UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Anže Pratnemer

Analiza algoritmov za iskanje podnizov s pomočjo sistema ALGator

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2014

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Anže Pratnemer

Analiza algoritmov za iskanje podnizov s pomočjo sistema ALGator

DIPLOMSKO DELO

VISOKOŠOLSKI STROKOVNI ŠTUDIJSKI PROGRAM PRVE STOPNJE RAČUNALNIŠTVO IN INFORMATIKA

MENTOR: doc. dr. Tomaž Dobravec

Ljubljana, 2014

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Analiza algoritmov za iskanje podnizov s pomočjo sistema ALGator

Tematika naloge:

Iskanje podnizov danega niza je ena od osnovnih računalniških operacij, ki med drugim omogoča hitro delovanje internetnih iskalnih orodij. V diplomskem delu preglejte področje iskanja podnizov v danem nizu. Poiščite najbolj znane in najpogosteje uporabljene algoritme s tega področja. Izbrane algoritme implementirajte v sistemu ALGator in jih poženite na standardnih množicah testnih podatkov. Dobljene rezultate časovne zahtevnosti analizirajte ter prikažite v grafični in tabelarični obliki. Algoritme razvrstite po kakovosti glede na hitrost izvajanja na različnih testnih množicah.

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Anže Pratnemer, z vpisno številko **63030273**, sem avtor diplomskega dela z naslovom:

Analiza algoritmov za iskanje podnizov s pomočjo sistema ALGator

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) in ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 25. maja 2014 Podpis avtorja:

Pričujoče diplomsko delo ne bi nastalo brez spodbude in neomajne podpore mojih staršev v mojih najtežjih časih. Iz srca se jima zahvaljujem za ves čas in trud, ki sta mi ju namenila. Zahvala gre tudi mentorju doc. dr. Tomažu Dobravcu, ki me je vzel pod okrilje, kljub zelo omejenim časovnim okvirjem, ter mi z nasveti in posvetovanji pomagal, da sem nalogo naredil kakovostno in brez večjih težav.

Moji dragi Janji.

Kazalo

Povzetek

Iskanje vzorcev v besedilih je zelo pomembno opravilo v veliko vedah. S pomočjo računalniških programov je ta postopek hiter in učinkovit, vendar so za to potrebni optimizirani algoritmi oziroma postopki. V tem delu obravnavamo deset različnih algoritmov, med katerimi ima vsak svoje lastnosti in uporabnosti, od zahtevnosti implementacije, iskalnega časa do odvisnosti od porabe sistemskih zmogljivosti oziroma prostora. Zaradi želje po prikazu delovanja algoritmov v praksi je v delu prikazano iskanje v naprej določenih vzorcih, kot tudi čisto naključnih vzorcih v različno strukturiranih besedilih. Analiza je sestavljena iz primerjav najkrajšega iskalnega časa in povprečnega iskalnega časa, pri opisih algoritmov pa je povzeta še prostorska zahtevnost za vsak algoritem posebej. Algoritmi so implementirani in analizirani s pomočjo okolja za analizo algoritmov ALGator, razvitega z strani doc. dr. Tomaža Dobravca. Rezultati so primerjani na podlagi teoretičnih pričakovanj.

Ključne besede: vzorec, iskanje v nizu, Bruteforce, Boyer-Moore, Turbo-Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt, Horsepool, Berry-Ravindran, Apostolico-Chrochemore, Quick Search

Abstract

Searching for patterns in texts is a very important task in numerous scientific fields. Using computer programs makes the procedure fast and efficient, however, it requires optimised algorithms or procedures. In this thesis we look at ten different algorithms with different characteristics and applications, from level of difficulty of implementation and search time to dependency on system capacity or storage usage. Because we wish to illustrate the practical operation of algorithms, we show how these algorithms search for specific patterns set in advance, as well as completely random patterns in variously structured texts. Analysis includes comparing the shortest search time and average search time, and we also present the storage usage of each individual algorithm. Algorithms are implemented and analysed using an algorithm analysis environment ALGator, developed by doc. dr. Tomaž Dobravc. Results are compared on the basis of theoretical expectations.

Keywords: pattern, search in a set, brute-force, Boyer-Moore, Turbo-Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt, Horsepool, Berry-Ravindran, Apostolico-Chrochemore, Quick Search

Poglavje 1 Uvod

V računalništvu je iskanje oziroma primerjava posameznih vzorcev oziroma zaporedij znakov v daljših besedilih ali nizih znakov zelo pomembno opravilo, ki pa ga je treba izvajati optimizirano in z porabo čim manj sistemskih sredstev. Iskanje vzorcev je primarno opravilo v skoraj vseh oblikah programske opreme v večini operacijskih sistemov. Še več, problem iskanja vzorcev predstavlja nove izzive razvoja algoritmov, tako v programskih kot strojnih okoljih.

Čeprav si podatke shranjujemo v različnih oblikah in na različnih medijih, so črke in besedilo še vedno glavna oblika izmenjave informacij. To je, recimo, vidno v lingvističnih vedah ali literaturi, kjer se informacije prenašajo oziroma iščejo z uporabo slovarjev ali dolgih besedil. Isto analogijo lahko izpeljemo iz računalniških ved, kjer se velike količine informacij po navadi nahajajo v navadnih zaporednih datotekah, vendar je iskanje krajšega vzorca neoptimalno, če bi morali tako datoteko prebrati od začetka do konca, znak po znak. Tudi na področju biologije, kjer zopet vidimo, da se informacija o DNK-ju skriva v zelo dolgem zaporedju črk, ki predstavljajo aminokisline, vidimo, da je iskanje vzorcev nepogrešljivo opravilo pri analizi DNK-ja.

Raziskave kažejo, da se količine teh podatkov podvojijo vsakih osemnajst mesecev, kar pomeni, da morajo biti postopki iskanja določenih zaporedij v večjih količinah informacij učinkoviti in dobro načrtovani ter posodobljeni v okviru napredka programske in strojne opreme.

V delu sem primerjal posamezne algoritme in njihovo učinkovitost na različnih strukturah besedil. Pomagal sem si s sistemom ALGator [1], ki omogoča implementacijo več algoritmov in skupno časovno analizo rezultatov iskanj z različnimi testi. Z implementacijo algoritmov v okolju ALGator in načrtovanjem logičnih testov sem prišel do ocen zmogljivosti iskanja različnih dolžin vzorcev v različnih besedilih.

Poglavje 2 Predstavitev problema

V tem delu se bom osredotočil na iskanje zaporedja znakov v daljših besedilih: Biblija v angleškem jeziku (približno 4.000.000 znakov s presledki), opis gena E.coli bakterije (približno 4.600.000 znakov s presledki), različna naključna zaporedja znakov (približno 5.100.000 znakov s presledki), projekt Gutenberg – 2010 CIA World Factbook (približno 12.500.000 znakov s presledki) in štiri vrste proteinov.

Sam postopek iskanja niza zahteva od nas določitev dveh vhodnih spremenljivk: iskani vzorec (ang. pattern) in besedilo, nad katerim bomo iskanje izvajali (ang. text). Obe spremenljivki sta določeni na končni množici znakov, ki jo bom poimenoval abeceda, označil pa z $\sum$, ki je velikosti σ. Prva spremenljivka ne sme biti daljša od druge, saj v tem primeru vsi algoritmi že na začetku prekinejo izvajanje in se sama logika algoritmov ne sproži, torej tudi ne moremo meriti časovne in prostorske zahtevnosti.

Kot rezultat bomo dobili pozitivno število odmika od začetka besedila, kjer se začne naš vzorec ali vrednost -1, če tak vzorec v našem besedilu ne obstaja. Za enkrat sem se osredotočil samo na iskanje prve pojavitve vzorca, medtem ko so vsi algoritmi zasnovani tako, da je možno dodati funkcionalnost iskanja vseh pojavitev vzorca.

Iskanje vzorca ne poteka pri vseh algoritmih enako. Najbolj osnovni algoritem je tako iskanje s silo oziroma angleško bruteforce, medtem ko lahko algoritmi uporabljajo tudi druge strukture za predstavitev vzorca ali besedila, kot so zgoščevalne funkcije in TRIE, drevesa.

Iskalne algoritme ločimo tudi na podlagi zaporedja vhodnih spremenljivk. Pri podanem vzorcu se le-ta najprej ustrezno obdela (zgoščevalne funkcije, ...) in se nato išče po besedilu, medtem ko se pri najprej podanem besedilu to besedilo najprej obdela (drevesna struktura besed, TRIE, ...). V svojem diplomskem delu se bom osredotočil na primere, ko najprej podamo vzorec in nato besedilo.

