UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Renato Urajnar

Analiza algoritmov za iskanje podnizov

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Borut Robič

Ljubljana, 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00129/2013
Datum: 11.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **RENATO URAJNAR**

Naslov: **ANALIZA ALGORITMOV ZA ISKANJE PODNIZOV**
EXACT STRING MATCHING ALGORITHMS

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

Analizirajte znane algoritme za iskanje podnizov. Opišite njihov pomen in uporabnost. Pri vsakem algoritmu opišite glavno idejo iskanja. Izbrane algoritme implementirajte v Javi in jih z različnimi vzorci preizkusite na daljšem besedilu. Rezultate poskusov ustrezno prikažite, komentirajte in ocenite učinkovitost implementiranih algoritmov.

Mentor:

prof. dr. Borut Robič

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani Renato Urajnar,

z vpisno številko 63040170,

sem avtor diplomskega dela z naslovom:

Analiza algoritmov za iskanje podnizov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  prof. dr. Boruta Robiča
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 25.1.2013 Podpis avtorja:

Zahvala

Zahvaljujem se svoji mami za podporo in potrpežljivost tekom študija. Velika zahvala pa gre tudi prijateljem, ki so mi pri študiju nesebično pomagali.

Kazalo

Seznam uporabljenih kratic in simbolov

KMP - Knuth-Morris-Pratt
RK - Rabin-Karp
AC - Aho-Corasick
BNDM - Backward Nondeterministic Dawg Matching
GG - Galil-Giancarlo
RC - Reverse Colussi
BM - Boyer-Moore
DKA - Deterministični končni avtomat
MS - Maximal suffix
DAWG - Directed acyclic word graph
MP - Morris-Pratt

Povzetek

Iskanje podnizov je eden vidnejših problemov na področju računalništva. V diplomski nalogi smo opisali in med seboj primerjali do sedaj znane pristope k reševanju te problematike. Algoritme smo razvrstili glede na njihov način delovanja in ocenili časovno zahtevnost posameznega algoritma. Skušali smo ugotoviti prednosti in glavno idejo algoritma. Glede na to, da je algoritmov zelo veliko in so med seboj sorodni, smo za implementacijo v Javi izbrali nekaj osnovnih algoritmov, ki iščejo točno določen vzorec. Preizkusili smo jih na praktičnem primeru in med njimi izbrali najboljšega.

Ključne besede:

podniz, iskanje nizov, substitucija niza, ujemanje nizov, ujemanje, niz, zamik, funkcija napake, vzorec, ujemanje vzorcev

Abstract

Substring searching is one of the most prominent problems in computer science. We described and compared with each other by now well-known approaches to the solution. Algorithms were classified according to their properties, such as number of patterns and the direction in which the comparisons are performed. We examined the main idea, advantages and time complexity for each algorithm. Some basic algorithms, which are searching for exactly one pattern, were implemented in Java and tested on a practical example. We picked the fastest one among them.

Key words:

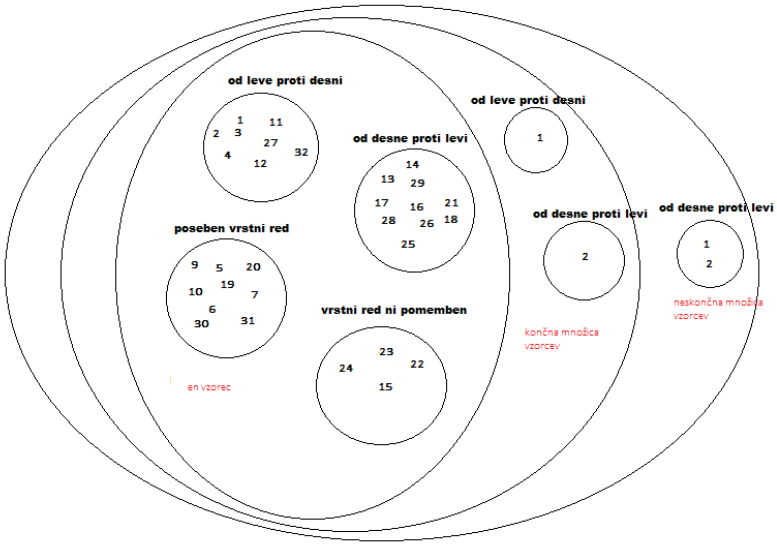
substring, string searching, string substitution, string matching, shift, failure function, pattern, pattern matching

Poglavje 1

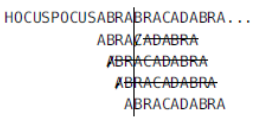
Uvod

Podatke shranjujemo in izmenjujemo na številne načine, vendar najvidnejša oblika za izmenjavo informacij ostaja besedilo. Količina teh podatkov se hitro povečuje, zato je zelo pomembno, da imamo učinkovite algoritme za preiskovanje. V računalništvu poznamo veliko algoritmov, ki operirajo nad nizi. Pomembnejši razred predstavljajo algoritmi, ki iščejo eno ali več pojavitev vzorca v daljšem nizu. V današnjih časih so to klasične komponente v številnih implementacijah programske opreme in imajo pomembno vlogo v teoretičnem računalništvu in ostalih znanstvenih področjih.
Iskanje podniza poteka tako, da nad neko abecedo zgradimo vzorec in niz, ki ga bomo preiskovali, in nato z drsečim oknom iščemo pojavitve vzorca v nizu. Glede na način, kako algoritmi preiskujejo besedilo, bi jih lahko razvrstili v več skupin. Razdelimo jih lahko na dva načina, in sicer glede na število vzorcev in vrstni red primerjanja elementov vzorca in niza. Glede na število vzorcev jih razdelimo v tri množice: algoritmi, ki iščejo točno določen vzorec, algoritmi, kjer imamo končno število vzorcev in algoritmi z neskončnim številom vzorcev. Glede na smer delovanja pa v štiri: od leve proti desni, od desne proti levi, posebni vrstni red in algoritmi, kjer vrstni red primerjav ni pomemben. Na sliki 1.1 so algoritmi (ostevilčeni kot v drugem poglavju) dodeljeni posamezni množici. Večina teh algoritmov v fazi predprocesiranja obdela vnaprej podan vzorec, nato se začne dejanska primerjava. Ujemanje je uspešno, če se vzorec in del niza povsem ujemata. To pomeni, da če označimo vzorec $P[1...m]$ in tekst $T[1...n]$ mora veljati $P[1...m] = T[i + 1...i + m]$. Cilj vsakega algoritma je porabiti čim manj časa za primerjave in preiskati celotno besedilo. Obstaja veliko algoritmov, ki so izboljšava prvotnega naivnega algoritma. Problem naivnega algoritma je, da izvaja redundančna primerjanja. Na sliki 1.2 je primer takega primerjanja. V prečrtanih vrsticah (tretja in četrta vrstica) se zgodijo

redundančne primerjave. V praksi se izkaže, da algoritmi, ki izvajajo primerjave od desne proti levi, delujejo najbolje, čeprav teoretične meje postavijo algoritmi s posebnim vrstnim redom.



Slika 1.1: Razdelitev algoritmov glede na njihovo delovanje. Algoritmi, ki lahko iščejo več vzorcev hkrati, bi lahko iskali tudi samo en sam vzorec, vendar za to niso optimizirani.



Slika 1.2: Primer redundančnih primerjav.

Poglavje 2

Algoritmi

2.1 En vzorec

2.1.1 Brute-force

Algoritem po nizu T dolžine n od začetka pomika drseče okno (vzorec P dolžine m) po en znak. Na i-tem koraku preveri, ali velja $P[0...m-1] = T[i...i+m-1]$, kar pomeni, da je našel pojavitev vzorca v nizu. Sicer pomakne vzorec za eno mesto v desno in postopek ponovi. Preiskati mora $n - m$ znakov v nizu, zato ima preiskovalna faza časovno kompleksnost reda $O(mn)$.

Lastnosti:

- ni predprocesiranja,
- poraba dodatnega prostora je konstantna,
- v splošnem vrstni red primerjav ni pomemben,
- pomik vedno za eno mesto,
- preiskovalna faza je reda $O(mn)$,
- $2n$ pričakovanih primerjav.

2.1.2 Izboljšan Brute-force

Algoritem uvede predprocesiranje, saj že pred začetkom iskanja primerja prvi in drugi znak vzorca, in na podlagi te informacije skuša drseče okno pomakniti za 2 mesti v desno, če je le to mogoče. Primerjave izvede od drugega do zadnjega znaka, na koncu pa primerja še prvi znak vzorca. Pri vsaki postavitvi drsečega okna preveri, ali velja $P[0] = P[1] \wedge P[1] \neq T[j+1] \vee P[0] \neq P[1] \wedge P[1] = P[j+1]$, in nato zamakne okno za 2 mesti, sicer pa za 1 mesto.

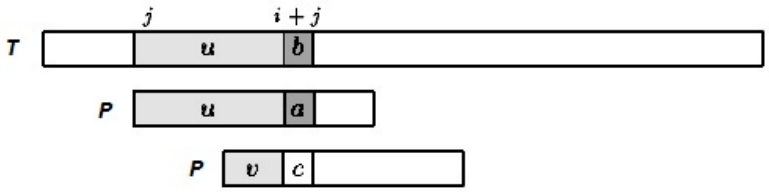
Algoritem ima v najslabšem primeru še vedno kvadratično časovno zahtevnost.

Lastnosti:

- predprocesiranje v konstantnem času in prostoru,
- faza iskanja ima časovno zahtevnost $O(mn)$.

2.1.3 Morris-Pratt

Motivacija za razvoj algoritma sledi iz analize načina *brute-force*. Slednji namreč zavrže celotno informacijo o besedilu, pridobljeno s prejšnimi primerjavami. Posledica tega je, da se pri premiku vzorca vedno pomaknemo le eno mesto v desno. Algoritem Morris-Pratt pa skuša s pomočjo informacije o že skeniranem tekstu odpraviti redundančne primerjave in pomakniti vzorec kar se da v desno. Recimo, da je algoritem med svojim delovanjem že našel ujemanje začetnega dela vzorca in naletel na neujemanje pri $P[i]$ in $T[i+j]$, kjer je $0 < i < m$. Potem velja $P[0...i-1] = T[j...i+j-1] = u$ in $a = P[i] \neq T[i+j] = b$ (glej sliko 2.1). Pri zamikanju vzorca moramo biti pozorni, ali se predpona vzorca v ujema s končnico v u. Najdaljša predpona v se imenuje rob u-ja (pojavi se na obeh koncih u-ja). Naj bo $mpNext[i]$ dolžina najdaljšega roba $P[0...i-1]$ za $0 < i <= m$. Robove izračunamo v fazi predprocesiranja, tako da algoritem izvedemo na vzorcu. Po zamiku se primerjave nadaljujejo od $c = P[mpNext[i]]$ in $T[i+j] = b$, ne da bi izgubili kakšno pojavitev P-ja v T [1].



Slika 2.1: Zamik vzorca z algoritmom Morris-Pratt.

Lastnosti:

- primerjave izvaja od leve proti desni,
- predprocesiranje ima časovno in prostorsko zahtevnost $O(m)$,
- faza iskanja je reda $O(n+m)$,
- največje število primerjav za en znak je omejeno z m.

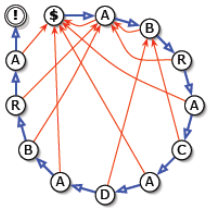
2.1.4 Knuth-Morris-Pratt

Po podrobni analizi algoritma Morris-Pratt pridemo do spoznanja, da lahko dolžino zamikov vzorca še izboljšamo. Oglejmo si situacijo s slike 2.1. Predpona v se ujema s končnico u-ja. Če se želimo v naslednjem koraku izogniti neujemanju, moramo zagotoviti, da se znak, ki sledi predponi v v vzorcu, razlikuje od a. Najdaljša taka predpona v se imenuje označen rob (*tagged border*) u-ja. Ideja algoritma Knuth-Morris-Pratt v primerjavi z algoritmom Morris-Pratt je torej izboljšava funkcije napake, ki računa najdaljše robove. $kmpNext[i]$ je dolžina najdaljšega roba $P[0...i-1]$, ki mu sledi znak c, ki se razlikuje od $P[i]$ in -1, če tak rob ne obstaja za $0 < i <= m$. Po zamiku se primerjave nadaljujejo pri znakih $P[kmpNext[i]]$ in $T[i+j]$ [2].
Lastnosti:

- primerjave izvaja od leve proti desni,
- predprocesiranje ima časovno in prostorsko zahtevnost $O(m)$,
- faza iskanja je reda $O(n+m)$,
- največje število primerjav za en znak je omejeno z $log_{\Phi}(m)$, kjer je $\Phi = \frac{1+\sqrt{5}}{2}$ zlato razmerje.

2.1.4.1 Računanje funkcije napake

Namen te funkcije je v fazi predprocesiranja izračunati indeks znaka v vzorcu, pri katerem se bodo primerjave nadaljevale, če je v trenutnem koraku prišlo do neujemanja (vrstica 29 v implementaciji). Indeks v tekstu ostane enak, torej se bodo primerjave nadaljevale na istem znaku v tekstu in drugem znaku v vzorcu. To imenujemo zamik vzorca. Če poznamo zamik pri vsaki predponi vzorca, dobimo vse možne zamike pri tem vzorcu. Funkcija je odvisna samo od vzorca, zato jo lahko izračunamo v fazi predprocesiranja. Za lažjo predstavo si oglejmo primer na besedi ABRACADABRA, realiziran s pomočjo končnega avtomata (slika 2.2).



Slika 2.2: Puščice nakazujejo spremembo indeksa v vzorcu.

2.1.4.2 Implementacija v Javi

```
1  /* Zaporedoma izpise vse indekse, pri katerih se vzorec nahaja v tekstu */
2  /* Vzorec je podan v obliki P$ */
3
4  public static void kmp(String T, String P) {
5          int i, j;
6          /* Predprocesiranje */
7          computeFailure(P);
8          /* Iskanje */
9          i = j = 0;
10         while (j < T.length()) {
11                 while (i > -1 && P.charAt(i) != T.charAt(j))
12                         i = failure[i];
13                 i++;
14                 j++;
15                 if (i >= P.length() - 1) {
16                         System.out.println(j - i);
17                         i = failure[i];
18                 }
19         }
20 }
21
22 /* Izracunamo funkcijo napake tako, da vzorec primerjamo s samim seboj. */
23
24 public static void computeFailure(String P) {
25         int i, j;
26         j = 0;
27         i = failure[0] = -1;
28         while (j < P.length() - 1) {
29                 while (i > -1 && P.charAt(j) != P.charAt(i))
30                         i = failure[i];
31                 i++;
32                 j++;
33                 if (P.charAt(j) == P.charAt(i))
34                         failure[j] = failure[i];
35                 else
36                         failure[j] = i;
37         }
38 }
```

2.1.5 Colussi

Colussijeva zamisel je osnovana na algoritmu KMP z nekaj spremembami. Indekse vzorca razdelimo v dve množici. Najprej preverjamo samo pozicije *noholes* od leve proti desni (pozicije, pri katerih ima funkcija $kmpNext$, izposojena od algoritma KMP, strogo večjo vrednost od -1) in če pride do neujemanja, še ostale pozicije (*holes*) od desne proti levi. Ta strategija nam prinese dve prednosti, ki sta razvidni in opisani na slikah 2.3 in 2.4. V primeru neujemanja zamaknemo okno s funkcijo $shift$, indeks, kjer se bodo primerjave nadaljevale, pa izračunamo s funkcijo $next$. Funkciji, izračunani v fazi predprocesiranja,

sta definirani tako:

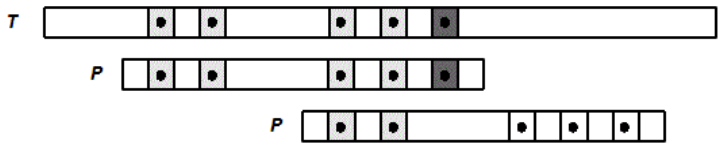
$shift(i) = kmin(h(i)) \wedge next(i) = nhd0(h(i) - kmin(h(i))$ za $i < nd$,
$shift(i) = rmin(h(i)) \wedge next(i) = nhd0(m - rmin(h(i))$ za $nd \leq i < m$,
$shift(m) = rmin(0) \wedge next(m) = nhd0(m - rmin(h(m-1))$ za $i < nd$,

kjer morajo veljati naslednje definicije:

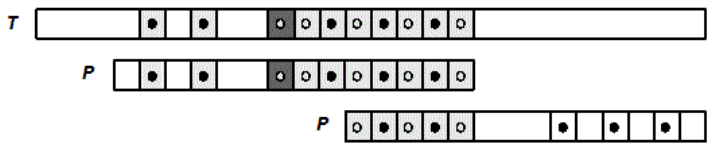
- vrednost $ndh0(i)$ je število *nohole* pozicij strogo manjše od i,
- za $0 \leq i \leq m-1 : kmin(i) = d > 0$, če $P[0...i-1-d] = P[d...i-1]$ in $P[i-d] \neq P[i]$, sicer 0. Če $kmin(i) \neq 0$, potem je i *nohole*,
- $nd + 1$ je število *nohole* pozicij v P,
- če je i *hole* , potem je $rmin(i)$ najmanjša perioda P-ja večja od i,
- tabela h po vrsti vsebuje $nd + 1$ *nohole* pozicij v naraščajočem vrstnem redu in potem $m - nd - 1$ v padajočem:

  - $0 \leq i \leq nd, h(i)$ je *nohole* in $h(i) < h(i+1)$ za $0 \leq i < nd$,
  - $nd < i < m, h(i)$ je *hole* in $h(i) > h(i+1)$ za $nd \leq i < m-1$.

