average case on random strings $O(n + m)$

- Build suffix tree for T in $O(n)$, then search for P in $O(m)$

- Knuth-Morris-Pratt algorithm $O(n + m)$

Nondeterministic finite automaton for $\{xP|x \in \Sigma^*\}$

P =aaba



The automaton can reach state q_i after reading T

iff suffix of T of length i is a prefix of P.

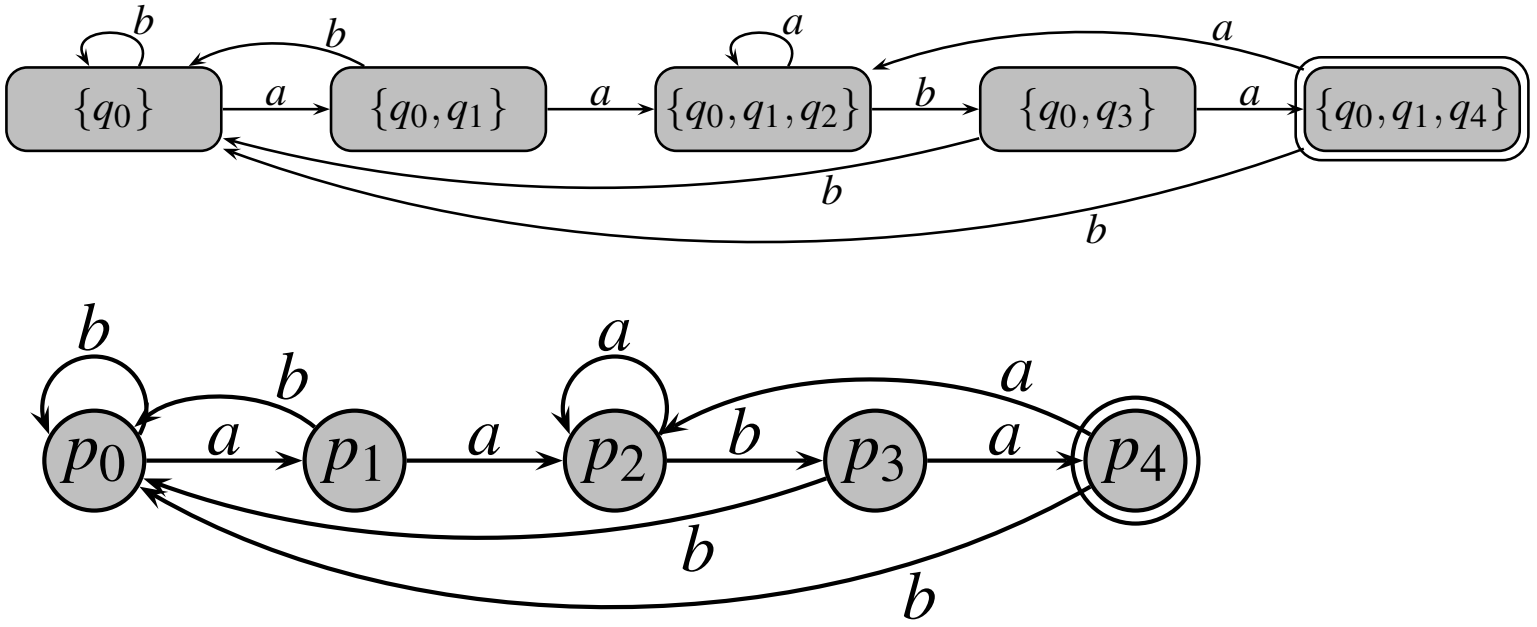
Number of states: $m + 1$

Simulate on a string T in $O(mn)$ time.

Simulation of nondeterministic finite automaton on text T

```
1  S = {0};
2  for(i=0; i<n; i++){
3     S1 = empty_set;
4     foreach state j in S {
5        add delta(j, T[i]) to S1;
6     }
7     if (m in S1){
8        print i-m+1;
9     }
10    S = S1;
11 }
```