Obstajajo postopki iskanja vzorcev z leve proti desni, kot so npr.: Karp-Rabin algoritem, Shift-Or, Knuth-Morris-Pratt ali pa Apostolico-Crochemore in obratni algoritmi, ki delujejo z desne proti levi, kot so recimo: Boyer-Moore, Berry-Ravindran idr. V praksi se izkaže, da so algoritmi, ki uporabljajo postopek iskanja z desne proti levi, najboljši. Poznamo še algoritme, pri katerih smer iskanja ni pomembna, kot je, recimo, prej omenjeni bruteforce ali Horspool.

Najboljše teoretične časovne zahtevnosti pa dobimo iz algoritmov, pri čemer se smer iskanja spreminja oziroma je v določenem zaporedju.

Postopek iskanja v implementiranih algoritmih bo z uporabo metode drsečega okna, ki je po navadi velikosti M. Z uporabo poizkusov se to okno premika po besedilu in primerja znake z vzorcem. Če se znaka na istem indeksu ne ujemata, se okno premakne za določeno pozicijo v levo ali desno, dokler okno ne prestopi konca oziroma začetka besedila ali pa smo v oknu našli iskani vzorec.

V diplomskem delu bom uporabljal sledeča poimenovanja:

- m dolžina vzorca x
- $x=x[0..m-1]$ struktura tabele znakov vzorca, ki je dolžine m
- n dolžina besedila y, v katerem iščemo vzorec x
- $y=y[0..n-1]$ struktura tabele znakov besedila, ki je dolžine n
- O asimptotski simbol (zgornja meja) za ocenjevanje časovne in prostorske zahtevnosti algoritma
- $|x|$ dolžina tabele znakov x
- σ dolžina abecede $\sum$ (število različnih znakov, ki se lahko pojavijo v našem besedilu)

Poglavje 3 Opisi algoritmov

Vsi algoritmi se zaženejo na isti način. V konstruktor razreda podamo vzorec, nato kličemo metodo search, ki ima za parameter besedilo, v katerem bomo iskali vzorec. Nekateri algoritmi nato najprej pripravijo vzorec in nato izvedejo iskanje po besedilu, ostali pa uporabijo kar podani vzorec brez pripravljalne obdelave. Razen Bruteforce algoritma so opisi algoritmov v tem poglavju povzeti po [2, 3].

3.1 Bruteforce

Značilnosti algoritma:

- Ni pripravljalne faze.
- Potrebujemo konstanten dodaten prostor.
- Vedno zamikamo drseče okno samo za eno pozicijo v desno ali levo.
- Primerjamo lahko v obe smeri.
- Iskalna faza ima *O(mn)* časovne zahtevnosti.
- Pričakovano število primerjav znakov je *2*n*.

Algoritem bere znake v besedilu z leve proti desni. Če se znak ujema s prvim znakom v vzorcu, se števca pozicije primerjanega znaka povečata za eno. Če se števec pozicije znaka za primerjavo v vzorcu ujema z dolžino vzorca, potem imamo ujemanje celotnega vzorca. Če pa naletimo na neujemanje para znakov, se števec pozicije znaka v vzorcu ponastavi na prvi znak in primerjava se zopet začne izvajati, vse dokler števec pozicije znaka v besedilu ne doseže velikosti *n-m*.

Ta algoritem deluje tudi, če bi želeli izvajati primerjave z desne proti levi.

Zaradi preprostosti in kratkosti bom pokazal kar celotno kodo algoritma in jo komentiral v vrsticah:

```
n = dolžina besedila txt
m = dolžina vzorca pat

for(i = 0; i < n-m+1; i++ ){
     j = 0;                    //ponastavi j, da kaže na prvi
                               znak v vzorcu
     while(j < m && txt.charAt(i+j) == pat.charAt(j)){
     //dokler je j manjši od dolžine vzorca in znaka na indeksu
       i+j v besedilu ter j v vzorcu, se ujemata
          j++;
          if(j == m) return i; //če je j enak dolžini vzorca,
                                    imamo ujemanje
    }

}
return -1;                          //ujemanja ni
```

Implementiral sem tudi algoritem BruteForce2, ki deluje na isti način, kot je opisano zgoraj, vendar uporablja malce drugačno sintakso jezika Java in se na koncu izkaže za bolj optimalnega.

3.2 Boyer-Moore

Značilnosti algoritma:

- Iskanje poteka z desne proti levi.
- Pripravljalna faza ima $O(m+\sigma)$ časovno in prostorsko zahtevnost.
- Iskalna faza ima $O(mn)$ časovno zahtevnost.
- Ko iščemo vzorec brez ponavljanj, imamo v najslabšem primeru $3*n$ primerjav znakov.
- V najboljšem primeru imamo $O(n/m)$ časovno zahtevnost.

Boyer-Moore algoritem je najbolj učinkovit algoritem za iskanje vzorcev v tekstih. Algoritem je po navadi uporabljen v tekstovnih urejevalnikih v funkcijah iskanja in zamenjave vzorcev.

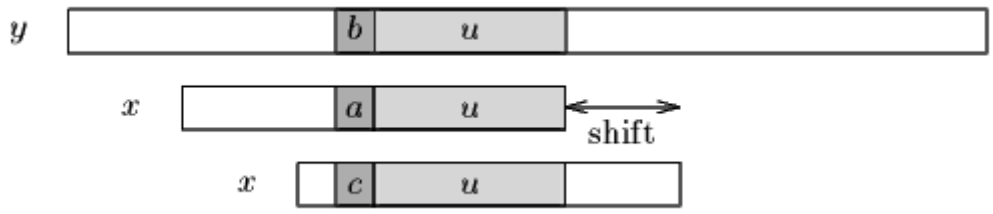
Algoritem preverja znake vzorca z desne proti levi. V primeru neujemanja (ali popolnega ujemanja, če iščemo več pojavitev vzorca v besedilu) uporabi dve funkciji, ki drseče okno

prestavita v desno. Funkciji imenujemo zamik dobre pripone (*ang. good-suffix shift* ali *matching shift*) in zamik slabega znaka (ang. *bad-charachter shift* ali *occurrence shift*).

Predvidevajmo, da se neujemanje zgodi med znakoma vzorca $x[i]=a$ in besedila $y[i+j]=b$ na poziciji j.

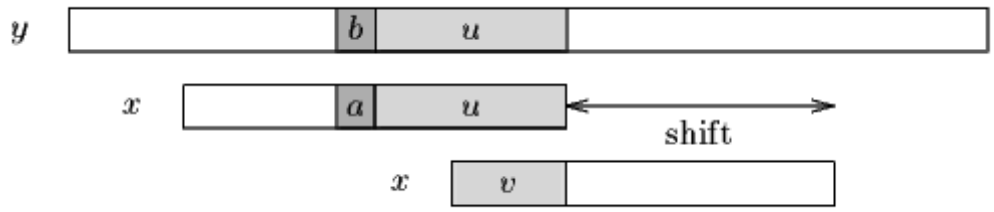
Potem imamo naslednje pogoje*:* $x[i+1..m-1] = y[i+j+1..j+m-1]=u$ *in* $x[i] \neq y[i+j]$.

Zamik dobre pripone nam nato predstavlja poravnavo segmenta $y[i+j+1 .. j+m-1] = x[i+1.. m-1]$ z najbolj desno pojavitvijo v vzorcu, ki ima znak pred tem segmentom različen od $x[i]$ (Slika 1).



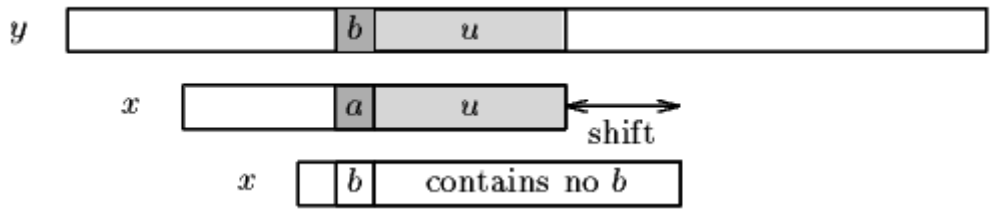
Slika 1: Zamik dobre pripone: preostanek vzorca u se ponovno pojavi, vendar ima vzorec drugačen znak na začetku

Če takega segmenta v vzorcu ni, potem se zamik zgodi s poravnavo najdaljše pripone *v* v *y[i+j+1 .. j+m-1]* z ujemajočo se predpono v vzorcu *x* (Slika 2).