Za lažje razumevanje predprocesiranja naj si bralec pomaga še s sliko 2.5 in literaturo [10].



Slika 2.3: Neujemanje pri *nohole* poziciji (*noholes* so črni krogci). Po zamiku ni treba znova primerjati prvih dveh *nohole* pozicij, ker smo to naredili že v prejšnjem koraku.



Slika 2.4: Ujemanje vseh *nohole* pozicij. Neujemanje pri *hole* (*holes* so beli krogci). Po zamiku ni treba znova primerjati že ujemajoče se predpone.

i	0	1	2	3	4	5	6	7	8
$P[i]$	G	C	A	G	A	G	A	G	
$kmpNext[i]$	-1	0	0	-1	1	-1	1	-1	1
$kmin[i]$	0	1	2	0	3	0	5	0	
$h[i]$	1	2	4	6	7	5	3	0	
$next[i]$	0	0	0	0	0	0	0	0	0
$shift[i]$	1	2	3	5	8	7	7	7	7
$hmax[i]$	0	1	2	4	4	6	6	8	8
$rmin[i]$	7	0	0	7	0	7	0	8	
$ndh0[i]$	0	0	1	2	2	3	3	4	

$nd = 3$

Slika 2.5: Vrednosti posameznih funkcij, izračunanih v fazi predprocesiranja, na konkretnem primeru.

Lastnosti:

- faza predprocesiranja v $O(m)$ času in prostoru,
- faza iskanja $O(n)$,
- v najslabšem primeru $\frac{3n}{2}$ znakovnih primerjav.

2.1.5.1 Implementacija v Javi

```
1  public static int preColussi(String P, int m, int h[], int next[], int shift[]) {
2
3          int i, k, nd, q, s;
4          int r = 0;
5          int[] kmin = new int[m];
6          int[] nhd0 = new int[m];
7          int[] rmin = new int[m];
8          int[] hmax = new int[m + 1];
9
10         /* racunanje hmax, tabela je dolzine m+1 zato pri vzorcu dodamo se en znak,  da ne presezemo
                polja, stevilo m pa pove dejanski vzorec */
11         i = k = 1;
12         do {
13                 while (P.charAt(i) == P.charAt(i - k))
14                         i++;
15                 hmax[k] = i;
16                 q = k + 1;
17                 while (hmax[q - k] + k < i) {
18                         hmax[q] = hmax[q - k] + k;
19                         q++;
20                 }
21                 k = q;
22                 if (k == i + 1)
23                         i = k;
24
25                 } while (k <= m);
26
27         /* racunanje kmin s pomocjo hmax tako da velja:*/
```

```
28          /* P[k ... hmax[k]-1] = P[0 ... hmax[k]-k-1] in P[hmax[k]] != P[hmax[k]-k] */
29          for (i = m; i >= 1; --i)
30                  if (hmax[i] < m)
31                          kmin[hmax[i]] = i;
32
33          /* racunanje rmin */
34          for (i = m - 1; i >= 0; --i) {
35                  if (hmax[i + 1] == m)
36                          r = i + 1;
37                  if (kmin[i] == 0)
38                          rmin[i] = r;
39                  else
40                          rmin[i] = 0;
41          }
42
43          /* racunanje h */
44          s = -1;
45          r = m;
46          for (i = 0; i < m; ++i)
47                  if (kmin[i] == 0)
48                          h[--r] = i;
49                  else
50                          h[++s] = i;
51          nd = s;
52
53          /* racunanje zamika (shift) */
54          for (i = 0; i <= nd; ++i)
55                  shift[i] = kmin[h[i]];
56          for (i = nd + 1; i < m; ++i)
57                  shift[i] = rmin[h[i]];
58          shift[m] = rmin[0];
59
60                  /* racunanje nhd0 */
61          s = 0;
62          for (i = 0; i < m; ++i) {
63                  nhd0[i] = s;
64                  if (kmin[i] > 0)
65                          ++s;
66          }
67
68          /* racunanje next */
69          for (i = 0; i <= nd; ++i)
70                  next[i] = nhd0[h[i] - kmin[h[i]]];
71          for (i = nd + 1; i < m; ++i)
72                  next[i] = nhd0[m - rmin[h[i]]];
73          next[m] = nhd0[m - rmin[h[m - 1]]];
74
75                  return (nd);
76
77  }
78
79  public static void colussi(String P, int m, String T, int n) {
80          int i, j, last, nd;
81          int[] h = new int[m];
82          int[] next = new int[m + 1];
83          int[] shift = new int[m + 1];
84
85                  /* Predprocesiranje */
86          nd = preColussi(P, m, h, next, shift);
```

```
87
88          /* Iskanje */
89          i = j = 0;
90          last = -1;
91          while (j <= n - m) {
92                  /* preverjanje holes in noholes, zgoraj glej opis tabele h */
93                  while (i < m && last < j + h[i]
94                                  && P.charAt(h[i]) == T.charAt(j + h[i]))
95                          i++;
96                  if (i >= m || last >= j + h[i]) {
97                          System.out.println(j);
98                          i = m;
99                  }
100                 if (i > nd)
101                         last = j + m - 1;
102                 j += shift[i];
103                 i = next[i];
104         }
105 }
```

2.1.6 Galil-Giancarlo

Algoritem izhaja iz algoritma Colussi, pri katerem izboljša le fazo iskanja (predprocesiranje je identično). Uporaben je pri vzorcih, ki niso niz enega samega znaka, torej: $P \neq c^m$, ker $c \in \Sigma$. Naj bo ℓ zadnji indeks v vzorcu, da velja: $0 \leq i \leq \ell$, kjer $P[0] = P[i]$ in $P[0] \neq P[\ell + 1]$. Recimo, da je algoritem med svojim delovanjem v prejšnjem koraku uspešno poiskal vse *nohole* pozicije (glej poglavje Colussi) in neko končnico vzorca. To pomeni, da se bo po ustreznem zamiku predpona vzorca ujemala z delom teksta. Pozicija okna v tekstu je $T[j...j+m-1]$ in del teksta $T[j...last]$ se ujema s $P[0...last-j]$. V naslednjem koraku se primerjave nadaljujejo pri indeksu $T[last + 1]$, dokler ne dosežemo konca teksta oziroma najdemo znak $P[0] \neq T[j + k]$. V drugem primeru sta možna dva izida (glej vrstico 35 v implementaciji):

- če $P[\ell + 1] \neq T[j + k]$ ali premalo najdenih znakov $P[0]$, torej $k \leq \ell$, potem okno pozicioniramo na $T[k + 1...k + m]$, skeniranje teksta pa se nadaljuje pri prvi *nohole* poziciji (kot pri Colussi);

- če $P[\ell+1] = T[j+k]$ in dovolj najdenih znakov $P[0]$, torej $k > \ell$, potem okno pozicioniramo na $T[k - \ell - 1...k - \ell + m - 2]$, skeniranje teksta se nadaljuje pri drugi *nohole* poziciji ($P[\ell + 1]$ je prva).

Za več informacij glej literaturo [14].

Lastnosti:

- predprocesiranje (čas, prostor) $O(m)$,
- faza iskanja v $O(n)$ času,
- v najslabšem primeru izvede $\frac{4n}{3}$ primerjav.

2.1.6.1 Implementacija v Javi

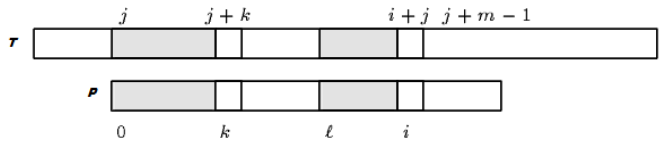
```
1  public static void GG(String P, int m, String T, int n) {
2          int i, j, k, ell, last, nd;
3          int[] h = new int[m];
4          int[] next = new int[m + 1];
5          int[] shift = new int[m + 1];
6          char heavy;
7
8          for (ell = 0; P.charAt(ell) == P.charAt(ell + 1) && ell < m - 1; ell++)
9                  ;
10         if (ell == m - 1)
11                 /* Iskanje nizov sestavljenih iz enega samega znaka */
12                 for (j = ell = 0; j < n; ++j)
13                         if (P.charAt(0) == T.charAt(j)) {
14                                 ++ell;
15                                 if (ell >= m)
16                                         System.out.println(j - m + 1);
17                         } else
18                                 ell = 0;
19         else {
20                 /* Predprocesiranje je identicno Colussi algoritmu */
21                 nd = preColussi(P, m, h, next, shift);
22
23                 /* Iskanje */
24                 i = j = heavy = 0;
25                 last = -1;
26                 while (j <= n - m) {
27                         /* v slednjem if pogoju je realizirana izboljsava */
28                         if (heavy == 0 && i == 0) {
29                                 k = last - j + 1;
30                                 while (P.charAt(0) == T.charAt(j + k)) {
31                                         k++;
32                                         if (j + k == T.length() - 1)
33                                                 break;
34                                 }
35                                 if (k <= ell || P.charAt(ell + 1) != T.charAt(j + k)) {
36                                         i = 0;
37                                         j += (k + 1);
38                                         last = j - 1;
39                                 } else {
40                                         i = 1;
41                                         last = j + k;
42                                         j = last - (ell + 1);
43                                 }
44                                 heavy = 0;
45                         } else {
46                                 while (i < m && last < j + h[i]
```

```
47                                          && P.charAt(h[i]) == T.charAt(j + h[i]))
48                                  ++i;
49                          if (i >= m || last >= j + h[i]) {
50                                  System.out.println(j);
51                                  i = m;
52                          }
53                          if (i > nd)
54                                  last = j + m - 1;
55                          j += shift[i];
56                          i = next[i];
57                  }
58                  if (j > last)
59                          heavy = 0;
60                  else
61                          heavy = 1;
62          }
63      }
64  }
```

2.1.7 Apostolico-Crochemore

Ideja algoritma je pregledovati znake v posebnem vrstnem redu in uporabiti funkcijo zamikov $kmpNext$. Naj bo $\ell = 0$, kadar je vzorec sestavljen iz enega samega znaka, sicer pa je vrednost enaka poziciji prvega znaka, ki se razlikuje od $P[0]$. V vsakem koraku se primerjave izvedejo v naslednjem vrstnem redu: $\ell, \ell+1, ..., m-2, m-1, 0, 1, ..., \ell-1$. Sestavimo trojico (i, j, k), katere začetna vrednost je $(\ell, 0, 0)$ (slika 2.6). Okno je pozicionirano na odseku $P[j...j+m-1]$ in velja:

- $0 \leq k \leq \ell$ in $P[0...k-1] = T[j...j+k-1]$,
- $\ell \leq i < m$ in $P[\ell...i-1] = T[j+\ell...i+j-1]$.



Slika 2.6: Na vsakem koraku opazujemo trojico *(i, j, k)*.

Izračun naslednje vrednosti trojice (i, j, k) je odvisen od i:

$i = \ell$:

- če $P[i] = T[i+j]$, potem je naslednja trojica $(i+1, j, k)$,
- če $P[i] \neq T[i+j]$, potem je naslednja trojica $(\ell, j+1, max\{0, k-1\})$.

$\ell < i < m$:

- če $P[i] = T[i + j]$, potem je naslednja trojica $(i + 1, j, k)$,
- če $P[i] \neq T[i+j]$, dobimo dva scenarija v odvisnosti od funkcije $kmpNext[i]$:

  1. $kmpNext[i] \leq \ell$: potem je naslednja trojica $(\ell, i + j - kmpNext[i], max\{0, kmpNext[i]\})$;
  2. $kmpNext[i] > \ell$: potem je naslednja trojica $(kmpNext[i], i + j - kmpNext[i], \ell)$.

$i = m$:

- če $k < \ell$ in $P[k] = T[j + k]$, potem je nasledna trojica $(i, j, k + 1)$.

Sicer $k < \ell$ in $P[k] \neq T[j+]$, oziroma je $k = \ell$. Če $k = \ell$, smo našli vzorec. V obeh primerih je nasledna trojica izračunana enako kot v primeru $\ell < i < m$. Več v literaturi [31].

Lastnosti:

- faza predprocesiranja v $O(m)$ času in prostoru,
- faza iskanja v $O(n)$ času,
- v najslabšem primeru izvede $\frac{3n}{2}$ primerjav.

2.1.8 Skip search

V fazi predprocesiranja za vsak simbol iz abecede v tabelo z shranimo vse indekse, pri katerih se ta simbol nahaja v vzorcu, torej za $c \in \Sigma$ $z[c] = \{i : 0 \leq i \leq m - 1 \wedge x[i] = c\}$. Če simbola ni v vzorcu, nam ni treba shraniti ničesar. Faza iskanja poteka tako, da preverjamo vsak m-ti simbol, kar je skupaj $\frac{n}{m}$ glavnih iteracij. V vsaki iteraciji za simbol $T[j]$ iz tabele $z[T[j]]$ preberemo indekse znaka v vzorcu, s čimer lahko določimo možne začetne pozicije okna. Za tem se izvajajo primerjave, dokler ne naletimo na neujemanje oziroma najdemo vzorec. Povejmo še, da bi v splošnem lahko pogoj v vrstici 19 pri implementaciji tudi izpustili, saj skrbi le zato, da pri indeksiranju ostanemo znotraj nizov. Bralec lahko prebere še literaturo [28].

Lastnosti:

- predprocesiranje v $O(m + \sigma)$ času in prostoru,
- faza iskanja v $O(mn)$ času.

2.1.8.1 Implementacija v Javi

```
1  public static void skipSearch(String P, int m, String T, int n) {
2          int i, j;
3          List ptr;
4          List[] z = new List[256];
5
6          /* Predprocesiranje */
7
8          for (i = 0; i < m; ++i) {
9                  ptr = new List();
10                 ptr.element = i;
11                 ptr.next = z[P.charAt(i)];
12                 z[P.codePointAt(i)] = ptr;
13         }
14
15         /* Iskanje */
16
17         for (j = m - 1; j < n; j += m)
18                 for (ptr = z[T.charAt(j)]; ptr != null; ptr = ptr.next)
19                         if (j - ptr.element + m <= n) {
20                                 if (P.equals(T.substring(j - ptr.element, j - ptr.element + m)))
21                                         System.out.println(j - ptr.element);
22                         }
23 }
```

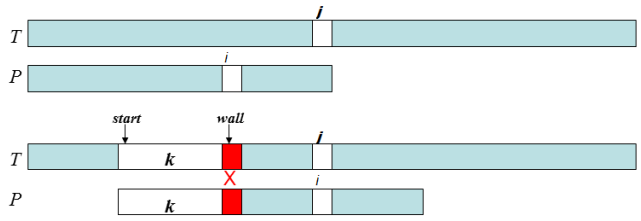
2.1.9 KMP skip search

Motivacija za algoritem je poskus izboljšave algoritma Skip search, da bi ta tekel v linearnem času. Ideja je, da pri izvajanju zamikov uporabimo tabeli zamikov algoritmov MP in KMP ($mpNext, kmpNext$). Predprocesiranje tako poleg shranjevanja seznama indeksov, kot je to pri algoritmu Skip search, zahteva še računanje obeh funkcij. Splošna situacija med fazo iskanja je prikazana na sliki 2.7. Algoritem sprva uporabi Skip search, da poravna $T[j] = P[i]$. Nato pregleduje znake od začetka okna proti desni, dokler ne naleti na *wall*, kar označuje prvo mesto neujemanja, k pa je dolžina ujemajoče se predpone. Algoritem nato izračuna *kmpStart* in *skipStart*. To so začetne pozicije okna, če bi vzorec zamaknili z algoritmom KMP oz. Skip. Če je $k = 0$, algoritem izvede zamik po algoritmu Skip search, sicer pa moramo za dolžino naslednjega zamika preveriti štiri scenarije:

1. Če $skipStart < kmpStart$, izvedemo zamik po algoritmu Skip, kar da novo vrednost za *skipStart* zato ponovno primerjamo *skipStart* in *kmpStart*.