Deterministic finite automaton for $\{xP|x \in \Sigma^*\}$



The automaton reaches state p_i after reading T

iff i is the length of the longest suffix of T which is a prefix of P

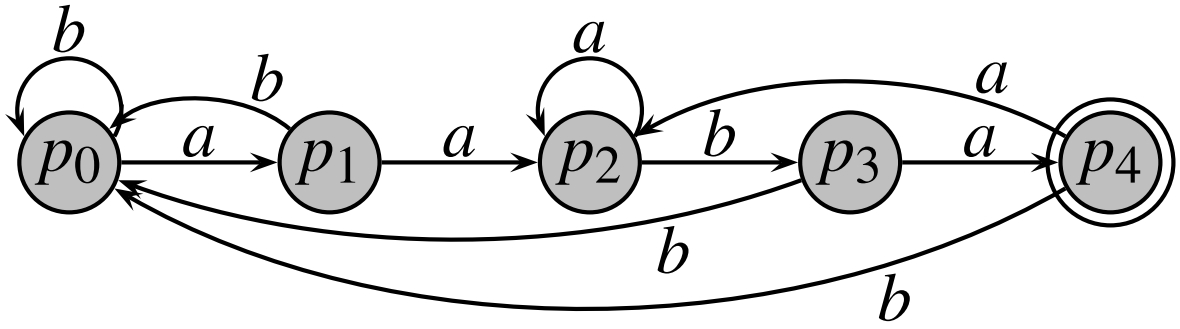
$\Rightarrow$ automaton finishes in state p_m iff T ends with P

Read T letter by letter, update state, print occurrence if state is p_m

Simulation of deterministic finite automaton on text T

```
1  state = 0;
2  for(i=0; i<n; i++){
3     state= delta(state,T[i]);
4     if (state==m){
5        print i-m+1;
6     }
7  }
```

Deterministic finite automaton for $\{xP|x \in \Sigma^*\}$



Number of states: $m + 1$

Simulate on string T in $O(n)$ time.

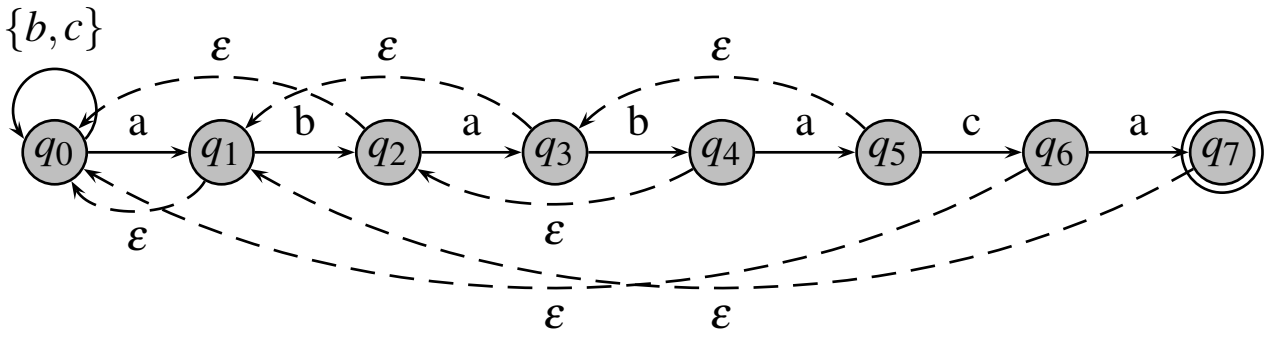
Memory $O(m\sigma)$

Building automaton $O(m\sigma)$ time – optional homework

Overall $O(n + m\sigma)$ time

Morris-Pratt algorithm 1970 $O(m+n)$

P =ababaca



Epsilon transition used when no other option.

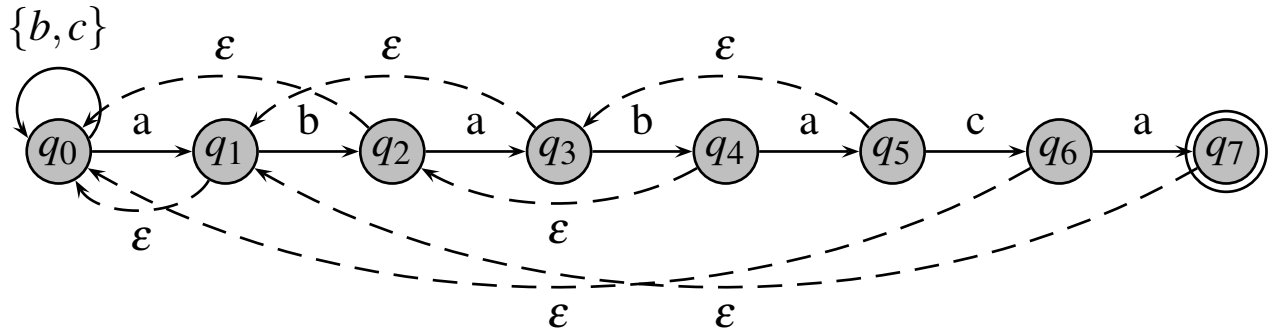
Epsilon transition from state i to $sp[i]$ ($sp[i] < i$).