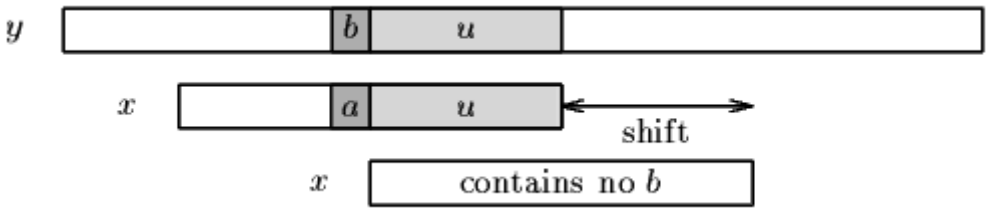
Slika 2: Zamik dobre pripone: samo pripona u se pojavi v vzorcu x

Zamik slabega znaka je sestavljen iz poravnave znaka iz besedila na indeksu *y[i+j]* z najbolj desno pojavitvijo v vzorcu *x[0 .. m-2]* (Slika 3).



Slika 3: Zamika slabega znaka: znak a se pojavi v vzorcu *x*

Če se znak v besedilu na indeksu *y[i+j]* ne pojavi v vzorcu *x*, potem vzorec *x* ne more vsebovati znaka na indeksu *y[i+j]*, zato je levi del vzorca poravnan na naslednji znak, ki bi ga že lahko vseboval, torej na indeks *y[i+j+1]* (Slika 4).



Slika 4: Zamik slabega znaka: znak *b* se ne pojavi v vzorcu *x*

Zamik slabega znaka je lahko tudi negativen, zato pri zamikanju drsečega okna algoritem zamikamo za večjega izmed vrednosti zamika dobre pripone in zamika slabega znaka.

Za obe funkciji lahko bolj formalno rečemo naslednje:

funkcija zamika dobrega znaka je shranjena v tabeli *bmGs* z velikostjo *m-1*.

Imamo dva pogoja:

1. *Cs(i, s):* za vsak *k*, da je $i < k < m$, $s \geq k$ ali $x[k-s]=x[k]$ in
2. *Co(i, s):* če je $s < i$, potem $x[i-s] \neq x[i]$

Potem za $0 \leq i < m$: $bmGs[i+1] = min\{s>0 : Cs(i,s)$ in $Co(i, s)\}$, opredelimo $bmGs[0]$ kot dolžino ponovitve vzorca *x*. Za izračun tabele *bmGs* uporabimo tabelo *suff*, ki je določena tako: za $1 \leq i < m$, $suff[i] = max\{k: x[i+k+1 .. i], x[m-k .. m+1]\}$.

Tabela *bmBc*, ki nam predstavlja izračun funkcije zamika slabega znaka, je velikosti σ. Znak *c* iz abecede $\sum$: $bmBc[c] = min\{i: 1 \leq m-1, x[m+1+i]=c\}$, če se *c* pojavi v vzorcu *x*, drugače uporabimo *m*.

Tabeli *bmBc* in *bmGc* lahko izračunamo v naprej v času $O(m+\sigma)$ pred samo iskalno fazo, za njiju pa potrebujemo $O(m+\sigma)$ dodatnega prostora. Časovna zahtevnost iskalne faze je kvadratna, vendar naredimo največ *3*n* primerjav znakov, če iščemo vzorec, ki nima ponavljanja. Na velikih abecedah (glede na dolžino vzorca) je algoritem izjemno hiter. Ko iščemo vzorec, v katerem se ponavlja znak *a* *m-1*-krat in mu sledi še besedilo, sestavljeno samo iz enega znaka (za primer: imamo vzorec *xxxxy*, ki ga želimo najti v besedilu *yyyyyy..yyy*), formalni zapis: vzorec: *a^(m-1)*b*; besedilo: *b^n* algoritem, naredi samo $O(n/m)$ primerjav, kar je najmanjša možna vrednost v modelu algoritmov, kjer pripravimo samo vzorec.

3.3 Turbo-Boyer-Moore

Značilnosti algoritma:

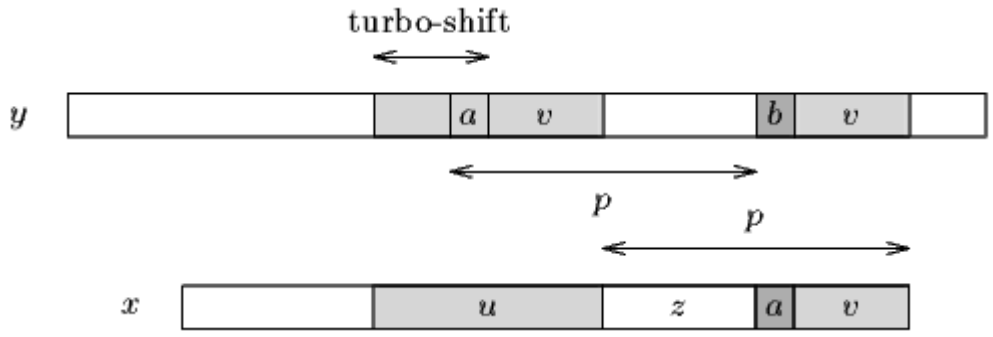
- Različica Boyer-Moore algoritma.
- Pripravljalna faza ima $O(m+\sigma)$ časovno in prostorsko zahtevnost.
- Iskalna faza ima $O(n)$ časovno zahtevnost.
- $2n$ primerjav znakov v najslabšem primeru.

Turbo-BM algoritem je izboljšava navadnega Boyer-Moore algoritma. Algoritem se izboljša tako, da si zapomnimo pozicijo besedila, ki se je ujemala priponi vzorca v naši zadnji primerjavi (in samo, če smo izvedli zamik dobrega znaka).

Ta tehnika ima dve prednosti:

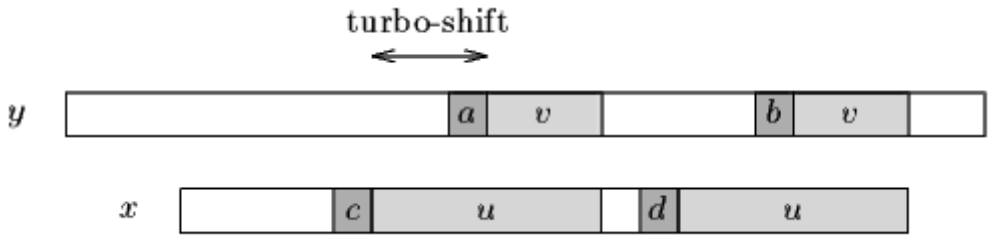
- omogoča preskok čez to pozicijo in
- nam omogoča hitri zamik.

Hitri zamik se lahko zgodi, če je znotraj trenutne primerjave pripona vzorca, ki se ujema z besedilom, krajša kot pripona, ki smo si jo zapomnili v prejšnji primerjavi. Naj bo u pozicija, ki smo si jo zapomnili in v pripona, ki se ujema v trenutni primerjavi, taka, da je uzv pripona x-a. Naj bosta a (znak v vzorcu) in b (znak v besedilu) znaka, ki povzročita neujemanje v našem poizkusu primerjave vzorca in besedila. Potem je av pripona x-a in ravno tako u-ja, ker velja $|v| < |u|$. Znaka a in b se pojavita v razmiku p znakov v besedilu in pripona x-a dolžine $|uzv|$ ima periodo dolžine $p=|zv|$, ker je u meja niza uzv, zato ne more prekrivati obeh pojavitev znakov a in b na razdalji p v besedilu. Najmanjši mogoči zamik ima dolžino $|u|-|v|$, kar imenujemo hitri zamik (Slika 5).



Slika 5: Hitri zamik: mogoč samo takrat, kadar $|v|<|u|$

Vseeno pa v primeru $|v| < |u|$, če imamo dolžino zamika slabega znaka večjo od dolžine zamika dobre pripone in dolžine hitrega zamika, potem mora biti dolžina izvedenega zamika večja ali enaka $|u|+1$. V tem primeru sta oba znaka, c in d, različna, ker smo predpostavili, da je bil prejšnji zamik zamik dobre pripone. (Slika 6)



Slika 6: $c \neq d$ zato ne moreta biti poravnana z istim znakom v priponi v

Potem bi zamik, ki je večji od hitrega zamika, ampak manjši od $|u|+1$, poravnal c in d z istim znakom v v, zato mora biti dolžina izvedenega zamika enaka $|u|+1$ ali večja.

3.4 Horspool

Značilnosti algoritma:

- poenostavitev Boyer-Moore algoritma,
- lahka implementacija,
- pripravljalna faza ima $O(m+\sigma)$ časovne zahtevnosti in $O(\sigma)$ prostorske zahtevnosti,
- iskalna faza ima $O(mn)$ časovne zahtevnosti in
- povprečno število primerjav znakov je med $1/\sigma$ in $2/(\sigma+1)$.

Funkcija zamika slabega znaka v Boyer-Moore algoritmu in njegovih izvedenkah ni učinkovita na majhnih abecedah, medtem ko je zelo uporabna v primerih, ko imamo za abecedo več znakov, kot je recimo ASCII kodiranje.