2. Če je $kmpStart < skipStart < wall$, se izvede zamik po MP tabeli, kar da novo vrednost za $kmpStart$, zato ponovno primerjamo $skipStart$ in $kmpStart$.
3. Če je $skipStart = kmpStart$, se nov korak izvede pri $start = skipStart$.
4. Če je $kmpStart < wall < skipStart$, se nov korak izvede pri $start = skipStart$.

Več informacij lahko bralec poišče v literaturi [28].



Slika 2.7: Splošna situacija med fazo iskanja

Lastnosti:

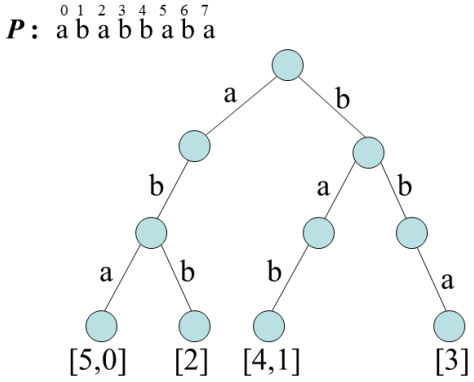
- predprocesiranje v $O(m + \sigma)$ času in prostoru,
- faza iskanja v $O(n)$ času.

2.1.10 Alpha skip search

Ideja algoritma temelji na algoritmu Skip search, ki za vsak simbol hrani seznam indeksov, kjer se ta simbol pojavi v vzorcu. Pri Alpha skip pa hranimo indekse odsekov vzorca dolžine $\ell = log_\sigma m$, kjer se le-ti pojavijo v vzorcu. To dosežemo tako, da v fazi predprocesiranja zgradimo drevo odsekov (slika 2.8). Faza iskanja poteka tako, da pregledujemo $T[j...j + \ell - 1]$ za vsak $j = k(m-\ell+1)-1$, kjer je k na intervalu $T[1, \frac{n-\ell}{m}]$, pri katerem je zgornja meja zaokrožena navzdol. Ko najdemo ujemanje odseka, ustrezno poravnamo okno in kot pri Skip searchu preverimo še ostale simbole. Več naj bralec prebere v literaturi [28].

Lastnosti:

- predprocesiranje v $O(m)$ času in prostoru,
- faza iskanja ima časovno zahtevnost $O(mn)$,
- pričakovano število znakovnih primerjav $O(log_\sigma m(\frac{n}{m-log_\sigma m}))$.



Slika 2.8: Drevo odsekov (faktorjev) vzorca.

2.1.11 Rabin-Karp

Namesto zapletenega izvajanja zamikov se algoritem Rabin-Karp izogne kvadratičnemu številu primerjav in išče pohitritev primerjanja vzorca z nizom z uporabo razpršilne funkcije. Namesto preverjanja za vsak $T[i]$, ali se tu pojavi vzorec, izgleda bolj učinkovito, če bi lahko preverili, ali je celotna vsebina trenutnega okna enaka vzorcu. Za to primerjavo je priročna uporaba razpršilne funkcije. Funkcija pretvori vsak niz v numerično vrednost. Rabin-Karp izkorišča dejstvo, da morata biti dva niza enaka, če imata enake vrednosti razpršilne funkcije. Vse, kar moramo storiti, je, da izračunamo razpršeno vrednost vzorca in nato poiskati podniz z isto razpršeno vrednostjo. Če bi naivno računali vrednosti $T[i+1...i+m]$ za vsak pomik okna za eno mesto v desno, bi za to potrebovali $O(mn)$, kar ni nič boljše od naivnega algoritma. Z uporabo verižne razpršilne funkcije (*rolling hash*) dosežemo izračun naslednje vrednosti $hash(T[i+1...i+m]) = rehash(T[i], T[i+m], hash(T[i...i+m-1])$ v konstantnem času in to slabost odpravimo. Izračunana razpršena vrednost v prejšnem koraku nam torej pomaga, da hitreje izračunamo razpršeno vrednost v trenutnem koraku, kjer smo samo odvzeli $T[i]$ in dodali $T[i+m]$.

2.1.11.1 Verižna razpršilna funkcija

Za boljšo predstavo o tem, kako funkcija računa novo vrednost, si oglejmo naslednji primer. Naj bo w beseda dolžine m. Potem je njena vrednost $hash(w)$ definirana tako: $hash(w[0...m-1]) = (w[0] * 2^{m-1} + w[1] * 2^{m-2} + ... + w[m-1] * 2^0)$ mod q, kjer je q veliko število [3]. Naslednja vrednost pa tako: $rehash(a, b, h) = ((h - a * 2^{m-1}) * 2 + b)$ mod q (glej vrstico 2 v imple-

mentaciji). Pazljivi moramo biti, da pri ujemanju vsebine preverimo še znak za znakom, ali se vzorec in del besedila ujemata, saj razpršilna funkcija zelo težko zagotovi, da imak čisto vsak podniz unikatno numerično vrednost (glej vrstico 30 v implementaciji).

Lastnosti:

- uporablja razpršilno funkcijo,
- časovna in prostorska kompleksnost predprocesiranja $O(m)$,
- faza preiskovanja ima v najslabšem primeru časovno kompleksnost reda $O(mn)$,
- povprečna časovna zahtevnost $O(n + m)$.

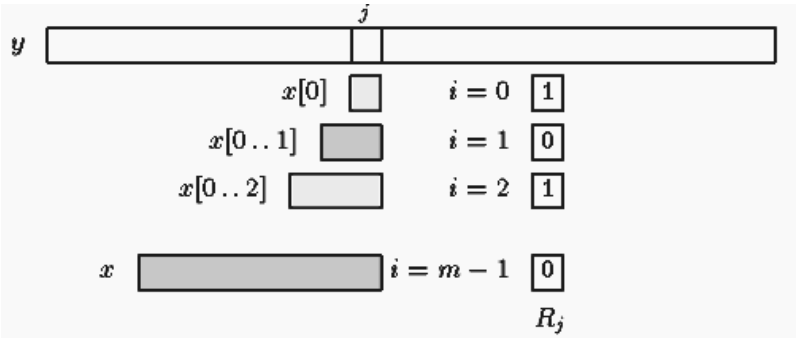
2.1.11.2 Implementacija v Javi

```
1  public static int rehash(char a, char b, int h, int d){
2          return ((((h) - (a)*d) << 1) + (b));
3          /* izracun naslednje razprsene vrednosti */
4  }
5
6  public static int rabinKarp(String T, String P) {
7          int d, hP, hT, i, j;
8
9          /* izracuna d = 2^(m-1) z levim zamikom */
10
11         for (d = i = 1; i < P.length(); ++i)
12                 d = (d<<1);
13
14         /* V fazi predprocesiranja izracunamo razprseno vrednost vzorca in
15          * razprseno vrednost prve dolzine drsecega okna, vzorca */
16
17         for (hT = hP = i = 0; i < P.length(); ++i) {
18                 hP = ((hP<<1) + (int)(P.charAt(i)));
19                 hT = ((hT<<1) + (int)(T.charAt(i)));
20         }
21
22         /* Preiskovanje */
23         j = 0;
24         while (j <= T.length()-P.length()) {
25                 /* ce je ujemanje uspesno izpisi indeks */
26                 if (hP == hT && P.equals(T.substring(j, j+P.length())))
27                         System.out.println(j);
28                 /* pogoj, da ne presezemo polja pri izbiri naslednjega znaka */
29                 if(j < T.length() - P.length())
30                         hT = rehash(T.charAt(j), T.charAt(j+P.length()), hT, d);
31                 ++j;
32         }
33         return 0;
34 }
```

2.1.12 ShiftOR

Algoritem za svoje delovanje izkorišča operacije nad biti. Naj bo R polje bitov velikosti m. Vektor R_j je vrednost polja R, potem ko preberemo znak $T[j]$ (2.9). Vsebuje informacije o vseh predponah vzorca, ki se končajo pri indeksu j za $0 < i \leq m - 1$:

$$R_j(i) = \begin{cases} 0 & if \quad P(0...i) = T(j-i...j) \\ 1 & \qquad\qquad sicer \end{cases}$$



Slika 2.9: Pomen vektorja Rj.

Oglejmo si kako iz vekorja R_j izračunamo vektor R_{j+1}. Za vsak $R_j(i) = 0$:

$$R_{j+1}(i+1) = \begin{cases} 0 & if \quad P[i+1] = T[j+1] \\ 1 & \qquad\qquad sicer \end{cases}$$

in

$$R_{j+1}(0) = \begin{cases} 0 & if \quad P[0] = T[j+1] \\ 1 & \qquad\qquad sicer \end{cases}$$

Če je $R_{j+1}(m-1) = 0$, pomeni, da smo našli vzorec. Iz R_j lahko dobimo R_{j+1} zelo hitro. Za vsak znak c iz abecede Σ naj bo S_c polje bitov velikosti m, tako da velja: za $0 \leq i < m - 1$, $S_c(i) = 0$, *če* $P[i] = c$. Polje S_c beleži indekse znaka c v vzorcu P, zato ga lahko izračunamo že v fazi predprocesiranja. Za izračun R_{j+1} potrebujemo dve operaciji nad biti - shift, or. $R_{j+1} = SHIFT(R_j)\ OR\ S_{T[j+1]}$. Da bi algoritem deloval učinkovito, moramo zagotoviti, da dolžina vzorca ni daljša od pomnilniške besede, sicer pa sama faza iskanja ni odvisna od dolžine abecede in dolžine vzorca. Za lažjo predstavo o preračunavanju vektorjev R_j naj si bralec ogleda še sliko (2.10), kjer

je prikazana faza iskanja. Algoritem je prepoznaven tudi pod drugimi imeni, za več informacij naj bralec prebere še literaturo [15].

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
		G	C	A	T	C	G	C	A	G	A	G	A	G	T	A	T	A	C	A	G	T	A	C	G
0	G	0	1	1	1	1	0	1	1	0	1	0	1	0	1	1	1	1	1	1	0	1	1	1	0
1	C	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	A	1	1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
3	G	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
4	A	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
5	G	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
6	A	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
7	G	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1

Slika 2.10: R12(7) = 0, kar pomeni, da smo našli vzorec na poziciji 12-8+1=5.

Lastnosti:

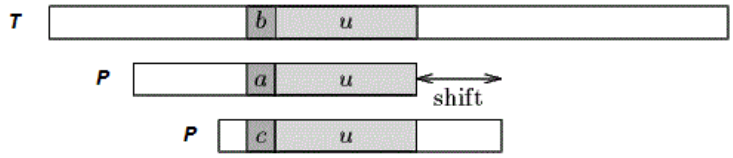
- uporablja operacije nad biti,
- predprocesiranje v $O(m + \sigma)$ času,
- faza iskanja v $O(n)$ času,
- lahko ga preuredimo v algoritem za približno iskanje podnizov.

2.1.12.1 Implementacija v Javi

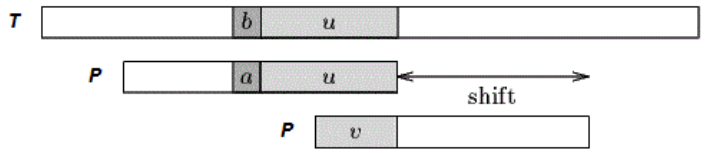
```
1  public static void soSearch(String P, String T) {
2          final int MAX_CHAR = 256; // najvisja numericna vrednost znaka abecede
3          int[] S = new int[MAX_CHAR];
4          int m = P.length();
5          int R;
6          int i;
7
8          R = ~1;
9
10         /* predprocesiranje */
11         for (i = 0; i < S.length; ++i)
12                 S[i] = ~0;
13         for (i = 0; i < m; ++i)
14                 S[P.charAt(i)] &= ~(1 << i);
15
16         /* iskanje */
17         for (i = 0; i < T.length(); ++i) {
18                 R |= S[T.charAt(i)];
19                 R <<= 1;
20                 if (0 == (R & (1 << m)))
21                         System.out.println(i - m + 1);
22         }
23 }
```

2.1.13 Boyer-Moore

Algoritem je eden najpomembnejših algoritmov s področja iskanja podnizov, saj je zelo učinkovit v običajnih aplikacijah. Uporabljajo ga npr. urejevalniki besedil pri izvajanju ukazov za iskanje in zamenjavo niza. Deluje tako, da bere znake od leve proti desni in začne pri najbolj desnem znaku vzorca. V primeru neujemanja ali ujemanja celotnega vzorca pri trenutnem znaku s pomočjo funkcij *good-suffix* in *bad-character* izračuna maksimalen zamik okna v desno, tako da izbere boljši rezultat od obeh funkcij (glej vrstico 14 v implementaciji). Recimo, da algoritem med svojim delovanjem naleti na neujemanje med znakom $P[i] = a$ in znakom $T[i + j] = b$. Potem mora veljati naslednje: $P[i + 1...m - 1] = T[i + j + 1...j + m - 1] = u$ in $P[i] \neq T[i + j]$. Funkcija *good-suffix* pomika okno v desno na dva načina. Skuša poravnati segmenta $T[i+j+1...j+m-1] = P[i+1...m-1]$ z najbolj desno pojavitvijo segmenta v P, ki je predznačen z drugačnim znakom od $P[i]$. Če tak segment slučajno ne obstaja, je zamik odvisen od poravnave najdaljše končnice P, ki je hkrati predpona P. Razliko med načinoma delovanja te funkcije prikazujeta sliki 2.11 in 2.12.



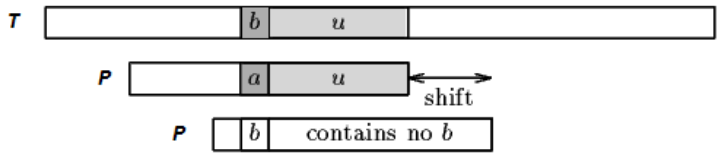
Slika 2.11: Segment u se znova pojavi v P predznačen z znakom c, ki se razlikuje od a.



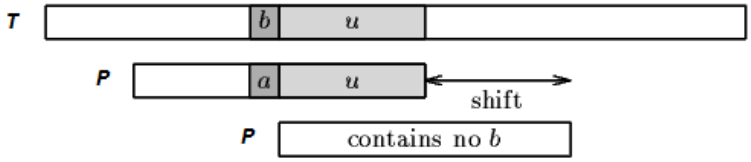
Slika 2.12: Samo končnica u-ja se pojavi kot predpona P.

Funkcija *bad-character* skuša poravnati znak $T[i + j]$ z njegovo najbolj desno pojavitvijo v $P[0...m-2]$ (slika 2.13). Če se $T[i+j]$ ne nahaja v vzorcu P, potem levi konec okna poravnamo z znakom $T[i + j + 1]$ (slika 2.14). Za

bolj formalno definicijo funkcij *good-suffix* in *bad-character* naj si bralec ogleda literaturo [7].



Slika 2.13: a se pojavi v P.



Slika 2.14: b se ne pojavi v P.

Lastnosti:

- deluje od leve proti desni,
- faza predprocesiranja ima časovno in prostorsko kompleksnost $O(m+\sigma)$,
- faza iskanja $O(mn)$,
- v najslabšem primeru $3n$ primerjav za neperiodičen vzorec,
- v najboljšem primeru $O(n/m)$.