$sp[i]$ = length of the longest proper suffix of $P[0..i-1]$ which is also a prefix of P.

The automaton reaches state q_i after reading T

iff i is the length of the longest suffix of T which is a prefix of P

Morris-Pratt algorithm



$i, sp[i], sp[sp[i]], \ldots, 0$ form a linked list of all suffixes of $P[0 \ldots i-1]$ which are prefixes of P.

E.g. $i = 5$, $P[0..4] =$ ababa

Linked list q_5, q_3, q_1, q_0

Suffixes/prefixes ababa, aba, a, ε

Next letter x arrives: find longest one that can be followed by x in P

$x =$a, state $q_0 \to q_1$; $x =$b, state $q_3 \to q_4$; $x =$c, state $q_5 \to q_6$

Morris-Pratt algorithm

```
1   state=0;
2   for(i=0; i<n; i++){
3     while(state>0 && T[i]!=P[state]){  //epsilon transitions
4       state=sp[state];
5     }
6     if(T[i]==P[state]){   //move to next state
7       state++;
8     }
9     if(state==m){   //accepting state - print occurrence
10      print i-m+1; state=sp[state];
11    }
12  }
```

Morris-Pratt algorithm preprocessing

```
1   sp[0]=sp[1]=0;
2   j=0;
3   for(i=2; i<=m; i++){
4     // invariant: j=sp[i-1];
5     while(j>0 && P[i-1]!=P[j]) {
6       j=sp[j];
7     }
8     if(P[i-1]==P[j]) {
9       j++;
10    }
11    sp[i]=j;
12  }
```

$sp[i]$ = length of the longest proper suffix of $P[0..i-1]$ which is also a prefix of P.

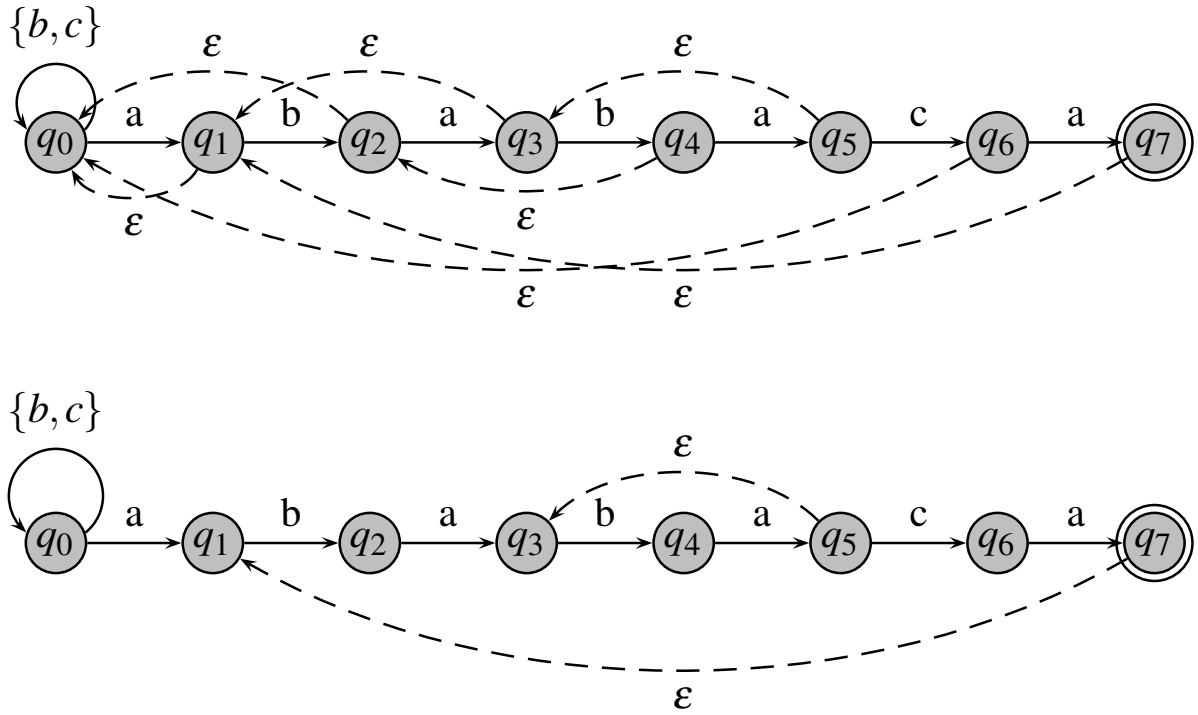
Morris Pratt algorithm 1970

- Build sp table in $O(m)$ time, $O(m)$ memory
- Simulate automaton on T in $O(n)$ time
- Works well for large alphabets

  characters used only for equality testing
- Works well for streaming T,

  but can have $O(m)$ delay between characters

Knuth-Morris-Pratt algorithm, 1977



Longer epsilon transitions:

from state i to $sp_2[i]$, where $sp_2[i]$ is the length of the longest proper sufix $P[0..i-1]$, which is a prefix of P and $P[j] \neq P[i]$