Horsepool algoritem uporabi samo funkcijo zamika slabega znaka na zadnjem – desnem znaku drsečega okna za izračun zamikov.

Iskalna faza algoritma ima kvadratno zahtevnost primerjav v najslabšem primeru, vendar se lahko dokaže, da je povprečno število primerjav znaka med $1/\sigma$ in $2/(\sigma+1)$.

Psevdokoda algoritma:

```
Horspool(P,T)
          //pripravljalna faza
          For c ∈ ∑  do d[c] ← m
          For j ∈ 1…m-1 do d[p_j] ← m - j

          //iskalna faza
          pos ← 0
          While pos ≤ n-m do
               j ←m
               While j > 0 And t_pos+j == p_j  do j ← j-1
               If j == 0: pattern found
               pos ← pos +d[t_pos+m]
          End of while
```

3.5 Knuth-Moriss-Pratt

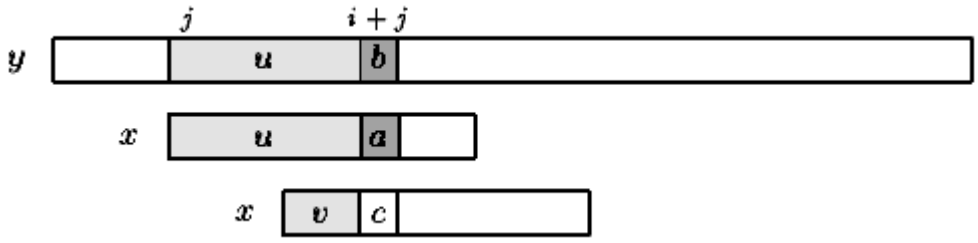
Značilnosti algoritma:

- Iskanje poteka z leve proti desni.
- Pripravljalna faza ima *O(m)* časovne in prostorske zahtevnosti.
- Iskalna faza ima *O(n-m)* časovne zahtevnosti v odvisnosti od velikosti naše abecede ∑.

Poglejmo si primer, ko imamo okno na poziciji j, torej gledamo znake besedila *y[j .. j+m-1]*. Predpostavimo, da se prvo neujemanje zgodi med *x[i]* in *y[i+j]*, ko je *0<i<m*. Iz tega sledi, da je *x[0 .. i-1] = y[j .. i+j-1] =u* in *a = x[i] ≠ y[i+j]=b*.

Ko drseče okno zamikamo, lahko pričakujemo, da se predpona *v* v vzorcu *x* ujema s pripono dela besedila *u*. Opazimo še, da če se želimo izogniti ponovnemu neujemanju, mora biti znak, ki sledi predponi *v* v vzorcu *x*, različen od znaka *a*. Taki najdaljši predponi *v* rečemo označilna meja *u-ja* (pojavi se na obeh koncih *u-ja*, katerima sledi drugačen znak v vzorcu *x*) (Slika 7).

S tem v mislih lahko napovemo naslednje: naj bo *kmpNext[i]* dolžina najdaljše take meje *x[0.. i-1]*, kateri sledi znak *c*, ki je različen od *x[i]* ali pa -1, če taka meja ne obstaja, za vsak *i* med 0 in *m*. Po zamiku se primerjava lahko nadaljuje med znakoma *x[kmpNext[i]]* in *y[i+j]*, brez, da bi zgrešili kakšno pojavitev vzorca *x* v besedilu *y*, in se izognemo vračanju v besedilu. Začetna vrednost *kmpNext[0]* je nastavljena na -1.



Slika 7: Zamik v Knuth-Morris-Pratt algoritmu (*v* predstavlja mejo *u*-ja in *c ≠b*)

Psevdokoda iskalnega dela algoritma, povzeta po [4]:

```
algorithm kmp_search:
    input:
        an array of characters, S (the text to be searched)
        an array of characters, W (the word sought)
    output:
        an integer (the zero-based position in S at which W is
found)

    define variables:
        an integer, m ← 0 (the beginning of the current match
in S)
        an integer, i ← 0 (the position of the current
character in W)
        an array of integers, T (the table, computed
elsewhere)

    while m + i < length(S) do
        if W[i] = S[m + i] then
            if i = length(W) - 1 then
                return m
            let i ← i + 1
        else
            if T[i] > -1 then
                let m ← m + i - T[i], i ← T[i]
            else
                let i ← 0, m ← m + 1

    (if we reach here, we have searched all of S
unsuccessfully)
    return the length of S
```

KMP algoritem opravi največ *2n-1* primerjav znakov v iskalni fazi. Izraz zamuda (ang. *delay*) nam opisuje največje število primerjav za posamezni znak in je omejen z $\log\Phi(m)$, kjer nam Φ predstavlja zlati rez ($\Phi = (1+\sqrt{5})/2$).

3.6 Apostolico-Crochemore

Značilnosti algoritma:

- pripravljalna faza v *O(m)* časovne in prostorske zahtevnosti,
- iskalna faza v *O(n)* časovne zahtevnosti,
- v najslabšem primeru izvede *3/2n* primerjav znakov.

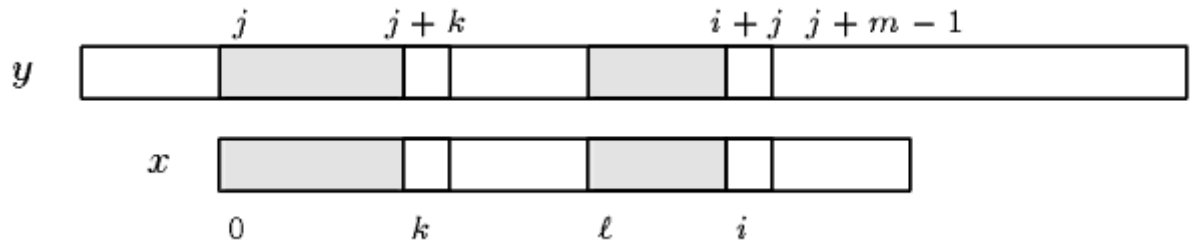
Apostolico-Crochemore algoritem uporablja tabelo zamikov, ki jo oblikuje Knuth-Morris-Pratt algoritem (funkcija *preKmp*) za izračun svojih zamikov.

Naj bo $\ell=0$, če je vzorec x sestavljen samo iz m ponovitev enega znaka c *(x=c^m, kjer je c v* $\sum$*)*, drugače naj bo ℓ enak poziciji prvega znaka v vzorcu x, ki je različna od *x[0]*. Primerjave so narejene s pozicijami v vzorcu v naslednjem zaporedju: $\ell, \ell+1, \ldots, m-2, m-1, 0, 1, \ldots, \ell-1$.

Skozi iskalno fazo imamo tri spremenljivke *(i, j, k)* (Slika 8), kjer:

- Drseče okno je postavljeno na *y[j .. j+m-1]* v besedilu;
- $0 \le k \le \ell$ in *x[0 .. k-1]=y[j .. j+k-1];*
- $\ell \le i < m$ in *x[ℓ .. i-1]=y[j+ℓ .. i+j-1]*.

Začetne vrednosti spremenljivk so nastavljene na $(\ell, 0, 0)$.



Slika 8: Pri vsaki primerjavi preverimo vse tri spremenljivke *i, j* in *k*

Naslednjo trojico spremenljivk (*i, j, k*) izračunamo po naslednjih postopkih glede na vrednost spremenljivke *i*:

- $i = \ell$
  Če je $x[i] = y[i+j]$, potem je naslednja trojica spremenljivk $(i+1, j, k)$. Če je $x[i] \neq y[i+j]$, potem je naslednja trojica spremenljivk $(\ell, j+1, max\{0, k-1\})$.

- $\ell < i < m$
  Če je $x[i] = y[i+j]$, potem je naslednja trojica spremenljivk $(i+1, j, k)$. Če je $x[i] \neq y[i+j]$, potem imamo zopet dva primera, ki sta odvisna od vrednosti *kmpNext[i]:*
  - $kmpNext[i] \leq \ell$: potem je naslednja trojica spremenljivk $(\ell, i+j-kmpNext[i], max\{0, kmpNext[i]\})$
  - $kmpNext[i] > \ell$: potem je naslednja trojica spremenljivk $(kmpNext[i], i+j-kmpNext[i], \ell)$.

- $i = m$
  Če je $k < \ell$ in $x[k]=y[j+k]$, potem je naslednja trojica spremenljivk $(i, j, k+1)$. Drugače imamo $k < \ell$ in $x[k] \neq y[j+k]$ ali pa $k=\ell$.

Če imamo $k=\ell$, smo našli naš vzorec v besedilu. V vsakem primeru bi se naslednja trojica spremenljivk izračunala po istem postopku, kot če imamo $\ell < i < m$.