2.1.13.1 Implementacija v Javi

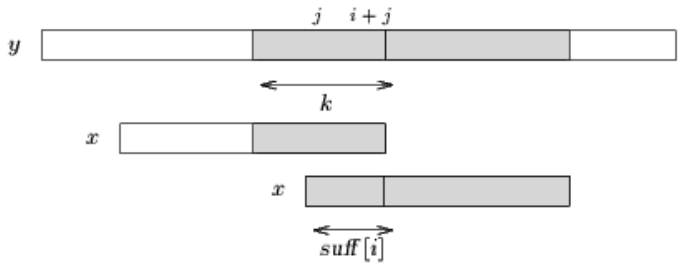
```
1  /* poisce vse indekse vseh pojavitev vzorca P v tekstu T */
2  public static int boyerMoore(String T, String P) {
3          if (P.length() == 0) {
4                  return 0;
5          }
6          int charTable[] = makeCharTable(P);
7          int offsetTable[] = makeOffsetTable(P);
8          for (int i = P.length() - 1, j; i < T.length();) {
9                  for (j = P.length() - 1; P.charAt(j) == T.charAt(i); --i, --j) {
10                         if (j == 0) {
11                         System.out.println(i); //izpise indeks, ce se vsi znaki ujemajo
12                       break;
13                         }
14                 }
15                 //max zamik
```

```
16                  i += Math.max(offsetTable[P.length() - 1 - j], charTable[T.charAt(i)]);
17          }
18          return -1;
19  }
20
21  /* tabela zamikov (velikost odvisna od velikosti abecede) za bad character funkcijo */
22   private static int[] makeCharTable(String P) {
23          final int ALPHABET_SIZE = 256;
24          int[] table = new int[ALPHABET_SIZE];
25          for (int i = 0; i < table.length; ++i) {
26                  table[i] = P.length();
27          }
28          for (int i = 0; i < P.length() - 1; ++i) {
29                   table[P.charAt(i)] = P.length() - 1 - i;
30          }
31          return table;
32  }
33
34  /* tabela zamikov za good-suffix funkcijo */
35  private static int[] makeOffsetTable(String P) {
36          int[] table = new int[P.length()];
37          int lastPrefixPosition = P.length();
38          for (int i = P.length() - 1; i >= 0; --i) {
39                  if (isPrefix(P, i + 1)) {
40                          lastPrefixPosition = i + 1;
41                  }
42                  table[P.length() - 1 - i] = lastPrefixPosition - i + P.length() - 1;
43          }
44          for (int i = 0; i < P.length() - 1; ++i) {
45                  int slen = suffixLength(P, i);
46                  table[slen] = P.length() - 1 - i + slen;
47          }
48          return table;
49  }
50
51  private static boolean isPrefix(String P, int p) {
52          for (int i = p, j = 0; i < P.length(); ++i, ++j) {
53                  if (P.charAt(i) != P.charAt(j)) {
54                          return false;
55                  }
56          }
57          return true;
58  }
59
60  /* Vrne maksimalno dolzino podniza, ki se konca pri indeksu p in je koncnica */
61   private static int suffixLength(String P, int p) {
62          int len = 0;
63          for (int i = p, j = P.length() - 1;
64                   i >= 0 && P.charAt(i) == P.charAt(j);
65                   --i, --j) {
66                   len += 1;
67          }
68          return len;
69  }
```

2.1.14 Apostolico-Giancarlo

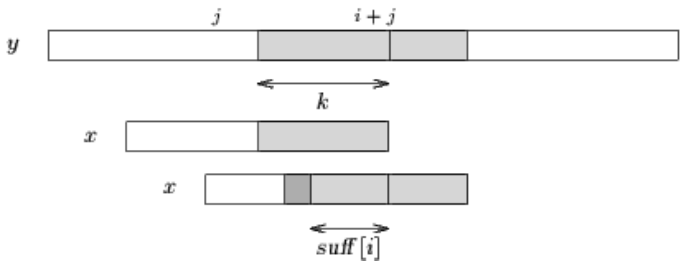
Motivacija za algoritem je odprava slabosti algoritma BM, ki na vsakem koraku zavrže informacijo o znakih, ki so se ujemali. Ideja je, da si na koncu vsakega koraka zapomnimo najdaljšo dolžino končnice, ki se je ujemala s tekstom (BM deluje od desne proti levi) in to informacijo shranimo v tabelo *skip*. Recimo, da je algoritem med svojim delovanjem v prejšnjem koraku uspel najti končnico dolžine k na poziciji $i + j$, kjer je $0 < i < m$. Iz tega sledi $skip[i + j] = k$. Definirajmo še eno tabelo, za katero velja $suff[i]$ je dolžina najdaljše končnice P, ki se konča na poziciji i v P (glej vrstico 60 pri implementaciji Boyer-Moore). Če algoritem uspešno izvede primerjave na odseku $T[i + j + 1..j + m - 1]$, se pri poskusu na poziciji j lahko razvijejo štirje scenariji:

1. $k > suff[i]$ in $suff[i] = i + 1$
   Pomeni, da smo našli pojavitev vzorca pri indeksu j in $skip[j + m - 1]$ postane m (slika 2.15). Zatem se izvede zamik dolžine vzorca.

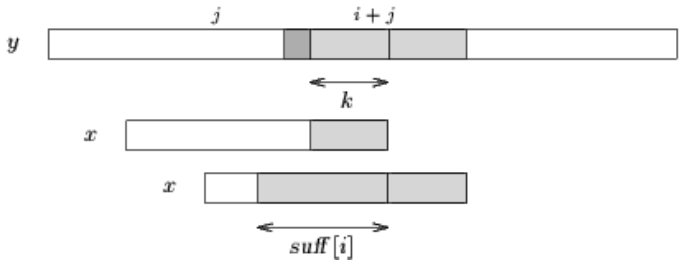


Slika 2.15: Našli smo pojavitev vzorca.

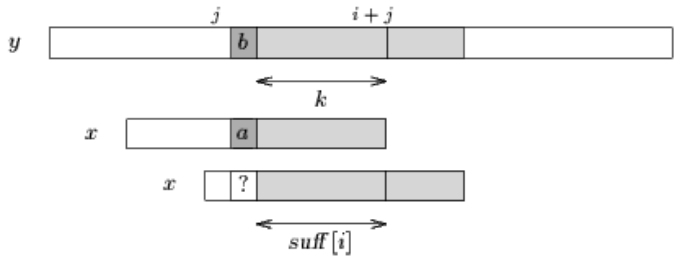
2. $k > suff[i]$ in $suff[i] \leq i$
   Pomeni, da je do neujemanja prišlo med $P[i-suff[i]]$ in $T[i+j-suff[i]]$ in $skip[j + m - 1]$ postane $m - 1 - i + suff[i]$ (slika 2.16). Zamik izvedemo tako, da poiščemo večjo vrednost med $BC[T[i+j-suff[i]]]$ in $GS[i - suff[i] + 1]$, kjer je BC tabela zamikov za funkcijo *bad-character* (BM implementacija vrstica 6, 22) in GS tabela zamikov za funkcijo *good-suffix* (BM implementacija vrstica 7, 35).
3. $k < suff[i]$
   Pomeni, da je do neujemanja prišlo med $P[i - k]$ in $T[i + j - k]$ in $skip[j + m - 1]$ postane $m - 1 - i + k$ (slika 2.17). Zamik izvedemo podobno kot prej z uporabo $BC[T[i + j - k]]$ in $GS[i - k + 1]$.
4. $k = suff[i]$
   Samo v tem primeru moramo preskočiti odsek $T[i + j - k + 1...i + j]$, da lahko nadaljujemo s primerjavami med $T[i + j - k]$ in $P[i - k]$ (slika 2.18).



Slika 2.16:



Slika 2.17:



Slika 2.18: a je različen b.

V vseh primerih je edina potrebna informacija le dolžina najdaljše končnice P, ki se konča na poziciji i v P. Več o algoritmu lahko bralec izve v literaturi [16].

Lastnosti:

- predprocesiranje v $O(m + \sigma)$ času in prostoru,
- faza iskanja ima časovno zahtevnost $O(n)$,
- $\frac{3n}{2}$ primerjav v najslabšem primeru.

2.1.15 Tuned Boyer-Moore

Algoritem je poenostavljena verzija algoritma BM, ki kljub precej preprosti implementaciji daje kar dobre rezultate na praktičnih primerih. Motivacija za algoritem je odprava prepogostih primerjanj, ali znak vzorca ustreza znaku okna, saj je to najbolj požrešen del pri algoritmih za ujemanje nizov. Ideja je izvesti nekaj zamikov, preden sploh začnemo s primerjanjem znakov. Algoritem uporablja samo funkcijo zamikov *bad-character* in skuša najti $P[m-1]$ v T v treh slepih zaporednih zamikih in začne pri $T[0...m-1]$. Dokler ne najde $P[m-1]$, se izvajanje treh slepih zamikov s pomočjo funkcije *bad-character* v zanki ponavlja. Da je to mogoče, je potrebnih nekaj modifikacij pri tabeli zamikov za funkcijo *bad-character* ($shift = BC[P[m-1]]$ in $BC[P[m-1]] = 0$, 0 je zato, da pri izvajanju slepih zamikov ostanemo pri indeksu, kjer smo našli zadnji znak vzorca, dokler se preostali slepi zamiki ne izvedejo) in dodati m pojavitev $P[m-1]$ na konec T. Oboje v fazi predprocesiranja. Ko najde $P[m-1]$, se izvedejo primerjave v poljubnem vrstnem redu še za preostale znake in izvede se zamik dolžine $shift$. Za bolj podrobno analizo naj si bralec ogleda še literaturo [17].

Lastnosti:

- enostavna implementacija,
- kvadratična časovna zahtevnost v najslabšem primeru, vendar hiter na praktičnih primerih.

2.1.16 Zhu-Takaoka

Algoritem je zopet izpeljanka algoritma BM, in sicer sta avtorja spremenila funkcijo *bad-character* tako, da deluje za dva zaporedna znaka [18]. Faza iskanja je identična algoritmu BM le pri izbiri večjega zamika od obeh funkcij, namesto običajne funkcije *bad-character* uporabimo spremenjeno (implementacija BM vrstica 16). Med iskanjem se primerjave izvajajo od desne proti levi, okno je pozicionirano na odseku $T[j...j+m-1]$ in do neujemanja pride med $P[m-k]$ in $T[j+m-k]$ med tem, ko $P[m-k+1...m-1] = T[j+m-k+1...j+m-1]$. *Bad-character* funkcija pri tem v izračun zamika vzame znaka $T[j+m-2]$ in $T[j+m-1]$. Predprocesiranje spremenjene funkcije *bad-character* izgleda tako, da za vsak par znakov a, b iz Σ izračunamo najbolj desno pojavitev niza ab v $P[0...m-2]$ (slika 2.19).

Za a, b v Σ: $ztBC[a,b] = k$, če:

- $k < m-2$ in $P[m-k...m-k+1] = ab$ in ab se ne pojavi v $P[m-k+2...m-2]$ ali
- $k = m-1$ in $P[0] = b$ in ab se ne pojavi v $P[0...m-2]$ ali
- $k = m$ in $P[0] \neq b$ in ab se ne pojavi v $P[0...m-2]$.

$ztBc$	A	C	G	T
A	8	8	2	8
C	5	8	7	8
G	1	6	7	8
T	8	8	7	8

i	0	1	2	3	4	5	6	7
$x[i]$	G	C	A	G	A	G	A	G
$bmGs[i]$	7	7	7	2	7	4	7	1

Slika 2.19: Izgled tabele ztBC na konkretnem primeru.

Lastnosti:

- uporaba dveh zaporednih znakov za računanje zamika s funkcijo *bad-character*,
- faza iskanja ima v najslabšem primeru kvadratično časovno zahtevnost,
- predprocesiranje $O(m + \sigma^2)$ časovno in prostorsko kompleksnost.

2.1.17 Berry-Ravindran

Algoritem je hibrid med algoritmoma Quick search in Zhu-Takaoka. Kot Zhu-Takaoka uporablja spremenjeno funkcijo *bad-character*, ki deluje za dva zaporedna znaka na desni strani okna. V fazi predprocesiranja za vsak par znakov (a, b) iz abecede izračunamo najbolj desno pojavitev ab v aPb. Dobljene dolžine zamikov:

$$brBC[a,b] = min \begin{cases} 1 \ \ if \ \ P[m-1] = a, \\ m-i+1 \ \ if \ \ P[i]P[i+1] = ab, \\ m+1 \ \ if \ \ P[0] = b, \\ m+2 \ \ sicer \end{cases}$$

V fazi iskanja je v nekem koraku okno pozicionirano na odseku $T[j...j+m-1]$. Algoritem poskuša s pregledovanjem od leve proti desni poiskati vzorec, v primeru neuspeha pa izvede zamik dolžine $brBC[T[j+m], T[j+m+1]]$. Več v literaturi [29].

Lastnosti:

- predprocesiranje v $O(m+\sigma^2)$ času in prostoru,
- faza iskanja v $O(mn)$ času.

2.1.18 Quick search

Algoritem je poenostavljena verzija algoritma BM, ki uporablja samo *bad-character* funkcijo zamikov in v nasprotju z BM izvaja primerjave od leve proti desni (primerjave bi se lahko izvajale v poljubnem vrstnem redu). V nekem koraku je okno pozicionirano na odseku $T[j...j+m-1]$ in dolžina zamika je vedno vsaj za eno mesto v desno. Znak $T[j+m]$ je torej vedno vpleten v naslednjem koraku, zato ga lahko že v trenutnem koraku uporabimo za zamik s funkcijo *bad-character* [19]. Algoritem na j-tem koraku preveri, ali je našel vzorec, ter ga izpiše, potem pa j zamakne za $qsBC[j+m]$, tudi če ni našel ujemanja. Za tak način delovanja moramo prej omenjeno funkcijo malo spremeniti: za c v Σ velja $qsBC[c] = min\{i : 0 < i \leq m \wedge P[m-i] = c\}$, če se c pojavi v P, sicer je zamik $m+1$. Faza iskanja ima kvadratično časovno zahtevnost, vendar pri kratkih vzorcih in dolgih abecedah daje dobre rezultate na praktičnih primerih.

Lastnosti:

- enostavna implementacija,
- predprocesiranje v $O(m+\sigma)$ času in $O(\sigma)$ prostoru,
- faza iskanja ima $O(mn)$ časovne zahtevnosti.

2.1.19 Maximal shift

Ideja algoritma je med fazo iskanja skenirati znake vzorca v takem vrstnem redu, da najprej preizkusimo tiste, pri katerih privede do daljšega zamika. Pri izvajanju zamikov algoritem izbere boljšo med funkcijama *bad-character* (različica Quick search) in *good-suffix*, ki sta izračunani v fazi predprocesiranja. *Good-suffix* je prilagojena vrstnemu redu primerjav. V fazi predprocesiranja je potrebno znake vzorca še urediti po padajočem vrstnem redu glede

na dolžino zamika oziroma po razdalji do njihove ponovne leve pojavitve ali do začetka vzorca. Za vzorec brez ponavljajočih se znakov se torej primerjave izvedejo zaporedno od desne proti levi. Več v literaturi [19].

Lastnosti:

- faza predprocesiranja v $O(m^2 + \sigma)$ času in $O(m + \sigma)$ prostoru,
- faza iskanja v $O(mn)$ času.

2.1.20 Optimal mismatch

Algoritem je različica algoritma Quick search, ki med fazo iskanja skuša skenirati znake vzorca v takem vrstnem redu, da najprej preizkusi tiste znake, ki imajo manjšo frekvenco v besedilu. S tem želimo povečati verjetnost zgodnje zaznave napake. V fazi predprocesiranja moramo znake urediti v naraščajočem vrstnem redu po frekvenci pojavitve. Pri izvajanju zamikov algoritem izbere boljšo med funkcijama *bad-character* (različica Quick search) in *good-suffix*, ki sta izračunani v fazi predprocesiranja. Funkcija *good-suffix* je prilagojena vrstnemu redu primerjav. Več lahko bralec prebere v literaturi [19].

Lastnosti:

- potrebuje frekvence simbolov v besedilu,
- faza predprocesiranja v $O(m^2 + \sigma)$ času in $O(m + \sigma)$,
- faza iskanja v $O(mn)$ času.

2.1.21 Reverse Colussi

Algoritem je izpeljanka algoritma Boyer-Moore, vendar pa opravlja znakovne primerjave v specifičnem vrstnem redu podanem v tabeli h. Po vzoru BM začne pri zadnjem znaku vzorca, če je ujemanje uspešno, nato indeks naslednje primerjave prebere iz tabele h, sicer pa vzorec zamakne s pomočjo tabele $rcBc$ (implementacija vrstica 98 in 108). Kadar pride do neujemanja pri že vsaj enem uspešnem ujemanju pri trenutni postavitvi okna, vzorec zamaknemo s tabelo $rcGs$ (implementacija 112). Za izgradnjo tabel h, $rcBc$ in $rcGs$ je potrebnega nekaj zahtevnejšega predprocesiranja [11]:

- za vsak i, kjer $0 \leq i \leq m$ ustvarimo dve množici:
  - $Pos(i) = \{k : 0 \leq k \leq i \wedge P[i] = P[i-k]\}$,
  - $Neg(i) = \{k : 0 \leq k \leq i \wedge P[i] \neq P[i-k]\}$.