Knuth-Morris-Pratt algorithm, preprocessing

Given values $sp[i]$ from MP algorithm, compute $sp_2[i]$ from KMP

```
1  sp2[0]=0;
2  for(i=1; i<=m; i++) {
3     if(i==m || P[sp[i]]!=P[i]) {
4        sp2[i]=sp[i];
5     }
6     else {
7        sp2[i]=sp2[sp[i]];
8     }
9  }
```

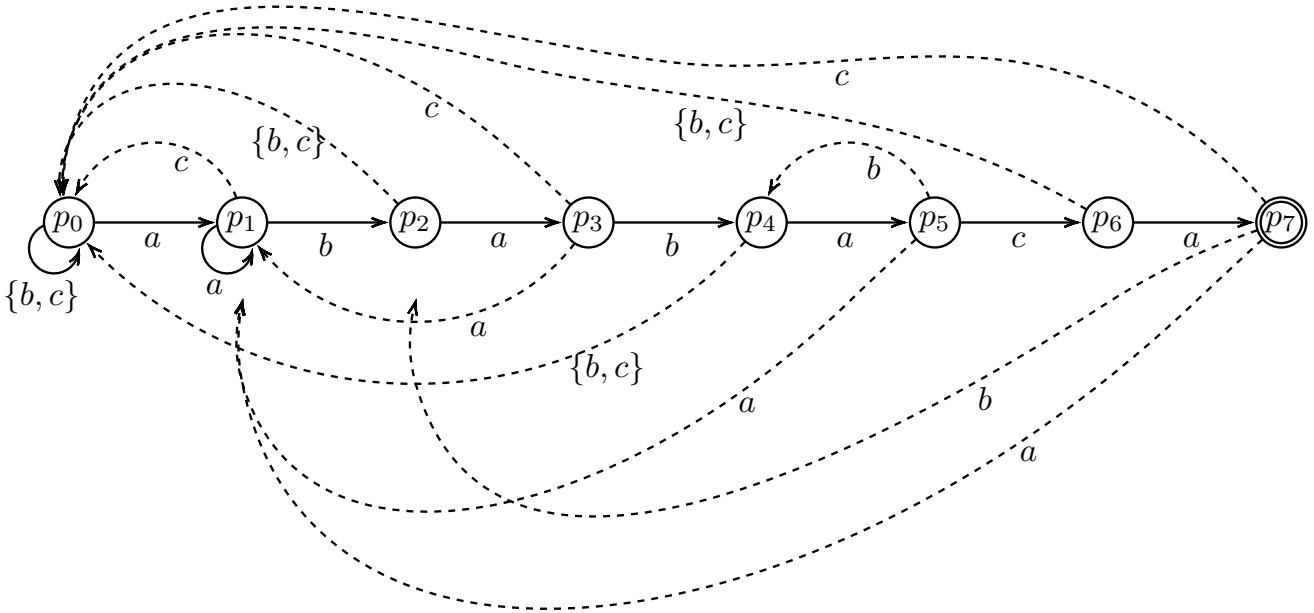
Knuth-Morris-Pratt algorithm, analysis

Theorem: The number of ε transitions in one iteration of KMP algorithm is at most $\log_\phi(m+1)$, where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Note: Overall running time still $O(n+m)$.

Optional homework

Compute DFA in $O(m\sigma)$ time using $sp[i]$ table from MP alg.