Pripravljalna faza je sestavljena iz izračuna tabele *kmpNext* s pomočjo funkcije *preKmp* in števila ℓ. Povprečni čas te faze je *O(m)* časovne in prostorske zahtevnosti.

Iskalna faza ima *O(n)* časovno zahtevnost. Celoten algoritem naredi največ *3/2*n* primerjav znakov.

3.7 Berry-Ravindran

Značilnosti algoritma:

- iskanje poteka z desne proti levi;
- Različica Boyer-Moore algoritma;
- mešanica algoritma Quick Search in Zhu and Takaoka;
- pripravljalna faza ima *O(m+σ^2)* časovne in prostorske zahtevnosti;
- iskalna faza ima *O(mn)* časovne zahtevnosti.

Avtorja Berry in Ravindran sta načrtovala algoritem, ki opravlja zamike drsnega okna z upoštevanjem funkcionalnosti Boyer-Moore-ovega algoritma zamika slabega znaka. Za razliko originalne funkcionalnosti sta se avtorja odločila, da bosta preverjala dva znaka desno od drsečega okna.

Pripravljalna faza algoritma je sestavljena iz izračuna pojavitve znakov (*a, b v* ∑) na desni strani vzorca *axb*.

Za vsak par (*a, b*) potem velja (SKLIC):

$$brBc[a,b] = \min \begin{cases} 1 & \text{if } x[m-1] = a \ , \\ m-i+1 & \text{if } x[i]x[i+1] = ab \ , \\ m+1 & \text{if } x[0] = b \ , \\ m+2 & \text{otherwise} \ . \end{cases}$$

Po primerjavi, kjer imamo drseče okno nastavljeno na poziciji besedila *y[j .. j+m-1]*, naredimo zamik okna dolžine *brBc[y[j+m],y[j+m+1]]*. Znaka v besedilu na indeksih *y[n]* in *y[n+1]* sta nastavljena na *null*, da lahko izračunamo zadnje zamike okna v tem algoritmu.

3.8 Rabin-Karp

Značilnosti algoritma:

- uporablja funkcijo zgoščevanja;
- pripravljalna faza ima *O(m)* časovno in prostorsko zahtevnost;
- iskalna faza ima *O(mn)* časovno zahtevnost;
- pričakovani čas izvajanja je *O(n+m)*.

Zgoščevanje nam predstavlja enostavno metodo, kako se izogniti kvadratnim številkam primerjanj znakov v večini situacij. Namesto, da preverjamo na vsaki poziciji besedila, ali se vzorec pojavi, je bolj učinkovito, če je vsebina našega drsečega okna podobna vzorcu. Za ta namen uporabimo zgoščevalno funkcijo.

Da je zgoščevalna funkcija primerna za delo z nizi, mora imeti naslednje lastnosti:

- enostavno izračunljiva;
- visoko razločevalna za nize znakov;
- vrednost *hash(y[j+1 .. j-m])* mora biti enostavno izračunljiva iz vrednosti *hash(y[j .. j-m+1])* in *y[j+m]: hash(y[j+1 .. j+m])= rehash(y[j], y[j+m], hash(y[j .. h+m-1]))*.

Naj bo za besedo *w* dolžine *m* zgoščevalna funkcija hash opredeljena tako:

*Hash(w[0 .. m-1]) = (w[0]*2m-1+w[1]*2m-2+ .. + w[m-1]*20)mod q*, kjer je *q* veliko število.

Funkcije ponovnega zgoščevanja rehash pa je: *rehash(a,b,h)=((h-a*2m-1)*2-b) mod q*.

Pripravljalna faza je v osnovi izračun hash vrednosti vzorca *hash(x)*. Potrebno je *O(m)* časa in konstanten prostor.

Skozi iskalno fazo je dovolj, če primerjamo zgoščeno vrednost funkcije *hash(x)* z vrednostjo zgoščevanja našega okna *hash(y[j .. j+m-1]) za* $0 \leq j < n-m$. Če najdemo podobno zgoščeno vrednost, moramo vseeno preveriti še znak po znak za popolno ujemanje.

3.9 Quick Search

Značilnosti algoritma:

- poenostavitev Boyer-Moore algoritma;
- uporablja samo zamik slabega znaka;
- lahka implementacija;
- pripravljalna faza ima $O(m+\sigma)$ časovne zahtevnosti in $O(\sigma)$ prostorske zahtevnosti;
- iskalna faza ima $O(mn)$ časovno zahtevnost;
- hiter za kratke vzorce in veliko abecedo.

Quick Search algoritem uporablja Boyer-Moore-ovo funkcionalnost zamika slabega znaka. Po primerjavi, kjer je okno postavljeno na besedilu $y[j..j+m-1]$, je dolžina zamika enaka enemu ali več znakov. Iz tega sledi, da je znak $y[j+m]$ vključen v naslednji poizkus in je zato lahko uporabljen za zamik slabega znaka.

Funkcija zamika slabega znaka je malo spremenjena, da lahko upošteva zadnji znak vzorca x, tako da je: za c v abecedi $\sum$, $qsBc[c]=min\{i: 0 < i \leq m \text{ in } x[m-i] = c\}$, le če se c pojavi v vzorcu x, drugače je vrednost $m+1$.

Skozi iskalno fazo je lahko smer primerjav med vzorcem in besedilom poljubna. Čeprav ima iskalna faza kvadratno zahtevnost v najslabšem primeru, se v praksi obnaša zelo dobro.

Poglavje 4 ALGator

Za analizo rezultatov sem uporabil sistem ALGator [3]. Namen sistema je analiza delovanja različnih algoritmov na določenem problemu in nato primerjava končnih rezultatov, kot so porabljeni čas, povprečni čas izvajanja, število primerjav ipd.

Sistem je neodvisen od platforme, prav tako ni treba nameščati njegovih komponent. Ravno zaradi tega razloga se s sistemom lahko prične delati takoj, ko ga prenesemo s spletne strani. S pripadajočo dokumentacijo in primeri že implementiranih projektov, algoritmov in testnih množic je učenje uporabe ter implementacije novega projekta primerno za vsakogar, ki pozna programski jezik Java.

Sistem nam omogoča dodajanje svojih projektov, v okviru katerih se opredelimo do problema (urejanje, iskanje, množenje matrik ipd.) in nato dodajamo algoritme, ki ta problem rešujejo. Algoritmi so implementirani po določenih pravilih, prav tako tudi testne množice vhodnih podatkov, na katerih naše algoritme izvajamo. Sistem na koncu predstavi rezultate posameznega algoritma in omogoči tudi primerjavo rezultatov med vsemi algoritmi v našem projektu z uporabo grafičnega okolja, kjer izberemo, katere lastnosti vhodnih podatkov in rezultatov nas zanimajo, ter nato vse skupaj predstavi z grafom.

Zagon sistema se za enkrat izvrši preko ukazne vrstice, medtem ko je za analizo rezultatov že narejen grafični vmesnik. Vse nastavitve projekta (imena algoritmov, testi, testne množice vhodnih podatkov) se urejajo preko tekstovnih datotek tipa JSON, prav tako je tudi datotečna struktura map določena. Na voljo imamo nekaj stikal pri samem zagonu sistema, ki določajo, ali se nam izpisujejo trenutne operacije, kateri projekt bomo zagnali, katere algoritme posameznega projekta bomo zagnali ipd.

Rezultati se zapišejo v tekstovne datoteke, v katerih imamo po vrsticah ločene posamezne teste in v vrsticah z dvopičjem ločene izhodne parametre testov, kot so na primer: ime testa, časovne spremenljivke izvedenih testov, podatki o vhodnih podatkih (npr. velikost podatkov za obdelavo, ime vhodnih datotek), število primerjav ali operacij v testih itd.

Nad rezultati je mogoče izvajati tudi nekaj poizvedb, predvsem uporabna so združevanja rezultatov v skupine in urejanja po velikosti ter seštevanje določenih izhodnih podatkov.

Poglavje 5 Implementacija projekta in algoritmov

Ustvarjanje samega projekta je potekalo hitro in brez problemov. S pomočjo že narejenih primerov sortiranja števil in množenja matrik sem ustvaril projekt, napisal vse konfiguracijske datoteke, da je sistem projekt prepoznal, nato sem najprej implementiral dva algoritma, in sicer Bruteforce in Boyer-Moore. Ko sta se oba algoritma uspešno prevedla in zagnala, sem na internetu našel primerne testne datoteke in spisal testne funkcije, ki so testirale algoritme. Na začetku sem se osredotočil na samo pravilnost izvedbe algoritmov in interpretacije rezultatov, nato pa sem teste naredil bolj obsežne in preverjal več parametrov izvajanja algoritmov.