- $1 \leq k \leq m$ naj bo $hmin(k)$ najmanjše število ℓ, kjer $\ell \geq k-1$ in $k \notin Neq(i)$ za vsak i, kjer $\ell < i \leq m-1$,
- $0 \leq \ell \leq m-1$ naj bo $kmin(\ell)$ najmanjše število k, kjer $hmin(k) = \ell \geq k$, če obstaja tak k, sicer $kmin(\ell) = 0$,
- $0 \leq \ell \leq m-1$ naj bo $rmin(\ell)$ najmanjše število k, kjer $r > \ell$ in $hmin(r) = r-1$.

Vrednost $h(0)$ nastavimo na $m-1$. Potem iz $kmin(\ell)$ v naraščajočem vrstnem redu izberemo vse indekse $h(1), ...h(d)$ tako, da $kmin(h(i)) \neq 0$ in nastavimo $rcGs(i)$ na $kmin(h(i))$ za $1 \leq i \leq d$. Potem v naraščajočem vrstnem redu izberemo indekse $h(d+1), ..., h(m-1)$ in nastavimo $rcGs(i)$ na $rmin(h(i))$ za $d < i < m$. Vrednost $rcGs(m)$ je nastavljena na periodo P-ja. Tabela $rcBc$ pa je definirana tako: $rcBc(a,s) = min\{k : (k = m \vee P[m-k-1] = a) \wedge (k > m-s-1 \vee P[m-k-s-1] = P[m-s-1])\}$. Za izračun tabele $rcBc$ definiramo še: za vsak $c \in \Sigma$ je $locc(c)$ indeks najbolj desne pojavitve c v $P[0...m-2]$. $locc(c)$ je -1, če se c ne pojavi v $P[0...m-2]$. Rezultat takega procesiranja na konkretnem primeru je prikazan na sliki 2.20.

a	A	C	G	T
$locc[a]$	6	1	5	-1

$rcBc$	0	1	2	3	4	5	6	7	8
A	0	8	5	5	3	3	3	1	1
C	0	8	6	6	6	6	6	6	6
G	0	2	2	2	4	4	2	2	2
T	0	8	8	8	8	8	8	8	8

i	0	1	2	3	4	5	6	7	8
$x[i]$	G	C	A	G	A	G	A	G	
$link[i]$	-1	-1	-1	-1	0	2	3	4	
$hmin[i]$	0	7	3	7	5	5	7	6	7
$kmin[i]$	0	0	0	2	0	4	7	1	0
$rmin[i]$	7	7	7	7	7	7	7	8	0
$rcGs[i]$	0	2	4	7	7	7	7	7	
$h[i]$	7	3	5	6	0	1	2	4	

Slika 2.20: Rezultat predprocesiranja z Reverse Colussi algoritmom.

Lastnosti:

- predprocesiranje v $O(m^2)$ času in $O(m\sigma)$ prostoru,
- faza iskanja $O(n)$,
- v najslabšem primeru $2n$ znakovnih primerjav.

2.1.21.1 Implementacija v Javi

```
1  public static void preRc(String P, int m, int h[], int[][] rcBc,
2          int rcGs[], String alphabet) {
3          int a, i, j, k, q, r = 0, s;
4          int[] kmin = new int[m];
5          int[] link = new int[m];
6          int[] hmin = new int[m + 1];
7          int[] rmin = new int[m];
8          int[] locc = new int[256];
9          /*
10          * locc za indeks uporablja numericno vrednost znaka abecede torej bi
11          * lahko velikost omejili le na numericno vrednost zadnjega znaka
12          * abecede, podobno tudi pri tabeli rcBc
13          */
14
15         /* racunanje locc in link */
16         for (a = 0; a < alphabet.length(); ++a)
17                 locc[alphabet.charAt(a)] = -1;
18         link[0] = -1;
19         for (i = 0; i < m - 1; ++i) {
20                 link[i + 1] = locc[P.charAt(i)];
21                 locc[P.charAt(i)] = i;
22         }
23
24         /* racunanje rcBc */
25         for (a = 0; a < alphabet.length(); ++a)
26                 for (s = 1; s <= m; ++s) {
27                         i = locc[alphabet.charAt(a)];
28                         j = link[m - s];
29                         while (i - j != s && j >= 0)
30                                 if (i - j > s)
31                                         i = link[i + 1];
32                                 else
33                                         j = link[j + 1];
34                         while (i - j > s)
35                                 i = link[i + 1];
36                         rcBc[alphabet.charAt(a)][s] = m - i - 1;
37                 }
38
39         /* racunanje hmin */
40         k = 1;
41         i = m - 1;
42         while (k <= m) {
43                 while (i - k >= 0 && P.charAt(i - k) == P.charAt(i))
44                         --i;
45                 hmin[k] = i;
46                 q = k + 1;
47                 while (hmin[q - k] - (q - k) > i) {
48                         hmin[q] = hmin[q - k];
49                         ++q;
50                 }
51                 i += (q - k);
52                 k = q;
53                 if (i == m)
54                         i = m - 1;
55         }
```

```
56
57          /* racunanje kmin */
58          for (k = m; k > 0; --k)
59                  kmin[hmin[k]] = k;
60
61          /* racunanje rmin */
62          for (i = m - 1; i >= 0; --i) {
63                  if (hmin[i + 1] == i)
64                          r = i + 1;
65                  rmin[i] = r;
66          }
67
68          /* racunanje rcGs */
69          i = 1;
70          for (k = 1; k <= m; ++k)
71                  if (hmin[k] != m - 1 && kmin[hmin[k]] == k) {
72                          h[i] = hmin[k];
73                          rcGs[i++] = k;
74                  }
75          i = m - 1;
76          for (j = m - 2; j >= 0; --j)
77                  if (kmin[j] == 0) {
78                          h[i] = j;
79                          rcGs[i--] = rmin[j];
80                  }
81          rcGs[m] = rmin[0];
82  }
83
84  /* abeceda je urejen niz vseh znakov, ki se vsaj enkrat pojavijo v T */
85  public static void RC(String P, int m, String T, int n, String alphabet) {
86          int i, j, s;
87          int[][] rcBc = new int[256][m + 1];
88          int[] rcGs = new int[m + 1];
89          int[] h = new int[m];
90
91          /* Predprocesiranje */
92          preRc(P, m, h, rcBc, rcGs, alphabet);
93          /* Iskanje */
94          j = 0;
95          s = m;
96          while (j <= n - m) {
97                  while (j <= n - m && P.charAt(m - 1) != T.charAt(j + m - 1)) {
98                          // zamik za nek znak iz abecede pri nekem indeksu
99                          s = rcBc[T.charAt(j + m - 1)][s];
100                         j += s;
101                 }
102                 // brez tega pogoja lahko presezemo polje, ko se priblizujemo koncu T
103                 if (j > n - m)
104                         break;
105                 for (i = 1; i < m && P.charAt(h[i]) == T.charAt(j + h[i]); ++i)
106                         ;
107                 if (i >= m)
108                         System.out.println(j);
109                 s = rcGs[i];
110                 j += s;
111         }
112 }
```

2.1.22 Horspool

Funkcija *bad-character*, ki jo uporablja BM pri izvajanju zamikov, ni preveč učinkovita pri majhnih abecedah. Nigel Horspool je menil [12], da kadar je abeceda zelo velika v primerjavi z vzorcem, postane funkcija zelo uporabna in jo lahko uporabimo samostojno ter dobimo precej spodoben, a zelo enostaven algoritem (npr. ASCII tabela pri iskanjih v urejevalnikih besedil, kjer je abeceda zelo dolga, običajno pa iščemo krajšo besedo ali le del besede).

Lastnosti:

- poenostavljen BM algoritem lahek za implementacijo,
- vrstni red primerjav ni pomemben,
- preprocesiranje v $O(m + \sigma)$ času in $O(\sigma)$ prostoru,
- faza iskanja v $O(mn)$,
- povprečno število primerjav za en znak je med $\frac{1}{\sigma}$ in $\frac{2}{\sigma+1}$.

2.1.22.1 Implementacija v Javi

```
1  /* racunanje bad-character funkcije */
2  private static int[] makeCharTable(String P) {
3          final int ALPHABET_SIZE = 256;
4          int[] table = new int[ALPHABET_SIZE];
5          for (int i = 0; i < table.length; ++i) {
6                  table[i] = P.length();
7          }
8          for (int i = 0; i < P.length() - 1; ++i) {
9                  table[P.charAt(i)] = P.length() - 1 - i;
10         }
11         return table;
12 }
13
14 public static void horspool(String P, int m, String T, int n) {
15         int j;
16         char c;
17
18         /* Predprocesiranje s funkcijo izsposojeno od Boyer-Moore algoritma */
19         int[] table = makeCharTable(P);
20
21         /* Iskanje */
22         j = 0;
23         while (j <= n - m) {
24                 c = T.charAt(j + m - 1);
25                 if (P.charAt(m - 1) == c && P.equals(T.substring(j, j + m)))
26                         System.out.println(j);
27                         j += table[c];
28                 }
29 }
```

2.1.23 Raita

Tim Raita je poizkušal izboljšati Horspoolov algoritem tako, da je spremenil fazo iskanja. Pri trenutni postavitvi okna sprva primerja najbolj desni znak; če se ujemata, preveri najbolj levi znak okna; če se ujemata, primerja znak na sredini; če je se ujemata, preveri še ostale indekse v poljubnem vrstnem redu. Opazil naj bi, da se njegov algoritem dobro obnese pri iskanju vzorcev v angleških besedilih. Za več informacij si lahko bralec ogleda še literaturo [13].

Lastnosti:

- mešanica specifičnega in poljubnega vrstnega reda primerjav,
- predprocesiranje v $O(m + \sigma)$ času in $O(\sigma)$ prostoru,
- faza iskanja v $O(mn)$ času.

2.1.23.1 Implementacija v Javi

```
1  private static int[] makeCharTable(String P) {
2          final int ALPHABET_SIZE = 256;
3          int[] table = new int[ALPHABET_SIZE];
4          for (int i = 0; i < table.length; ++i) {
5                  table[i] = P.length();
6          }
7          for (int i = 0; i < P.length() - 1; ++i) {
8                  table[P.charAt(i)] = P.length() - 1 - i;
9          }
10         return table;
11 }
12
13 public static void raita(String P, int m, String T, int n) {
14         int j;
15         char c, firstCh, secondCh, middleCh, lastCh;
16         /* Predprocesiranje */
17         int[] table = makeCharTable(P);
18         firstCh = P.charAt(0);
19         middleCh = P.charAt(m / 2);
20         lastCh = P.charAt(m - 1);
21         /* Razlicica faze iskanja, kjer srednji znak preverimo 2krat (redundanca) */
22         j = 0;
23         while (j <= n - m) {
24                 c = T.charAt(j + m - 1);
25                 if (lastCh == c
26                                 && middleCh == T.charAt(j + m / 2)
27                                 && firstCh == T.charAt(j)
28                                 && P.substring(1, m - 1).equals(T.substring(j + 1, j + m - 1)))
29                                 System.out.println(j);
30                 j += table[c];
31         }
32 }
```

2.1.24 Smith

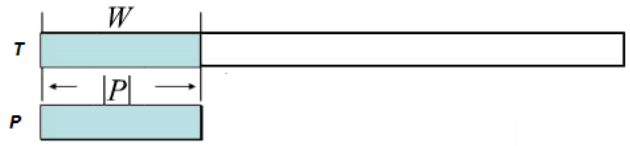
Smith je opazil, da računanje zamika za znak desno od najbolj desnega znaka v oknu včasih vrne krajši zamik, kakor če ga bi izračunali za najbolj desni znak. Ideja je uporabiti funkciji zamikov *bad-character* (Horspool, Quick Search) in pri zamiku izbrati boljšo, torej z večjo vrednostjo. Več v literaturi [30].

Lastnosti:

- faza predprocesiranja v $O(m + \sigma)$ času in $O(\sigma)$ prostoru,
- faza iskanja v $O(mn)$ času.

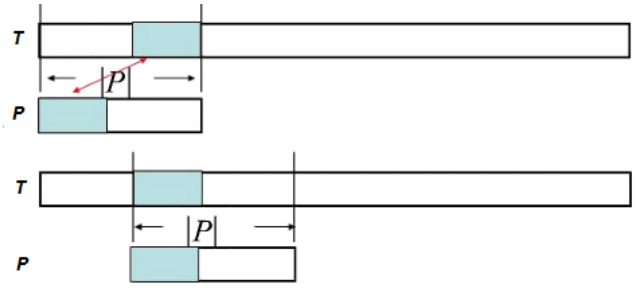
2.1.25 Reverse factor

Z algoritmi tipa BM pregledujemo znake od desne proti levi in skušamo najti ujemanje končnice vzorca. Lahko pa ravno tako pregledujemo znake od desne proti levi, toda skušamo najti ujemanje predpone vzorca, da bi izboljšali dolžino zamikov. Ideja je poiskati najdaljšo končnico drsečega okna, ki je hkrati predpona vzorca. Pri tem se pojavijo trije scenariji, ki so prikazani na slikah 2.21, 2.22 in 2.23. Lahko bi rekli, da pri vsaki postavitvi okna v končnici skušamo poiskati vsaj neko predpono vzorca, če smo bili pri ujemanju vzorca neuspešni.

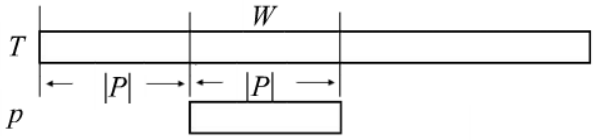


Slika 2.21: Vzorec se ujema s tekstom.

Iskanje take končnice okna oziroma predpone vzorca lahko dosežemo z uporabo minimalnega *DKA* (*suffix automaton* $S(w)$), ki sprejema jezik $L(S(w)) = \{u \in \Sigma^* : \exists v \in \Sigma^* \wedge w = vu\}$. V fazi predprocesiranja moramo zgraditi tak *DKA* za nazaj prebran vzorec P^R. Primer takega avtomata vidimo na sliki 2.24. V fazi iskanja začnemo v stanju q_0 in z branjem znakov okna od leve proti desni korakoma prehajamo v naslednje stanje, v kolikor je le-to definirano za trenutni vhodni znak. Končno stanje pomeni ujemanje končnice okna s predpono vzorca (slika 2.22) oz. celotnim vzorcem (slika 2.21). Dolžina ujemajoče se predpone je enaka dolžini poti od q_0 do zadnjega končnega stanja na tej poti, preden smo naleteli na neujemanje znakov. Ko poznamo dolžino



Slika 2.22: Končnica okna se ujema s predpono vzorca. Okno zamaknemo.



Slika 2.23: Taka končnica ne obstaja, okno pomaknemo za dolžino vzorca.

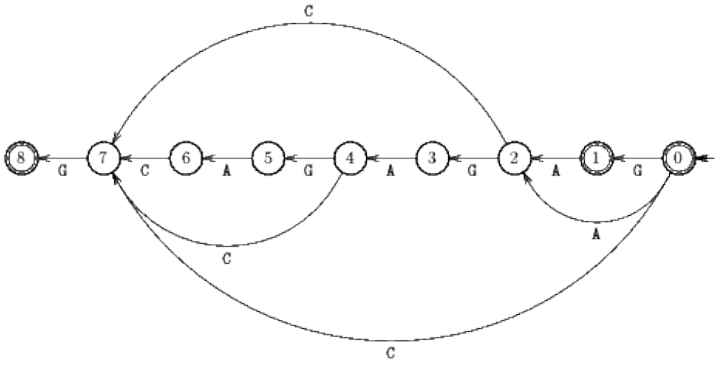
najdaljše predpone, trivialno izračunamo desni zamik. Več podrobnosti lahko bralec prebere v literaturi [20].

Lastnosti:

- hiter na praktičnih primerih za dolge vzorce in majhne abecede,
- predprocesiranje v $O(m)$ času in prostoru,
- faza iskanja v $O(mn)$ času.