DFA reaches state p_i after reading T

iff i is the length of the longest suffix of T which is a prefix of P

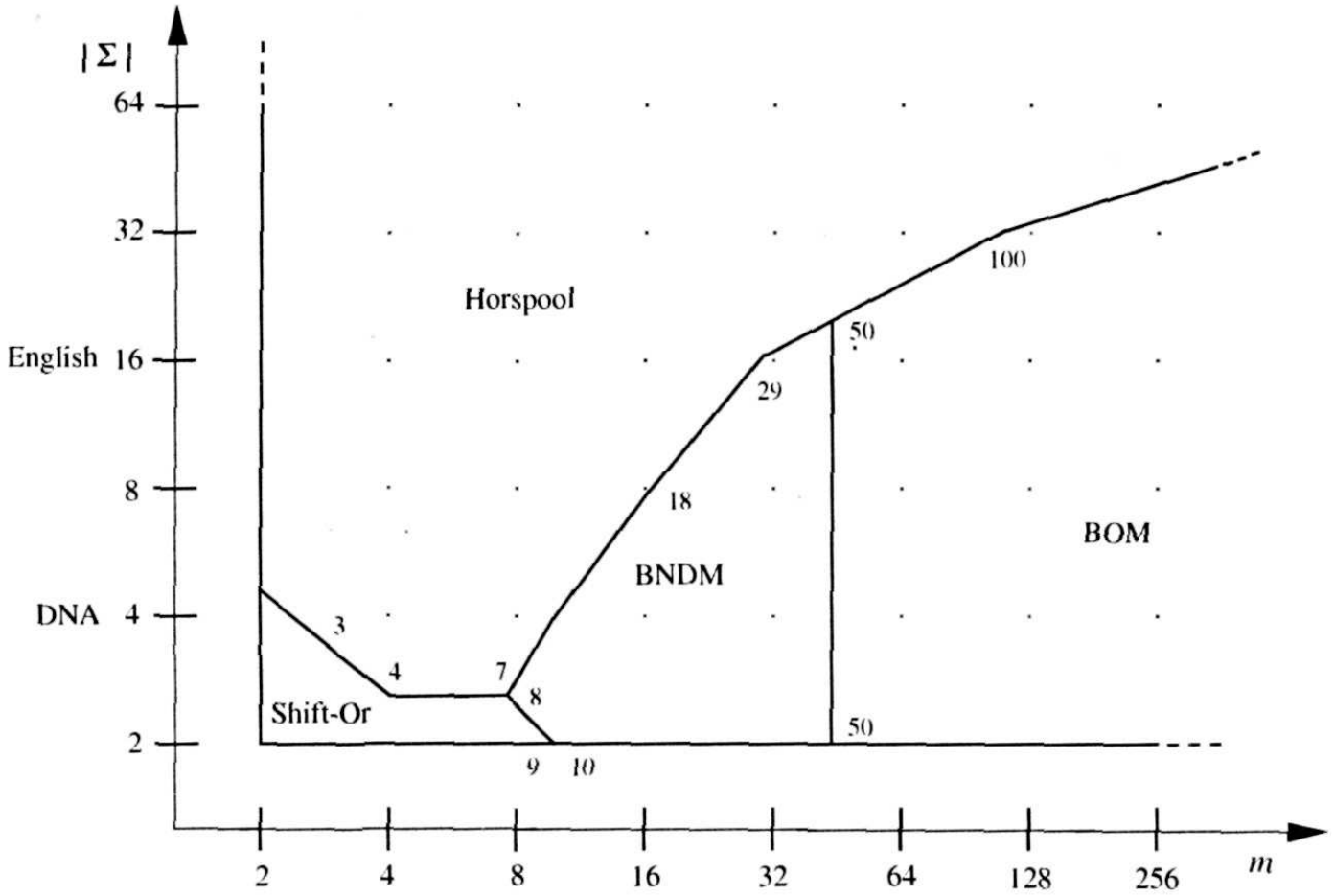
Overview of string matching algorithms

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

Approximate string matching

Hamming distance $d_H(S_1, S_2)$ between two strings of equal length:

the number of positions where they differ

Task: find approximate occurrences of P in T with Hamming distance $\leq k$

$\{i \mid d_H(P, T[i..i+m-1]) \leq k\}$

Trivial algorithm $O(nm)$

Algorithm with suffix trees and LCA: $O(nk)$ [Landau, Vishkin 1986]

More complex version: $O(n\sqrt{k \log k})$ [Amir, Lewenstein, Porat 2000]

Using fast Fourier transform: $O(n\sigma \log m)$ [Fischer and Paterson 1974]

Edit distance, Levenshtein distance (editačná vzdialenosť)

Edit operations: $(u, v \in \Sigma^*, a, b \in \Sigma)$

– insertion (inzercia) $uv \rightarrow uav$

– deletion (delécia) $uav \rightarrow uv$

– substitution (substitúcia) $uav \rightarrow ubv$

Edit distance $d_E(S, T) =$

shortest sequence of edit operations that changes S to T

Example:

$S =$ema ma mamu, $T =$mama sa ma, $d_E(S, T) = 5$

ema␣ma␣mamu (delete e) ma␣ma␣mamu (delete space)

mama␣mamu (substitute m->s) mama␣samu (insert space)

mama␣sa␣mu (substitute u-a) mama␣sa␣ma

Edit distance as an alignment

Sequence alignment (zarovnanie):

insert gaps ($-$) to S and T to get matrix with 2 rows

— column with a gap (insertion or deletion): cost 1

— column with two different symbols (substitution): cost 1

— column with equal symbols: cost 0

Example:

```
ema␣ma␣ma-mu
-ma-ma␣sa␣ma
100100010101
```

Problem: compute $d_E(S, T)$ for two input strings S and T

(note $d_H(S, T)$ trivially in $O(n)$ time)

Dynamic programming for $d_E(S, T)$

Let $m = |S|$, $n = |T|$, indexing from 1: $S[1..m]$, $T[1..n]$

Let $A[i, j] = d_E(S[1..i], T[1..j])$

Compute $A[i, j]$ for $0 \leq i \leq m$, $0 \leq j \leq n$

Example:

`12345678901`

`ema␣ma␣mamu`

`mama␣sa␣ma`

$A[3, 4] = 2$

Dynamic programming for $d_E(S, T)$

Let $m = |S|$, $n = |T|$, indexing from 1: $S[1..m]$, $T[1..n]$

Let $A[i, j] = d_E(S[1..i], T[1..j])$

Compute $A[i, j]$ for $0 \leq i \leq m$, $0 \leq j \leq n$:

$$A[0, i] = A[i, 0] = i$$

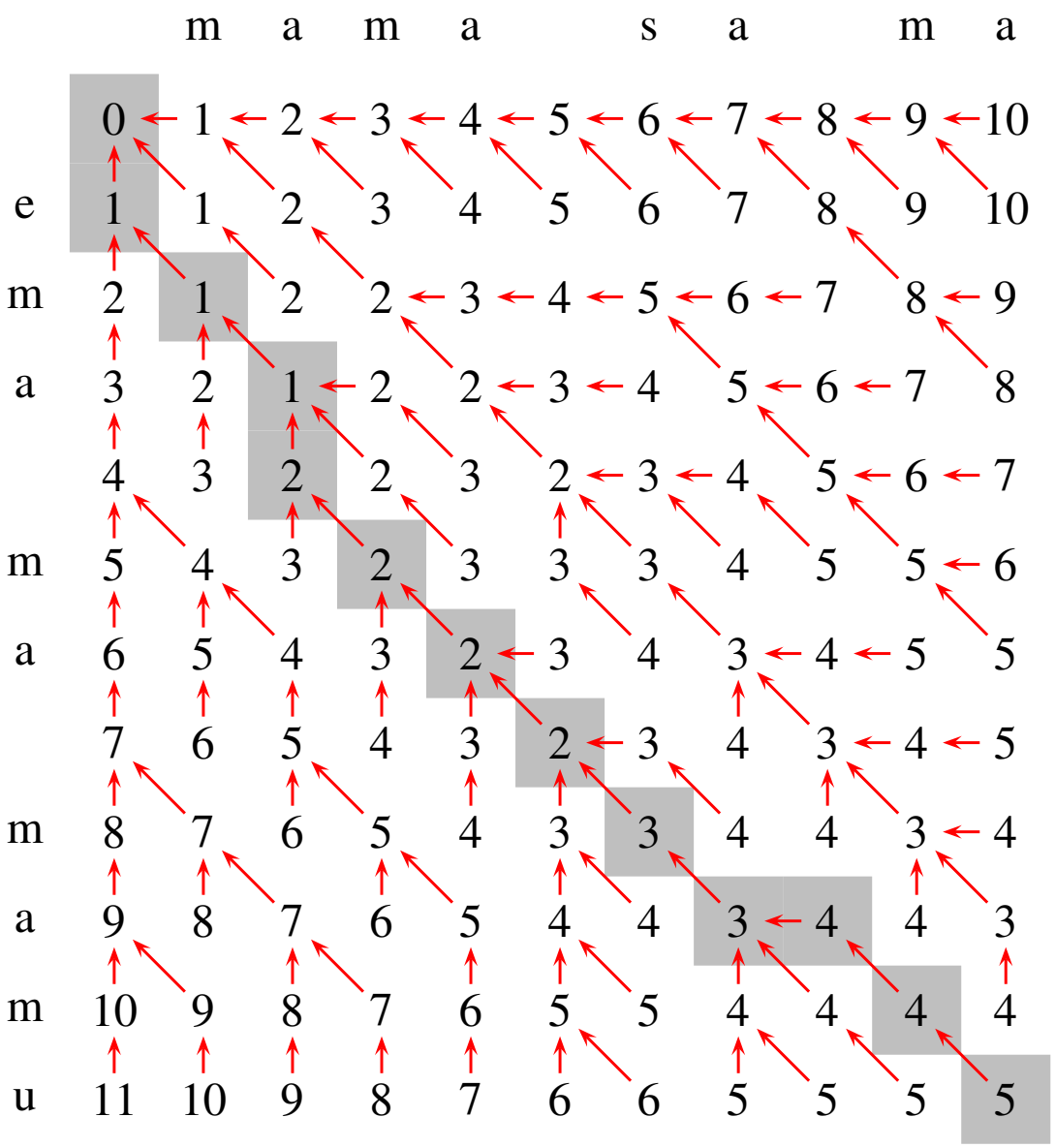
$$A[i, j] = \min \begin{cases} A[i-1, j-1] + c(S[i], T[j]) \\ A[i-1, j] + 1 \\ A[i, j-1] + 1 \end{cases}$$