Med samo implementacijo algoritmov sem uporabil funkcionalnost dedovanja med posameznimi razredi projekta, kar mi je precej olajšalo delo, saj je veliko funkcij za delo z algoritmi že implementiranih v samem sistemu, jaz sem jim moral samo dodati funkcionalnost, ki je svojevrstna za razreševanje mojega zastavljenega problema.

Ko sem imel dva delujoča algoritma in sem uspešno razbral rezultate ter jih pravilno interpretiral, sem dodal še osem algoritmov.

Na koncu sem imel implementirane naslednje algoritme:

- Bruteforce, Bruteforce2 (izboljšana različica osnovnega algoritma),
- Boyer-Moore,
- Turbo-Boyer-Moore,
- Rabin-Karp,
- Knuth-Morris-Pratt,
- Horspool,
- Berry-Ravindran,
- Apostolico-Crochemore in
- Quick Search.

Večina primerov kode za algoritme na internetu že obstaja, tako da sem si pomagal z njimi, dodatno pa sem jih moral prilagoditi za izvajanje v sistemu ALGator.

Algoritmi se pri poročanju delovanja povezujejo s sistemom s pomočjo dveh funkcij. V funkciji execute imamo določen sam zagon algoritma, kjer skozi tri vhodne parametre

podamo vzorec in besedilo, v katerem vzorec iščemo, rezultat pa se nam vrne v spremenljivko offset, kjer se zapiše odmik najdenega vzorca v besedilo ali pa vrednost -1, kar predstavlja, da tega vzorca v iskanem besedilu ni.

Drugi način interakcije programske kode s samim sistemom pa je z uporabo merilnih parametrov, kjer v samo kodo algoritma dodamo ukaz *//@COUNT(compare)*, kar sistemu pove, da mora šteti, kolikokrat se vrstica za tem ukazom izvede. Spremenljivka *compare* na koncu predstavlja število primerjav znakov med samim primerjanjem algoritma.

Poglavje 6 Meritve

Rezultate iskanja niza v besedilu lahko razdelimo na tri skupine:

- Uspešnost, ki sem jo meril tako, da sem v funkciji, ki preverja uspešnost algoritmov, preveril, ali ima spremenljivka *offset* pravilno vrednost. Same teste sem zasnoval tako, da na vsakem algoritmu izvedem iskanja, za katere vem, kakšen odmik najdenega vzorca mora biti, če algoritem deluje pravilno. Pri prvem testu sem nastavil, da algoritem išče niz dolžine 8, ki se nahaja čisto na začetku besedila. Kot rezultat sem pri vseh algoritmih dobil vrednost spremenljivke *offset 0*. Drugi test sem zasnoval tako, da mora vrniti odmik, ki je vrednost dolžine besedila minus dolžina vzorca, ki v teh testih znaša osem znakov. Za tretji test sem izbral niz v besedilu, ki se zagotovo pojavi samo enkrat in se nahaja približno na sredini celotnega besedila. Zadnji test pa išče vzorec, ki v besedilu ne obstaja, tako da mora vedno vrniti vrednost *offset -1*.

- Časovni rezultati so zelo pomemben pokazatelj učinkovitosti algoritma. Vsi testi se ponovijo večkrat, trenutna vrednost je nastavljena na dvesto, tako da lahko izničimo vplive zasedenosti procesorja ali morebitne motnje delovanja sistema s strani drugih aplikacij. Merim najkrajši čas izvedbe algoritma, najdaljši čas in povprečni čas, ki ga algoritem porabi za vsak test. S temi spremenljivkami bom lahko kasneje primerjal algoritme med seboj in ocenil njihovo uspešnost.

- Drugi pokazatelj učinkovitosti algoritmov je število primerjav znakov v enem iskanju vsakega algoritma. Tukaj beležimo število primerjav samo enkrat in ne za vsa ponavljanja posameznega testa, saj algoritmi delujejo vedno enako v tem pogledu. Vsaka primerjava porabi nekaj procesorskega časa, tako da stremimo k temu, da algoritem naredi čim manj primerjav z uporabo tehnik preskokov ali pa zgoščevalnih funkcij. Sistem beleži in šteje, kolikokrat se izvede primerjava znaka iz besedila in znaka iz vzorca, ter na koncu vrne to informacijo zapisano v datoteko z rezultatom. Nekateri algoritmi nimajo neposrednih primerjav med posameznimi znaki besedila in vzorca, saj na začetku najprej vzorec pripravijo na svojevrsten način (recimo zgoščevanje) ali pa ne primerjajo znakov, temveč bite znakov (recimo algoritem Shift-Or), tako da se pri teh algoritmih merjenje primerjav znakov ne more izvesti.

Poglavje 7 Testne množice in testi

Vse algoritme je seveda treba testirati na dovolj različnih in velikih množicah podatkov, da lahko vidimo morebitne učinke različnosti podatkov na samo uspešnost algoritmov. Testiranje z različnimi strukturami podatkov je prav tako pomembno iz vidika samega delovanja algoritmov, kajti nekateri algoritmi delujejo bolje, če so recimo vzorci daljši, ker jim to omogoča večji preskok neustreznih podatkov. Nekateri algoritmi se slabše odrežejo pri strukturi vzorca, ki je preveč homogen, torej je dolg, vendar ima zelo malo različnih znakov v sebi.

Za množice sem izbral dve zelo obsežni besedili, Biblijo v angleškem jeziku in besedilo projekta Gutenberg – 2010 CIA World Factbook (krajše CIAWF). Obe besedili predstavljata zelo različna zaporedja znakov, vendar lahko nad njima simuliramo iskanje posameznih besed, kot bi to resnično potrebovali v stvarnem življenju. Biblija ima približno 4.000.000 znakov s presledki, ki jih ravno tako jemljem kot posamezni znak za primerjavo, medtem ko ima drugo besedilo preko 12.500.000 znakov s presledki.

Drugi sklop so zapisi zaporedij znakov, ki večini ljudi ne pomenijo nič, če jih preberejo. Gre se za zapis sekvence proteina človeškega gena in gena bakterije E.Coli, ki imata precej naključno postavitev znakov in štiri datoteke, kjer so generirani čisto naključni znaki. V datoteki rand2 imamo samo dva različna znaka, ki se naključno menjavata, v datoteki so ti različni znaki štirje, imamo datoteko z osmimi različnimi znaki, ki se pojavljajo v naključnem vrstnem redu, datoteko s šestnajstimi znaki in na koncu še datoteko z 32 različnimi znaki. Vse štiri datoteke so iste dolžine s 5.242.880 znaki. Vsa besedila so tudi bila uporabljena pri testiranju podobnega projekta SMART [4] in sem ga zato nameraval uporabiti za primerjalno referenco mojih implementacij algoritmov, vendar avtorji projekta niso več dosegljivi, prav tako niso objavili svojih rezultatov, javna programska koda pa se ne prevede več uspešno.

Teste sem ločil v dve skupini:

- Iskanje naključno izbranega vzorca v različnih dolžinah. Za to skupino sem uporabil vsa besedila. V Bibliji sem iskal naključni vzorec naključne dolžine in dolžine 100 znakov. V CIAWF sem iskal naključne vzorce dolžin: 3, 10, 50, 100, 1.000 in 10.000. Iskanje vzorca dolžine 1.000 sem ponovil trikrat z različnim naključnim vzorcem. Pri ostalih datotekah, ki predstavljajo naključne znake, pa sem naredil teste z naključnim vzorcem dolžine 10, 100 in 1000. V imena testov sem vključil besedo *RND*, kar

pomeni, da bo vzorec izbran naključno, ter dolžino vzorca pri testih, ki uporabljajo Biblijo in CIAWF.

- Druga skupina pa predstavlja teste, ki preverjajo pravilnost algoritmov. Te teste sem izvajal samo na Bibliji, ker je najlažje najti osamljene primere vzorcev, za katere lahko napovemo odmike. Tukaj se izvedejo štirje testi: iskanje vzorca na odmiku 0, iskanje vzorca na sredini besedila, iskanje vzorca, ki se pojavi čisto na koncu besedila, in iskanje vzorca, ki ga v besedilu ni.

Vrstica v datoteki, ki opisuje teste, izgleda takole:

testSETBibleMiddle8:SET:odious w:2178291:testset1/bible.txt.

Posamezni parametri so ločeni z dvopičjem, kar nam na pokazanem primeru poda naslednje parametre (z leve proti desni):

- Ime testa: ime vsebuje besedo *SET*, ki nam takoj pove, da bo ta test delal na v naprej določenem vzorcu datoteke Bible. Beseda Middle pove, da bo iskani vzorec nekje na sredini besedila, in številka pove dolžino vzorca. S tem poimenovanjem lahko takoj razberemo parametre testa.