2.1.26 Turbo Reverse factor

Algoritem Reverse factor se da izboljšati, da faza iskanja poteka v linearnem času. Dovolj je, da si zapomnimo predpono vzorca u, ki jo je algoritem uspel najti v prejšnjem koraku. Pokažemo lahko, da je v naslednjem koraku zadosti, če preberemo samo desno stran u-ja. Splošna situacija izgleda tako, da algoritem po najdeni predponi u primerja faktor v dolžine $m - |u|$, to je odsek neposredno desno od u. Če je niz z faktor niza w, definiramo še $disp(z, w)$ kot najmanjše število $d > 0$, da velja $w[m - d - |z| - 1...m - d] = z$. Če v ni faktor vzorca P, potem se zamik izračuna kot pri algoritmu Reverse factor. Če je v končnica P, potem smo našli pojavitev vzorca. Če pa je v faktor P,



Slika 2.24: Končno stanje pomeni ujemanje končnice okna s predpono vzorca oziroma ujemanje celotnega vzorca.

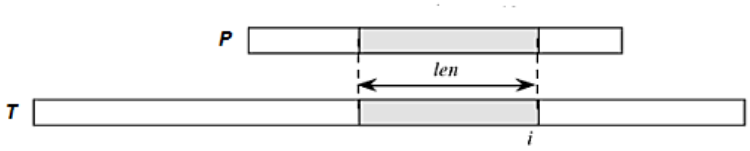
potem je zadosti, da ponovno skeniramo le $min(per(u), |u|/2)$ najbolj desnih znakov v u. Če je u periodičen ($per(u) \leq \frac{|u|}{2}$), naj bo z končnica u dolžine $per(u)$. Potem se z lahko pojavi le pri razdaljah, ki so večkratnik $per(u)$, iz česar sledi, da ima najmanjša ustrezna končnica niza uv in hkrati predpona vzorca P dolžino $|uv| - disp(zv, x) = m - disp(zv, x)$. Torej je dolžina zamika, ki ga moramo izvesti, enaka $disp(zv, x)$. Če u ni periodičen ($per(u) > \frac{|u|}{2}$), je očitno, da se P ne more pojaviti v levem delu u-ja dolžine $per(u)$. Zato je pri iskanju nedefinirane povezave v *suffix DKA* zadosti, da skeniramo desni del u-ja dolžine $|u| - per(u) < \frac{|u|}{2}$. Funkcijo *disp* vgradimo v avtomat. Več naj bralec prebere v literaturi [21].

Lastnosti:

- predprocesiranje v $O(m)$,
- faza iskanja $O(n)$,
- izvede največ $2n$ primerjav.

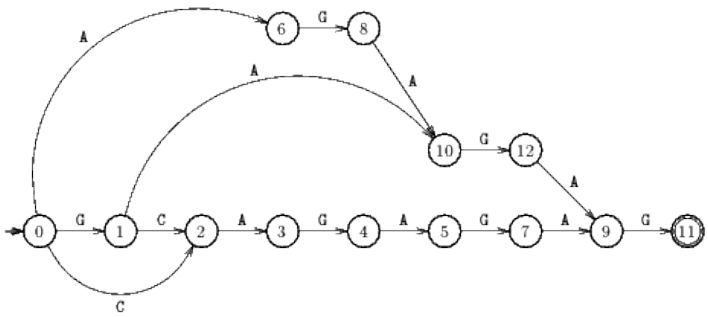
2.1.27 Forward DAWG

Algoritem pregleduje znake od leve proti desni in za vsako pozicijo v besedilu izračuna najdaljši faktor vzorca, ki se konča pri tej poziciji (slika 2.25). To je mogoče z izgradnjo *suffix DKA* (ali *DAWG* - directed acyclic word graph) za vzorec P. Primer takega *DKA* vidimo na sliki 2.26. Algoritem prične v stanju q_0 in za vsak zaporedno prebrani znak $T[j]$, za katerega je definiran prehod iz trenutnega v neko naslednje stanje, preide v naslednje stanje. Če prehod iz trenutnega stanja za tak vhod ni definiran, se trenutno stanje osveži z začetnim



Slika 2.25: Najdaljši faktor vzorca dolžine *len*, ki se konča na poziciji i.

stanjem. Na sliki 2.26 prehodi v začetno stanje niso označeni. Prihod v končno stanje še ne pomeni, da smo našli celoten vzorec (lahko, da gre le za faktor vzorca), saj je končno stanje dosegljivo po različnih poteh. Težavo odpravimo tako, da sproti izračunamo dolžino poti od začetnega stanja in če je v končnem stanju dolžina poti enaka m, smo našli vzorec. Več informacij v literaturi [22].



Slika 2.26: *Suffix DKA*, ki ga uporablja Forward DAWG.

Lastnosti:

- v najslabšem primeru je časovna zahtevnost $O(n)$,
- izvede natako n znakovnih primerjav.

2.1.28 Backward nondeterministic DAWG

Algoritem oponaša Reverse factor in za simulacijo *suffix DKA* za P^R izrablja paralelizem bitov. V fazi predprocesiranja za vsak znak c iz abecede v tabelo B shrani bitno masko, ki je nastavljena samo če je $x_i = c$ (implementacija vrstica 7). Med iskanjem se stanje hrani v besedi $d = d_{m-1}...d_0$. V k-ti iteraciji je bit d_i postavljen, če velja $P[m-i...m-1-i+k] = T[j+m-k...j+m-1]$. V iteraciji 0 je $d = 1^{m-1}$. V vsaki iteraciji zelo hitro izračunamo novo stanje d' s pomočjo tabele B in prejšnjega stanja $d' = (d \wedge B[T_j]) << 1$ (implementacija vrstica 20). Ujemanje vzorca dobimo, če v m-ti iteraciji velja $d_{m-1} = 1$.

Kadarkoli je $d_{m-1} = 1$, pomeni, da je algoritem našel ujemanje predpone pri tej postavitvi okna. S pomočjo najdaljše ujemajoče se predpone lahko nato izračunamo zamik. Več podrobnosti naj bralec poišče v literaturi [23].

Lastnosti:

- učinkovit samo, če je dolžina vzorca krajša od pomnilniške besede.

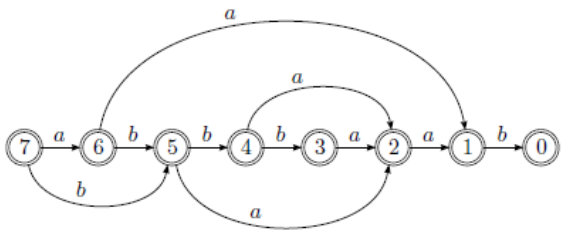
2.1.28.1 Implementacija v Javi

```
1  public static void BNDM(String P, int m, String T, int n) {
2          final int MAX_CHAR = 256;
3          int[] B = new int[MAX_CHAR];
4          int i, j, s, d, last;
5
6          /* predprocesiranje */
7          s = 1;
8          for (i = m - 1; i >= 0; i--) {
9                  B[P.charAt(i)] |= s;
10                 s <<= 1;
11         }
12
13         /* iskanje */
14         j = 0;
15         while (j <= n - m) {
16                 i = m - 1;
17                 last = m;
18                 d = ~0;
19                 while (i >= 0 && d != 0) {
20                         d &= B[T.charAt(j + i)];
21                         i--;
22                         if (d != 0) {
23                                 if (i >= 0)
24                                         last = i + 1;
25                                 else
26                                         System.out.println(j);
27                         }
28                         d <<= 1;
29                 }
30                 j += last;
31         }
32 }
```

2.1.29 Backward oracle

Algoritem je različica algoritma Reverse factor, ki namesto *suffix DKA* za P^R uporablja *suffix oracle* za P^R, ki ga izračuna v fazi predprocesiranja. Struktura *oracle* je dejansko DKA, ki prepoznava vsaj vsak faktor P-ja in še nekaj drugih besed. Formalno ga opišemo kot peterko $O(P) = \{Q, m, Q, \Sigma, \delta\}$. Q vsebuje

natanko $m+1$ stanj, kjer je m začetno stanje in so vsa stanja končna. Za jezik, ki ga sprejema *oracle*, velja $L(S(P)) \subseteq L(O(P))$. Kljub temu, da prepoznava tudi besede, ki niso faktor vzorca, lahko *oracle* uporabimo za iskanje vzorcev, saj je edina beseda dolžine m, ki jo *oracle* sprejme, kar P^R. Iskanje poteka tako, da začne v stanju q_0 in menja stanja vse dokler prehod ni definiran za vhodni simbol. Med iskanjem je v nekem trenutku dolžina najdaljše predpone vzorca, ki je končnica skeniranega dela besedila, krajša od poti opravljene v $O(P^R)$ in sicer od q_0 do zadnjega obiskanega končnega stanja. Če poznamo to dolžino, trivialno izračunamo dolžino ustreznega zamika. Primer *oracle* strukture naj si bralec ogleda na sliki 2.27, več podrobnosti pa lahko najde v literaturi [24].



Slika 2.27: oracle za P = baabbba

Lastnosti:

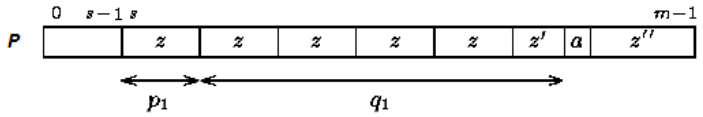
- hiter za dolge vzorcev in majhne abecede,
- predprocesiranje v $O(m)$ času in prostoru,
- faza iskanja v $O(mn)$ času.

2.1.30 Galil-Seiferas

Ideja algoritma je, da v fazi predprocesiranja izvedemo popolno faktorizacijo nad vzorcem (slika 2.28). To pomeni iskanje dekompozicije P na uv, tako da ima v največ eno *prefix periodo* in je dolžina u enaka odmiku periode od začetka vzorca (več o *prefix periodi* lahko bralec prebere v literaturi [25]). Torej $u = P[0...s-1]$ in $v = P[s...m-1]$. Iskanje poteka tako, da najprej iščemo v in če smo uspešni, še naivno preverimo, ali se odsek u nahaja natanko pred tem. Med iskanjem $P[s...m-1]$ v T se zgodita dva scenarija:

- Če se je odsek $P[s...s+p_1+q_1-1]$ ujemal, izvedemo zamik dolžine p_1 in primerjave se nadaljujejo pri $P[s+q_1]$.

- Če pride do neujemanja pri $P[s+q]$, kjer $q \neq p_1 + q_1$, izvedemo zamik dolžine $\frac{q}{k} + 1$ in primerjave se nadaljujejo pri $P[0]$.



Slika 2.28: *z'* je predpona *z* in *z'a* ni predpona *z*. Dolžina periode je *p1*.

Lastnosti:

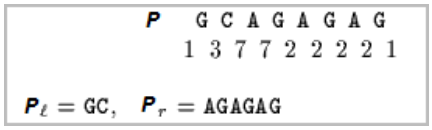
- predprocesiranje v $O(m)$ času,
- iskanje je reda $O(n)$,
- izvede največ $5n$ znakovnih primerjav.

2.1.31 Two way

Ideja algoritma je, da v fazi predprocesiranja izberemo kritično razdelitev vzorca na P_l in P_r, da velja $P = P_l P_r$. Naj bo (u, v) razdelitev P. Ponovitev v (u, v) je beseda w, za katero veljata naslednji dve lastnosti:

- w je končnica u ali u je končnica w,
- w je predpona v ali v je predpona w.

Dolžina ponovitve v (u, v) se imenuje globalna perioda, dolžina najkrajše take ponovitve pa lokalna perioda, ki jo označimo $r(u, v)$. Vsaka razdelitev (u, v) ima vsaj eno ponovitev, torej $1 \leq r(u, v) \leq |P|$ in je kritična, če je $r(u, v) = per(P)$. Pri kritični razdelitvi sta pri odseku u globalna in lokalna perioda enaki. Algoritem izbere tako kritično razdelitev (P_l, P_r), kjer $|P_l| < per(P)$ in je dolžina $|P_l|$ minimalna. Najprej moramo izračunati najdaljšo končnico z vzorca P (maximal suffix of pattern) za leksikografsko ureditev $\leq$ in nato še analogno z' za obraten vrstni red, iz česar dobimo $|P_l| = max\{|z|, |z'|\}$. Faza iskanja poteka tako, da algoritem sprva preverja znake od leve proti desni na odseku P_r in nato še znake na odseku P_l od desne proti levi. Pri neujemanju k-teqa znaka v P_r, se izvede zamik dolžine k. Pri primerjavah v levem delu P_l pa se pri neujemanju izvede zamik dolžine $per(P)$. Algoritem še izboljšamo tako, da si po periodnem zamiku zapomni dolžino ujemajoče se predpone in se v naslednjem koraku izogne ponovnemu skeniranju tega dela. Bralec lahko prebere še literaturo [27].



Slika 2.29: Primer kritične razdelitve, števila so lokalne periode.

Lastnosti:

- leksikografska ureditev,
- predprocesiranje v $O(m)$ času in konstantnem prostoru,
- faza iskanja v $O(n)$,
- izvede največ $2n - m$ primerjav.

2.1.32 String matching on ordered alphabets

Ideja algoritma je, da med pregledovanjem od leve proti desni vsaj približno izračunamo periodo končnice ujemajoče se predpone pred mestom neujemanja. Potrebujemo le nekaj dodatnega prostora in ta aproksimacija je dovolj za delovanje algoritma v linearnem času kljub temu, da ne izvajamo nobenega predprocesiranja. Za izračun periode na predponi izvedemo MS razcep (Maximal suffix decomposition [26]) in dobimo uv, kjer je $v = w^e w'$ in w' je predpona w, $|w|$ pa je perioda v. Dobljen v imenujemo *maximal suffix* (maksimalen glede na leksikografsko urejenost npr. $a < b < c$) in po zamiku dolžine $|w|$ je lahko prednost to, da se algoritem zaradi lastnosti MS razcepa pri ponovnem računanju izogne računanju *maximal suffix*-a od začetka.

Lastnosti:

- leksikografska ureditev,
- ni predprocesiranja,
- faza iskanja je reda $O(n)$,
- izvede največ $6n + 5$ znakovnih primerjav.

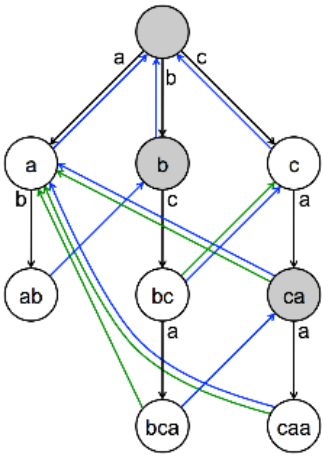
2.2 Končna množica vzorcev

2.2.1 Aho-Corasick

Algoritem AC [5] je namenjen simultanemu iskanju indeksov ujemanja vzorcev (omejenih s končno množico) in besedila. Naj bo $\beta = \{P_1, ..., P_k\}$ množica vseh vzorcev. V fazi predprocesiranja na podlagi te množice s pomočjo drevesa ključnih besed zgradimo avtomat (glej primer s slike 2.30). Ta struktura ima po eno vozlišče za vsako predpono vsakega izmed elementov množice. V konkretnem primeru s slike 2.30 bi množica izgledala tako $\beta = \{a, ab, bc, bca, c, caa\}$. Algoritem deluje tako, da bere iz besedila po en znak in se pomika po vozliščih drevesa, začenši pri korenu. Na vsakem koraku skuša poiskati sina; če ta ne obstaja, išče sina v vozlišču z najdaljšo končnico glede na trenutni vzorec in postopek rekurzivno ponavlja vse do korena. Npr: vzorec *caa* ima končnice $[aa], [a], []$. Po modri povezavi pridemo v vozlišče, ki ima najdaljšo končnico trenutnega vozlišča v celotnem grafu. Po črnih povezavah se premaknemo, ko je ujemanje uspešno, oziroma ko pri sestavljanju vzorca iz besedila uspešno dodamo en znak. Če torej prvi prebrani znak teksta ne ustreza nobeni predponi vzorcev, ostanemo v korenu. Z belo barvo so označena vozlišča, kjer naletimo na ujemanje vzorca. Povezave zelene barve so povezave do vzorcev, ki smo jih našli na trenutnem indeksu. Dobimo pa jih tako, da sledimo modrim povezavam dokler v vozlišču ne naletimo na enega izmed vzorcev. Pri izpisovanju indeksov vzorcev nam zelene povezave tako povedo, da smo na tem indeksu našli še nek drug vzorec. Prednost algoritma AC v primerjavi z izvajanjem nekega algoritma za iskanje enega vzorca večkrat zaporedoma $O(|P_1| + n + ... + |P_k| + n) = O(m + kn)$ se pokaže v nizji časovni zahtevnosti. Združuje ideji zamikov algoritma KMP in končnih avtomatov.