$$c(a, b) = \begin{cases} 0 & \text{if } a = b \\ 1 & \text{if } a \neq b \end{cases}$$

		m	a	m	a		s	a		m	a
	0	1	2	3	4	5	6	7	8	9	10
e	1	1	2	3	4	5	6	7	8	9	10
m	2	1	2	2	3	4	5	6	7	8	9
a	3	2	1	2	2	3	4	5	6	7	8
	4	3	2	2	3	2	3	4	5	6	7
m	5	4	3	2	3	3	3	4	5		
a											

$$A[i,j] = \min \begin{cases} A[i-1,j-1] \\ \quad +c(S[i],T[j]) \\ A[i-1,j]+1 \\ A[i,j-1]+1 \end{cases}$$

m											
a											
m											
u											



Alignment:

```
ema␣ma␣ma-mu
-ma-ma␣sa␣ma
```

Generalized edit distance

(zovšeobecnená editačná vzdialenosť)

Table of weights $w(a, b)$ for $a, b \in \Sigma \cup \{-\}$

For $a, b \in \Sigma$:

$w(a, -)$ cost of deletion, $w(-, b)$ cost of insertion

$w(a, b)$ cost of substitution from a to b, or cost of identity if $a = b$

Dynamic programming to find the alignment with lowest cost

– for some strange weights not the lowest sequence of operations

– not always a distance function

$$A[i, j] = \min \begin{cases} A[i-1, j-1] + w(S[i], T[j]) \\ A[i-1, j] + w(S[i], -) \\ A[i, j-1] + w(-, T[j]) \end{cases}$$

Longest common subsequence lcs

Najdlhšia spoločná podpostupnosť

Def: subsequence of a sequence $a_1, \ldots, a_n$ is sequence $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$ such that $1 \leq i_1 < i_2 < \cdots < i_k \leq n$

$lcs(S, T) =$ length of the longest sequence which is a common subsequence of both S and T

Example:

$S =$ema ma mamu, $T =$mama sa ma, $lcs(S, T) = 7$

Also alignment (disallow substitutions):

```
ema␣ma␣---mamu
-ma-ma␣sa␣ma--
.**.***.....**..
```

How to find using DP for generalized edit distance?

Program diff

Compare two files line by line, e.g. two versions of a source code.

Find (approximation of) lcs(S,T) where symbols are lines of the files.

```
1522a1540
>     my $last_line = undef;
1525,1526c1543,1544
<     foreach my $gtf_line (@$transcript) {
<       # printf STDERR Dumper($gtf_line);
---
>     for(my $line_num = 0; $line_num<@$transcript; $line_num++) {
>       my $gtf_line = $transcript->[$line_num];
```

Running time of dynamic programming

$O(nm)$ on strings of lengths n and m

How long does it takes?

(straightforward implementation, random sequences of length n,

ordinary desktop couple of years ago)

n	time
100	0.0008s
1,000	0.08s
10,000	8s
100,000	13 minutes (*)
1,000,000	22 hours (*)
10,000,000	3 months (*)
100,000,000	25 years (*)

Improvements of dynamic programming

Hunt-Szymanski algorithm for LCS: $O(m + n + r \log r)$

where $0 \leq r \leq mn$, $r = |\{(i, j) : S[i] = T[j]\}|$

Uses $O(r \log r)$ algorithm to find the longest increasing subsequence

Four Russians technique: $O(mn/\log m)$ for a constant σ

(precompute small squares, save time)

Hirschberg's algorithm: $O(mn)$ time but $O(m + n)$ memory

(compute where the paths crosses middle of the matrix,

divide and conquer)