- Vrsta testa: ločimo med *RND* in *SET* (skupina testov za preverjanje pravilnosti algoritmov) vrednostjo parametra. Pri vrednosti *SET* imamo vzorec ročno določen, nahaja se v naslednjem parametru, medtem ko nam pri *RND* sistem sam določi vzorec, dolžina vzorca pa se nahaja v naslednjem parametru.

- V primeru vrste testa *SET* je ta parameter niz, ki predstavlja iskani vzorec, v primeru vrste testa *RND* pa se v tem parametru nahaja številčna vrednost dolžine naključnega vzorca. Ta parameter je lahko tudi prazen, v tem primeru pa imamo pri vrsti testa *SET* vedno izbran vzorec prvih deset znakov besedila (nekateri algoritmi delujejo od zadaj naprej, tako da uporaba praznega tretjega parametra ni priporočljiva zaradi konsistentnosti rezultatov), pri vrsti testa *RND* pa se dolžina vzorca določi naključno med nič in sto.

- Četrta vrednost v vrstici je uporabljena samo v primeru vrste testa *SET* in nam poda dodatno preverjanje, če je bil vzorec najden na pravilnem odmiku. V ta parameter vpišemo pravilni odmik našega vzorca, ki smo ga ročno določili v tretjem parametru. V primeru, da pustimo ta parameter prazen, bo sistem predpostavil, da smo pustili

prazen tudi tretji parameter in da iščemo vzorec na odmiku *0*. Pri *RND* testih se ta parameter preskoči.

- Zadnji parameter nam pove, v kateri datoteki se nahaja naše besedilo. Ta parameter ne sme biti prazen in mora kazati na obstoječo datoteko.

Vse skupine testov morajo biti klicane v datoteki *StringSearch.atp* v razdelku *TestSets*, medtem ko imamo za določitev izvajanja posamezne skupine testov parametre zapisane v datoteki *TestSet1.atts* in *TestSet2.atts*.

Poglavje 8 Rezultati

Vsi algoritmi so testirani na različnih kombinacijah testov. Poizkusil sem jih združiti v najbolj logične skupine, da so vsi algoritmi pošteno ocenjeni glede na čas izvajanja, saj imajo nekateri algoritmi boljši čas pri drugačnih vzorcih kot drugi zaradi samega postopka primerjave ali predpriprave vzorca.

Rezultate testov, ki so zapisani v datoteki .em, sem z lastnim programom, napisanim v Javi, najprej uredil tako, da mi je združil posamezne teste po dolžini vzorca, katere sem potem uvozil v Microsoft Excel, kjer sem izdelal grafe.

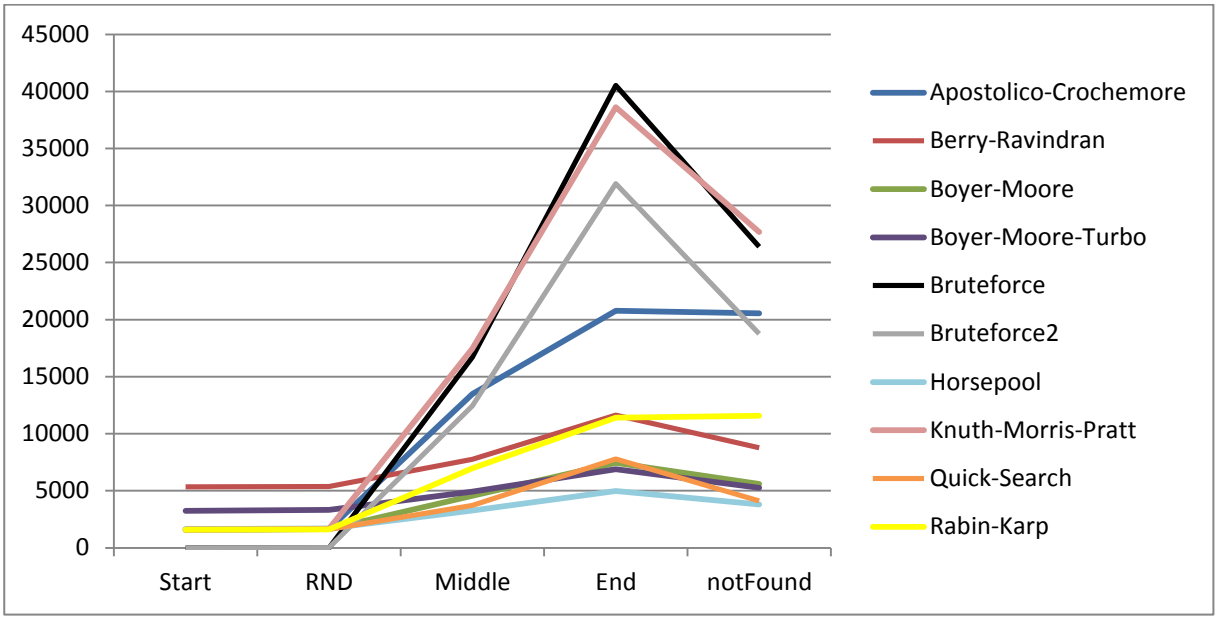
Rezultate sem ločil glede na uporabljeno besedilo za iskanje. Kjer so vzorci naključni, bom predstavil samo povprečni čas dvestotih zagonov testa.

Vsi prikazani izmerjeni časi v tabelah in na y-osi v grafih so prikazani v milisekundah (ms).

8.1 Biblija

Na besedilu Biblije so testi iskali pet različnih pozicij iskanega vzorca dolžine osem znakov: naključno, vzorec se začne na začetku, vzorec se nahaja na sredini besedila, vzorec je zadnjih osem znakov in vzorec ne obstaja v našem besedilu.

Primerjava minimalnega iskalnega časa v 200 ponovitvah testov:



Slika 9: Primerjava minimalnega iskalnega časa v 200 ponovitvah testov

Pattern Position	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
Start	1588	5348	1614	3256	1	1	1644	1559	1640	1586
RND	1633	5385	1654	3317	2	9	1688	1664	1626	1612
Middle	13492	7768	4551	4928	16720	12448	3263	17477	3711	6967
End	20776	11620	7440	6895	40524	31904	4990	38626	7802	11405
notFound	20561	8780	5612	5274	26398	18769	3801	27691	4122	11571

Tabela 1: Primerjava minimalnega iskalnega časa v 200 ponovitvah testov

Iz grafa je razvidno, da sta oba Bruteforce algoritma najhitrejša, ko najdemo vzorec takoj na začetku, ker nimata nobene pripravljalne faze, medtem ko sta najpočasnejša, ko iščemo znak

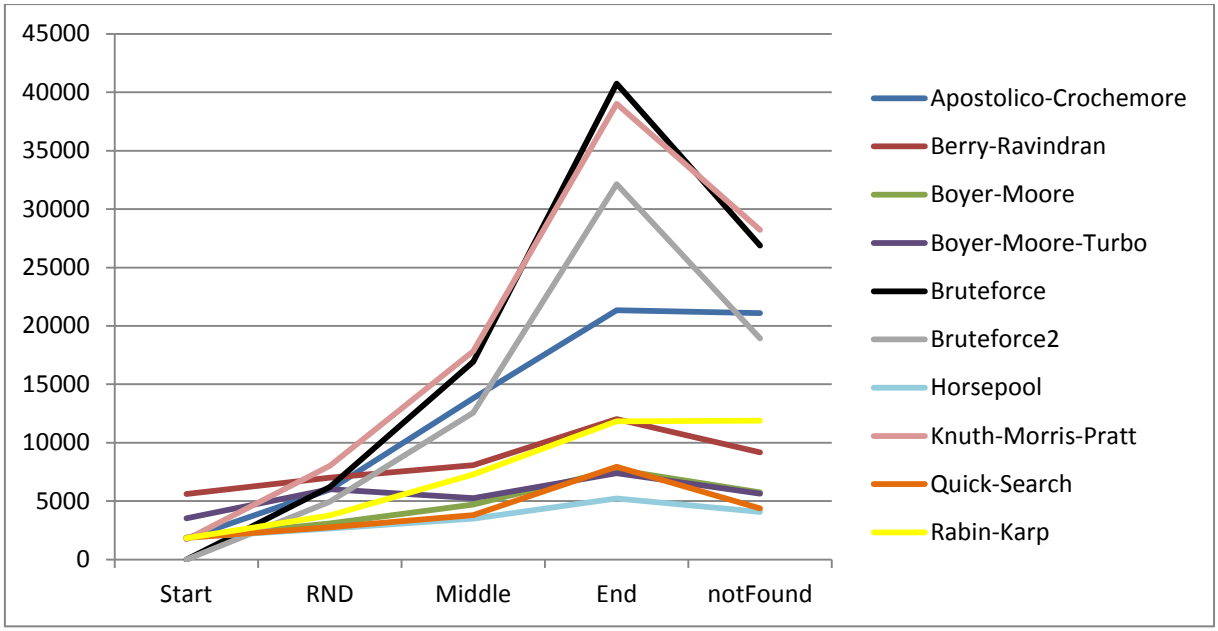
proti koncu besedila, ker preverjata vse znake brez preskakovanja. Ker imamo 200 ponovitev testa, sta bruteforce algoritma uspešna tudi pri naključni poziciji niza, saj se očitno zgodi, da je naključno izbrani niz vsaj enkrat na začetku.