Lasnosti:

- iskanje s končno množico vzorcev,
- konstrukcija avtomata reda $O(|P_1| + ... + |P_k|) = m$,
- časovna kompleksnost algoritma $O(m+n+z)$, kjer je z število pojavitev vzorcev v besedilu T dolžine n.



Slika 2.30: primer AC avtomata

2.2.2 Commentz-Walter

Podobno kot AC tudi ta algoritem [6] simultano odkriva več vzorcev v besedilu. Ravno tako v fazi predprocesiranja zgradi drevo ključnih besed. Na sliki 2.31 je primer takega drevesa. Vsa vozlišča so označena z nekim znakom, razen korena, ki je označen s praznim znakom. Veljati mora tudi, da če sta dva vozlišča sinova nekega vozlišča, potem sta označena z različnim znakom. Naj bo $\beta = \{P_1, ..., P_k\}$ množica vseh vzorcev, ki jih želimo poiskati v besedilu. Za dejanski primer vzemimo sliko 2.31 $\beta = \{cacbaa, acb, aba, acbab, ccbab\}$. Za $h = 1...k$ obstaja vozlišče v_h, ki predstavlja nazaj prebrano besedo w_h^R. Vsakemu vozlišču dodamo še funkcijo $out(v) = \{w; w^R = w(v), w \in \beta\}$. Algoritem deluje tako, da bere znak po znak iz besedila upoštevajoč Boyer-Moore idejo in posledično izvaja primerjave od desne proti levi. Če glede na vhodni znak najde sina v drevesu (začne pri korenu), v tem vozlišču izvede funkcijo $out(v)$, nato pa prebere naslednji znak iz besedila (faza skeniranja). Če sin v drevesu ne obstaja, pomeni, da se noben izmed vzorcev ne ujema z delom besedila, ki je trenutno prebran. V tem primeru izvede zamik S v desno, ki ga izračuna s pomočjo spodaj opisanih funkcij $char(a), shift1(v), shift2(v)$ in zopet začne pri korenu drevesa (faza zamika). Obe fazi ponavlja, dokler ne preišče celotnega besedila, zamike pa izvede brez izgube pojavitve kakšnega vzorca. Definicija zamika je naslednja: $S(v, d_{i-j}) = min(max(shift1(v), char(d_{i-j} - j - 1)), shift2(v))$. Rezultat funkcije je dolžina zamika. Definirajmo še funkcijo: $char(a) = min(d(v), wmin + 1)$, kjer je $d(v)$ globina vozlišča v drevesu (koren ima globino 0), $wmin$ pa je dolžina najkrajšega vzorca. Definicija funkcij $shift1(v)$ in $shift2(v)$ pa je osnovana na množici vozlišč. Za vsako vozlišče,

razen korena, velja: $set1(v) = \{v'; w(v') = uw(v)$, kjer je u neprazna beseda$\}$ in $set2(v) = \{v'; v' \in set1(v)$ in $out(v') \neq \emptyset\}$.

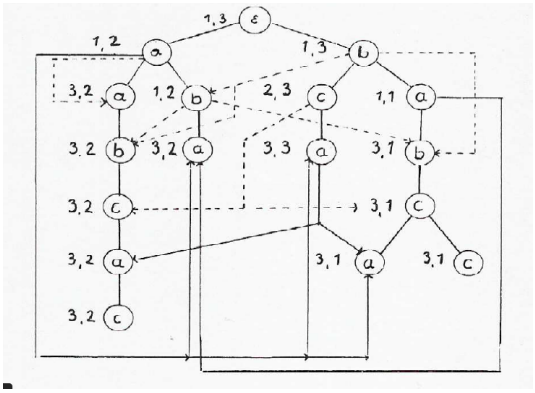
$$shift1(v) = \begin{cases} 1 & v = koren \\ min(k; k = \{d(v') - d(v), v' \in set1(v)\} \cup \{wmin\}) & sicer \end{cases}$$

$$shift2(v) = \begin{cases} wmin & v = koren \\ min(k; k = \{d(v') - d(v), v' \in set2(v)\} \cup & \\ \{shift2(\text{oče od } v')\}) & sicer \end{cases}$$

Na sliki 2.31 so poleg vozlišč označene funkcije $shift1(v)$ in $shift2(v)$. Črtkane in polne povezave vodijo do vozlišč $set1(v)$, polne povezave pa do vozlišč $set2(v)$.

Lastnosti:

- drevo obrnjenih ključnih besed,
- primerjave izvaja od desne proti levi,
- zamik S temelji na kombinaciji hrevristik,
- faza predprocesiranja $O(m)$, kjer je m vsota dolžin vzorcev,
- algoritem teče v linearnem času za skupno število primerjav.



Slika 2.31: Primer CW drevesa

2.3 Neskončna množica vzorcev

2.3.1 DKA

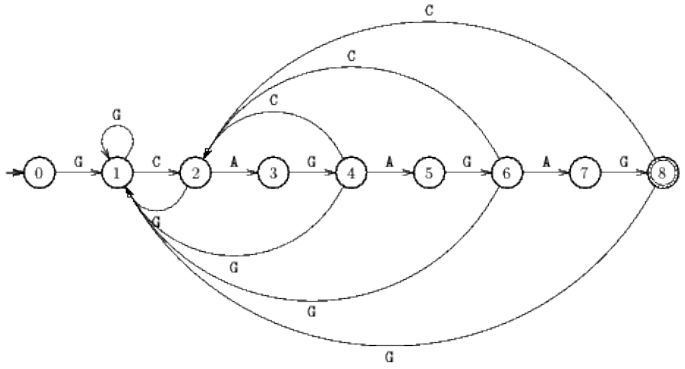
Oglejmo si, zakaj je deterministični končni avtomat primeren za uporabo, kadar imamo neskončno množico vzorcev, ki se lahko pojavijo v nekem besedilu. V fazi predprocesiranja moramo zgraditi minimalni DKA, ki bo prepoznaval jezik, katerega besede so vzorci. Bolj formalno nek avtomat M zapišemo kot peterko: $M = (Q, \Sigma, \delta, q_0, F)$:

- Q je množica vseh stanj avtomata, za vsako predpono nekega vzorca obstaja neko stanje q;
- Σ je množica vseh vhodnih simbolov;
- δ je funkcija prehodov $Q \times \Sigma = Q$:
  - (q, a, qa) samo, če je qa tudi predpona vzorca P, sicer
  - (q, a, p) tako, da je p najdaljša končnica v qa in qa je predpona vzorca P;
- q_0 je začetno stanje;
- F je množica končnih stanj (končno stanje pomeni, da smo našli nek vzorec).

Za boljšo predstavo o tem, kako izgleda konstruiran minimalen DKA, si oglejmo primer s slike 2.32. V naravi avtomatov je, da se lahko poljubnokrat vrnejo v neko stanje, npr. stanje q_1 s slike 2.32. Ta lastnost nam omogoča, da iščemo poljubno dolge vzorce, ki jih je lahko neskončno. DKA deluje tako, da iz besedila bere znak po znak in se glede na trenutni vhodni znak in trenutno stanje deterministično odloči za naslednje stanje, kot je opisano v funkciji prehodov. Začetno stanje je q_0. Končno stanje pomeni, da smo našli nek vzorec. Za podrobnejši vpogled v iskanje podnizov z avtomati naj bralec pregleda še literaturo [8].

Lasnosti:

- za konstrukcijo DKA potrebujemo $O(m + \sigma)$ časa in $O(m\sigma)$ prostora (odvisno od dolžine vzorca in abecede),
- fazo iskanja je mogoče izvesti v $O(n)$ času.



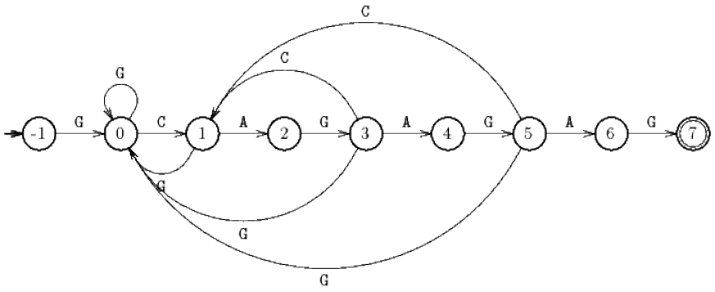
Slika 2.32: Mankajoče povezave vodijo v stanje q0.

2.3.2 Simon

Glavna pomankljivost minimalnega DKA je prostor, ki je potreben za konstrukcijo avtomata. Simon je opazil, da so nekateri prehodi med stanji odveč, in prihranil nekaj prostora potrebnega za hranjenje povezav. Ohranil je samo naprej usmerjene povezave, ki gredo od predpone vzorca dolžine k do predpone dolžine $k+1$ za $0 \leq k < m$ in nazaj usmerjene povezave, ki gredo od predpone vzorca dolžine k do manjše neničelne predpone [9]. Število takih povezav je torej omejeno z dvakratno dolžino vzorca $2m$. Vse ostale povezave vodijo v začetno stanje, zato jih lahko odstranimo. Stanje je označeno z dolžino njegove predpone zmanjšano za 1 zato, da je vsaka povezava, ki vodi v stanje $i - 1 \leq i \leq m - 1$ označena s $P(i)$, tako nam ni treba hraniti oznak za povezave. Naprej usmerjene povezave so jasne zaradi vzorca in jih ravno tako ne hranimo, potrebno je samo shraniti pomembne nazaj usmerjene povezave. Glej sliko 2.33. Uporabimo tabelo L velikosti $m-2$. Vsak element tabele $L(i)$ je seznam možnih stanj, v katere lahko pridemo iz stanja i. Za stanje $m-1$ nam ni treba shraniti seznama, vendar moramo izračunati število ℓ, tako da je $\ell+1$ dolžina najdaljšega roba vzorca P. V fazi predprocesiranja moramo torej izračunati tabelo L in število ℓ. Iskanje poteka analogno kot pri DKA, razlika je le, da ko Simonov DKA najde ujemanje vzorca, trenutno stanje osveži s stanjem ℓ.

Lastnosti:

- ekonomična implementacija minimalnega DKA,
- predprocesiranje (čas, prostor) $O(m)$,
- faza iskanja v času $O(m+n)$ (neodvisno od dolžine abecede),
- največje skupno število primerjav $2n-1$.

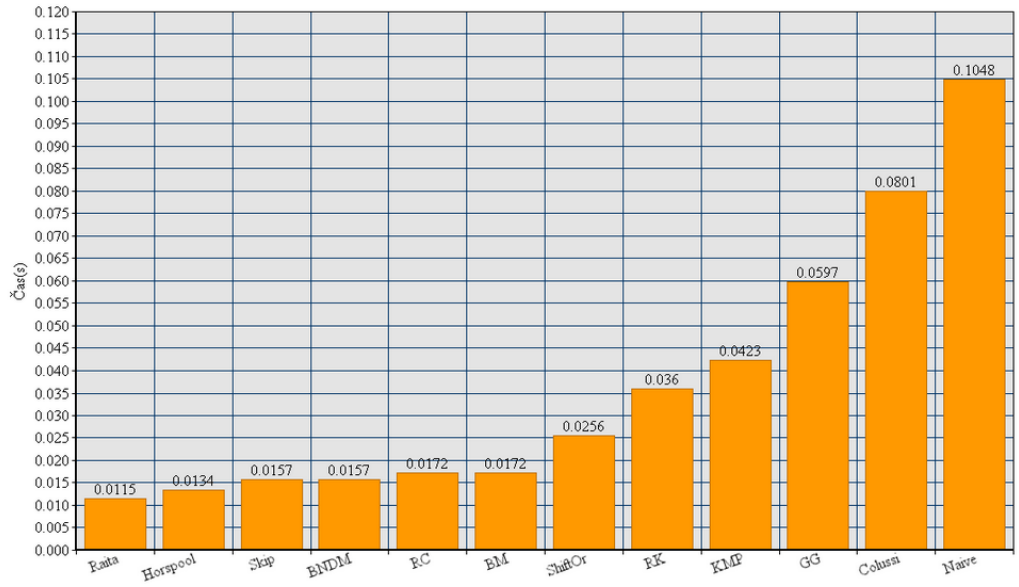


Slika 2.33: Simonov izboljšan *DKA*.

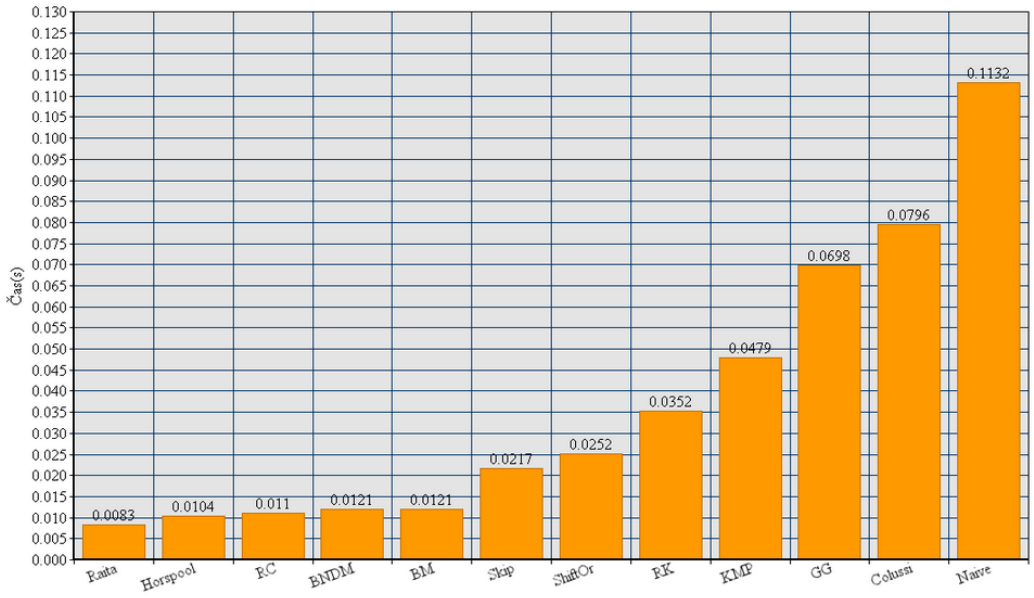
Poglavje 3

Rezultati praktične uporabe

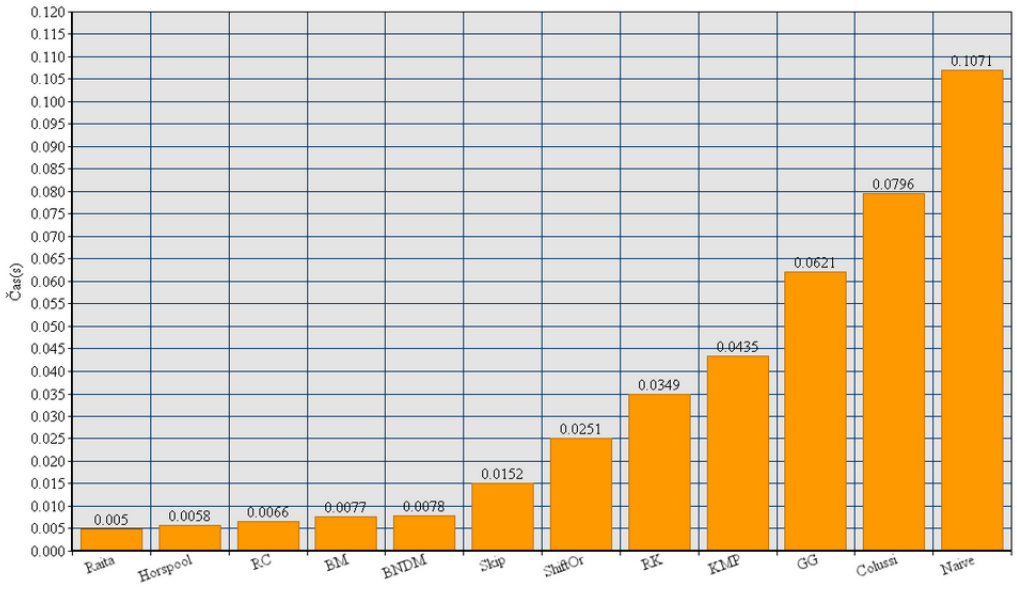
Izbrane algoritme smo preizkusili na tekstovni datoteki veliki 1.2 MB (Holy Bible [4]), ki vsebuje 4,329.250 znakov. Stolpični diagrami na slikah 3.1, 3.2, 3.3, 3.4, 3.5 in 3.6 prikazujejo, koliko časa je porabil posamezen algoritem, da je preiskal celotno besedilo in našel vse pojavitve izbranega vzorca vključno s predprocesiranjem. Manj pomeni boljše. Na sliki 3.7 pa je prikazan skupen porabljen čas.