Ukkonen's algorithm: $O(md)$ time where $d = d_E(S, T)$

Guess d, verify in a band of diagonals

Theory of edit distance computation

Compute d_E of two strings of length n

Exact algorithm: $O(n^2/\log n)$ time

Approximation algorithm: $(\log n)^{O(1/\epsilon)}$ approximation in $n^{1+\epsilon}$ time [Andoni, Krauthgamer, Onak 2010]

Conditional lower bound: If edit distance can be computed in $O(n^{2-\delta})$ for some constant $\delta > 0$, then SAT for formula with N variables and M clauses can be solved in $M^{O(1)}2^{(1-\epsilon)N}$ for a constant $\epsilon > 0$, which would violate Strong Exponential Time Hypothesis (SETH). [Backurs, Indyk 2015]

Approximate string matching (približné výskyty vzorky)

Find substrings of T s.t. $d_E(T[i..j], P) \leq k$

How to modify dynamic programming for edit distance?

$$A[0, i] = A[i, 0] = i$$

$$A[i, j] = \min \begin{cases} A[i-1, j-1] + c(S[i], T[j]) \\ A[i-1, j] + 1 \\ A[i, j-1] + 1 \end{cases}$$

Simple dynamic programming $O(mn)$

Subproblem: $A[i,j] = \min_\ell d_E(P[0..i], T[\ell..j])$

Recurrence:

$A[0,j] = 0$

$A[i,0] = i$

$$A[i,j] = \min \begin{cases} A[i-1,j-1] + c(P[i], T[j]) \\ A[i-1,j] + 1 \\ A[i,j-1] + 1 \end{cases}$$

Print all j s.t. $A[m,j] \leq k$ (ends of occurrences)

We need only 2 columns from A

What if we want the best start (with smallest d_E) for each end?

Simple improvement, faster on average

Compute correct values only for active cells

We call $A[i, j]$ active if $A[i, j] \leq k$

Let $L[j]$ be largest i s.t. $A[i, j]$ is active

Lemma: Adjacent values in a row or column of A differ by at most 1

Lemma: $L[j] \leq L[j-1] + 1$

Simple improvement, faster on average

```
1   Compute A[*,0];
2   L = k;
3   for(j=1; j<=n; j++) {
4     L2 = 0;
5     for(i=0; i<=L+1; i++) {
6       compute A[i,j];   // if unknown A[i,j-1], assume k+1
7       if(A[i,j]<=k) L2 = i;
8     }
9     L = L2;
10  }
```

Average-case running time $O(kn)$, worst-case $O(mn)$

Landau-Vishkin 1986 $O(kn)$

Diagonal number d : all values $A[i, j]$ where $j - i = d$

$L[d, e]$: maximum row i on diagonal d such that $A[i, i + d] \leq e$

```
1  // L[d,−1] = −1 (+ plus other boundary cases)
2  for(e=0; e<=k; e++) {
3     for(d= −e; d<=n; d++) {
4        i = min(m, max(L[d, e−1]+1, L[d−1,e−1], L[d+1,e−1]+1))
5        while(i<m && i+d<n && P[i]==T[i+d]) { i++; }
6        L[d,e] = i;
7        if(L[d,e]==m) { print occurrence ending at d+m }
8     }
9  }
```

Landau-Vishkin 1986 $O(kn)$

```
1  for(e=0; e<=k; e++) {
2    for(d=-e; d<=n; d++) {
3      i = min(m, max(L[d, e-1]+1, L[d-1,e-1], L[d+1,e-1]+1))
4      while(i<m && i+d<n && P[i]==T[i+d]) { i++; } // (*)
5      L[d,e] = i;
6      if(L[d,e]==m) { print occurrence ending at d+m }
7    }
8  }
```

$L[d, e]$: maximum row i on diagonal d such that $A[i, i + d] \leq e$

Clearly algorithm computes lower bound of $L[d, e]$

But correctness needs more proof

Line (*) replaced with LCA on suffix tree for P and T in $O(1)$

$i+=$ longest common prefix of $P[i..m]$ and $T[i + d..n]$

Baeza-Yates, Perleberg 1992

Divide P to $k+1$ substrings of size $r = \lfloor \frac{m}{k+1} \rfloor$ (last one possibly longer). Denote the set of these substrings $\mathcal{P}$.

Lemma: Let T' be a substring of T. If $d_E(T', P) \leq k$, at least one of the strings in $\mathcal{P}$ is a substring of T'.

Find occurrences of strings in $\mathcal{P}$ in T, search around each.

Average-case running time $O(n + n(k+1)m^2/\sigma^r)$

Filtering in general: Similar exact or heuristic techniques widely used