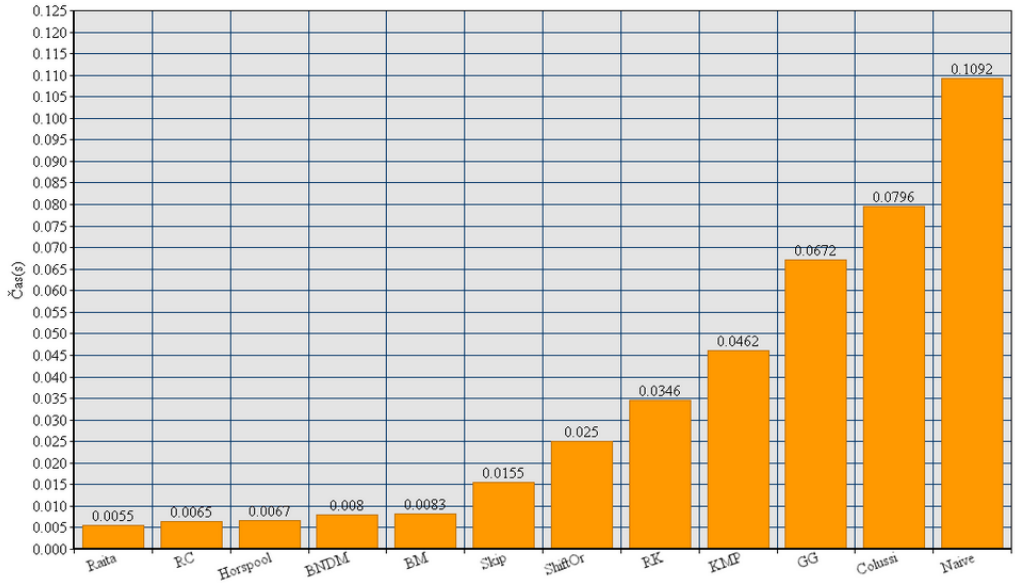
Slika 3.1: Diagram porabljenega časa posameznega algoritma za vzorec 'baby'.



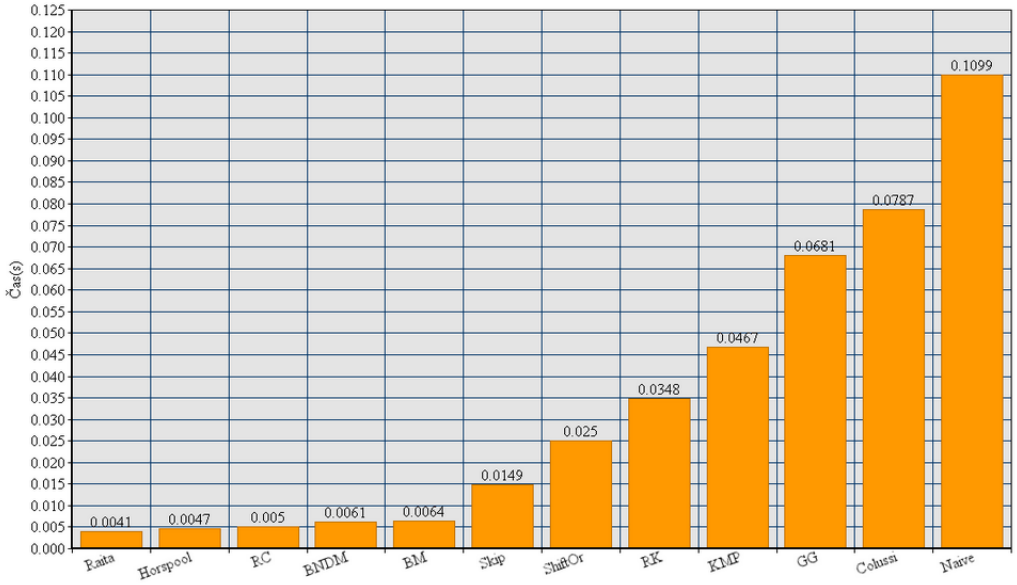
Slika 3.2: Diagram porabljenega časa posameznega algoritma za vzorec 'therefore the'.



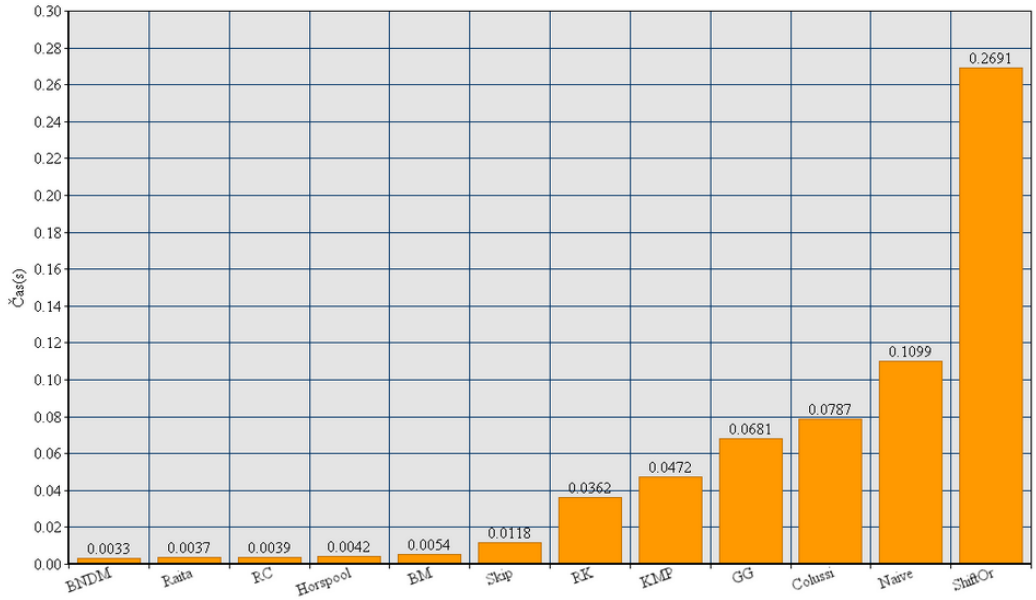
Slika 3.3: Diagram porabljenega časa posameznega algoritma za vzorec 'son of man is lord'.



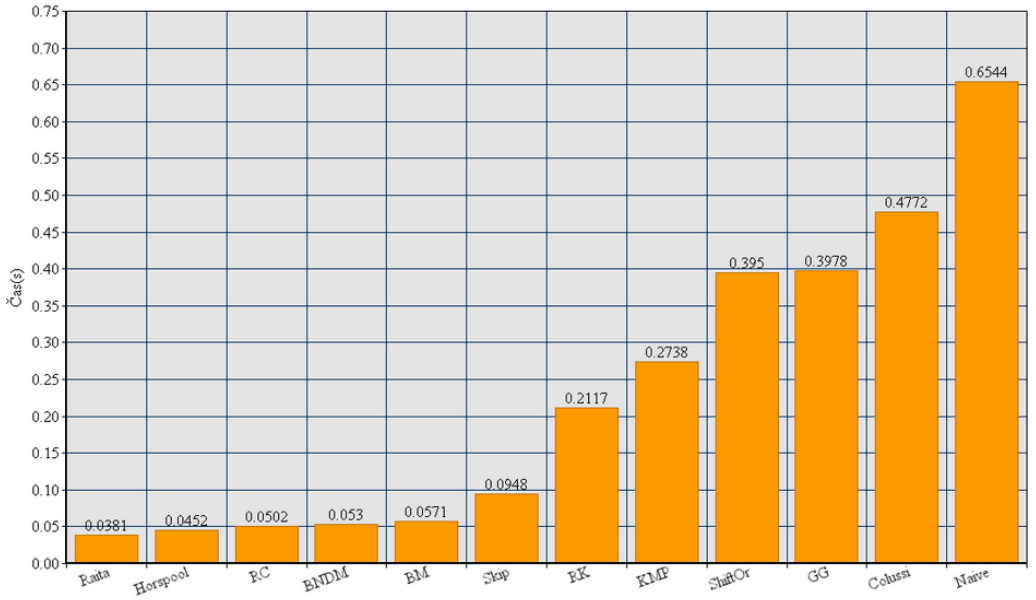
Slika 3.4: Diagram porabljenega časa posameznega algoritma za vzorec 'he answered them not'.



Slika 3.5: Diagram porabljenega časa posameznega algoritma za vzorec 'that the spirit of the holy'.



Slika 3.6: Diagram porabljenega časa posameznega algoritma za vzorec 'things which have been kept secret from the foundation of the world'.



Slika 3.7: Diagram porabljenega časa posameznegea algoritma za vse vzorce skupaj.

Poglavje 4

Zaključek

V nalogi smo analizirali algoritme za iskanje podnizov oziroma vzorcev v nekem daljšem nizu. Povedali smo zakaj, so ti algoritmi v računalništvu pomembni in kje se uporabljajo. Opisali smo slabost naivnega algoritma in kot njegovo izboljšavo predstavili ostale algoritme, pri katerih smo skušali opisati glavno idejo in posebne lastnosti posameznega algoritma. Časovno zahtevnost faze iskanja naivnega algoritma $O(mn)$ je z ustreznim algoritmom mogoče znižati na $O(n)$. Algoritme smo po načinu delovanja razdelili v več skupin in jih nekaj tudi implementirali v Javi. Opazili smo, da je veliko algoritmov med seboj strateško podobnih, kar nakazuje, da so se algoritmi s ponovnim preučevanjem postopoma razvijali in izboljševali. Implementirane algoritme smo preizkusili na daljšem besedilu (Biblija) in rezultate predstavili v diagramski obliki. Ugotovili smo, da je bil pri izbranih algoritmih faktor časovne pohitritve preiskovanja celotnega besedila v primerjavi z naivnim algoritmom približno od 1,25 do 20. V splošnem je najhitreje deloval algoritem Raita in s tem potrdil svojo lastnost. Izkaže se tudi, da je bil za učinkovito delovanje algoritma ShiftOr najdaljši vzorec predolg. Poleg tega so se algoritmi, ki pregledujejo znake od desne proti levi, v splošnem izkazali bolje kot tisti, ki izvajajo primerjave od leve proti desni. Nalogo bi lahko nadaljevali z implementacijo preostalih še neimplementiranih algoritmov. Vse algoritme bi nato preizkusili z veliko več vzorci različnih dolžin in s tem dobili algoritem, ki bi na prej omenjenem besedilu najbolje deloval. Če bi želeli dobiti na splošno najboljši algoritem, bi morali take preizkuse izvesti na različnih besedilih, kjer bi spremenili dolžino abecede in frekvenco posameznega znaka v besedilu.

Literatura

[1] MORRIS (Jr) J.H., PRATT V.R., 1970, A linear pattern-matching algorithm, Technical Report 40, University of California, Berkeley.

[2] KNUTH D.E., MORRIS (Jr) J.H., PRATT V.R., 1977, Fast pattern matching in strings, SIAM Journal on Computing 6(1):323-350.

[3] KARP R.M., RABIN M.O., 1987, Efficient randomized pattern-matching algorithms. IBM J. Res. Dev. 31(2):249-260.

[4] http://printkjv.ifbweb.com/

[5] Aho, Alfred V., Margaret J. Corasick (June 1975). Efficient string matching: An aid to bibliographic search. Communications of the ACM 18 (6): 333-340.

[6] Commentz-Walter, B. A String Matching Algorithm Fast on the AverageŠcientific Center Heidelberg, Techical Report, in print.

[7] BOYER R.S., MOORE J.S., 1977, A fast string searching algorithm. Communications of the ACM. 20:762-772.

[8] CROCHEMORE, M., HANCART, C., 1997. Automata for Matching Patterns, in Handbook of Formal Languages, Volume 2, Linear Modeling: Background and Application, G. Rozenberg and A. Salomaa ed., Chapter 9, pp 399-462, Springer-Verlag, Berlin.

[9] SIMON I., 1993, String matching algorithms and automata, in in Proceedings of 1st American Workshop on String Processing, R.A. Baeza-Yates and N. Ziviani ed., pp 151-157, Universidade Federal de Minas Gerais, Brazil.

[10] COLUSSI L., 1991, Correctness and efficiency of the pattern matching algorithms, Information and Computation 95(2):225-251.

[11] COLUSSI L., 1994, Fastest pattern matching in strings, Journal of Algorithms. 16(2):163-189.

[12] HORSPOOL R.N., 1980, Practical fast searching in strings, Software - Practice and Experience, 10(6):501-506.

[13] RAITA T., 1992, Tuning the Boyer-Moore-Horspool string searching algorithm, Software - Practice and Experience, 22(10):879-884.

[14] GALIL Z., GIANCARLO R., 1992, On the exact complexity of string matching: upper bounds, SIAM Journal on Computing, 21(3):407-437.

[15] BAEZA-YATES, R.A., GONNET, G.H., 1992, A new approach to text searching, Communications of the ACM . 35(10):74-82.

[16] APOSTOLICO A., GIANCARLO R., 1986, The Boyer-Moore-Galil string searching strategies revisited, SIAM Journal on Computing 15(1):98-105.

[17] HUME A. and SUNDAY D.M. , 1991. Fast string searching. Software - Practice and Experience 21(11):1221-1248.

[18] ZHU R.F., TAKAOKA T., 1987, On improving the average case of the Boyer-Moore string matching algorithm, Journal of Information Processing 10(3):173-177.

[19] SUNDAY D.M., 1990, A very fast substring search algorithm, Communications of the ACM . 33(8):132-142.

[20] LECROQ T., 1992, A variation on the Boyer-Moore algorithm, Theoretical Computer Science 92(1):119–144.

[21] CROCHEMORE, M., CZUMAJ, A., GASIENIEC, L., JAROMINEK, S., LECROQ, T., PLANDOWSKI, W., RYTTER, W., 1994, Speeding up two string matching algorithms, Algorithmica 12(4/5):247-267.

[22] CROCHEMORE, M., RYTTER, W., 1994, Text Algorithms, Oxford University Press.

[23] NAVARRO G., RAFFINOT M., 1998. A Bit-Parallel Approach to Suffix Automata: Fast Extended String Matching, In Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science 1448, Springer-Verlag, Berlin, 14-31.

[24] ALLAUZEN C., CROCHEMORE M., RAFFINOT M., 1999, Factor oracle: a new structure for pattern matching, in Proceedings of SOFSEM'99, Theory and Practice of Informatics, J. Pavelka, G. Tel and M. Bartosek ed., Milovy, Czech Republic, Lecture Notes in Computer Science 1725, pp 291-306, Springer-Verlag, Berlin.

[25] GALIL Z., SEIFERAS J., 1983, Time-space optimal string matching, Journal of Computer and System Science 26(3):280-294.

[26] CROCHEMORE M., 1992, String-matching on ordered alphabets, Theoretical Computer Science 92(1):33-47.

[27] CROCHEMORE M., PERRIN D., 1991, Two-way string-matching, Journal of the ACM 38(3):651-675.

[28] CHARRAS C., LECROQ T., PEHOUSHEK J.D., 1998, A very fast string matching algorithm for small alphabets and long patterns, in Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching , M. Farach-Colton ed., Piscataway, New Jersey, Lecture Notes in Computer Science 1448, pp 55-64, Springer-Verlag, Berlin.

[29] BERRY, T., RAVINDRAN, S., 1999, A fast string matching algorithm and experimental results, in Proceedings of the Prague Stringology Club Workshop 99, J. Holub and M. Simanek ed., Collaborative Report DC-99-05, Czech Technical University, Prague, Czech Republic, 1999, pp 16-26.

[30] SMITH P.D., 1991, Experiments with a very fast substring search algorithm, Software - Practice and Experience 21(10):1065-1074.

[31] APOSTOLICO A., CROCHEMORE M., 1991, Optimal canonization of all substrings of a string, Information and Computation 95(1):76-95.