Krajši časi pri testu, kjer vzorca ni v besedilu, pripišemo funkciji v vseh algoritmih, ki nadzirajo, da prenehamo z iskanjem, če nam je ostalo manj besedila, kot je dolžina nepreiskanega besedila.

Najbolje se obnesejo algoritmi, ki delujejo na podlagi Boyer-Moore analize delovanja s primerjanjem vzorca z desne proti levi. Ostali algoritmi primerjajo vzorec z leve proti desni in so tako počasnejši.

Knuth-Morris-Pratt algoritem je v tem primeru počasen zaradi kratkosti vzorca.

Primerjava povprečnega iskalnega časa pri 200 ponovitvah testov:



Slika 10: Primerjava povprečnega iskalnega časa pri 200 ponovitvah testov

Pattern Position	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
Start	1799	5620	1890	3540	1	1	1880	1732	1868	1830
RND	5988	7014	3082	6039	6232	4945	2653	8033	2756	3779
Middle	13803	8085	4705	5256	16927	12575	3495	17845	3818	7287
End	21341	12021	7705	7403	40730	32138	5226	39025	7950	11849
notFound	21108	9171	5760	5630	26885	18925	4087	28233	4392	11882

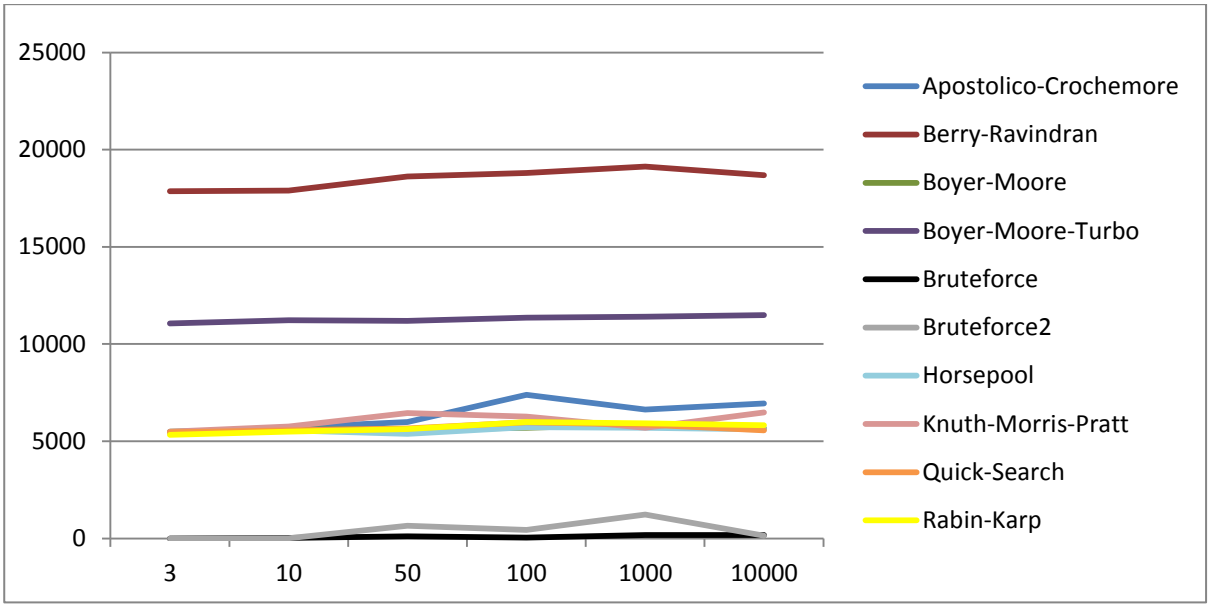
Tabela 2: Primerjava povprečnega iskalnega časa pri 200 ponovitvah testov

Pri merjenju povprečnega iskalnega časa vidimo, da ne odstopamo preveč od smernic najkrajšega iskalnega časa, razen pri testu, kjer vzorec izberemo naključno. Pri tem testu sem pričakoval, da se bodo časi bolj ujemali s testom, kjer iščemo vzorec na sredini besedila.

8.2 Projekt Gutenberg – 2010 CIA World Factbook

Na besedilu CIAWF so testi iskali vzorce šestih različnih dolžin, ki jih je algoritem naključno izbral v celotnem besedilu. Vsi testi so se ponovili 200-krat.

Primerjava najkrajšega iskalnega časa pri naključnih vzorcih določenih dolžin:



Slika 11: Primerjava najkrajšega iskalnega časa pri naključnih vzorcih določenih dolžin

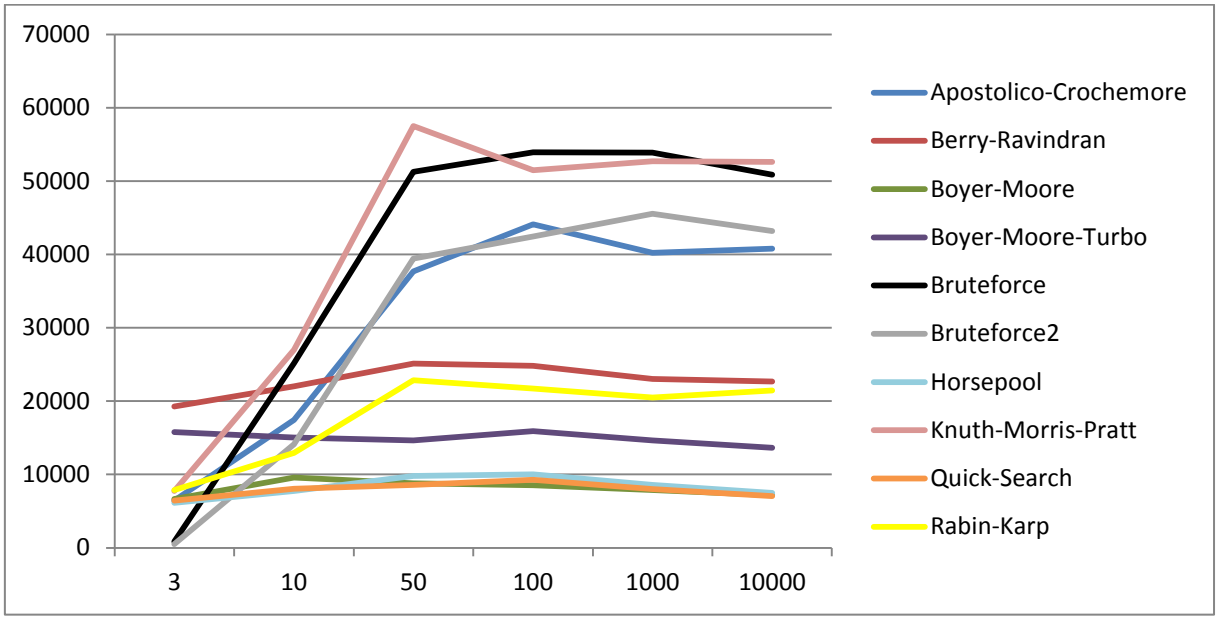
Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
3	5446	17857	5502	11055	2	2	5425	5503	5461	5333
10	5707	17893	5612	11227	30	17	5565	5754	5559	5492
50	5988	18623	5603	11188	108	658	5383	6454	5657	5624
100	7387	18807	5691	11362	40	442	5719	6273	5973	5994
1000	6627	19124	5873	11414	171	1236	5695	5691	5854	5921
10000	6945	18689	5664	11484	182	145	5610	6483	5561	5828

Tabela 3: Primerjava najkrajšega iskalnega časa pri naključnih vzorcih določenih dolžin

Zopet vidimo, da sta pri najkrajšem iskalnem času zaradi manjka pripravljalne faze najboljša bruteforce algoritma. V tem primeru je tudi razvidno, da je bil vsaj en vzorec izmed 200-tih izbran čisto na začetku.

Berry-Ravindran algoritem ima najdaljšo pripravljalno fazo, zato izpadejo najkrajši časi najslabši.

Primerjava povprečnega iskalnega časa pri naključnih vzorcih določenih dolžin, 200 ponovitev testa:



Slika 12: Primerjava povprečnega iskalnega časa pri naključnih vzorcih določenih dolžin

Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
3	6451	19257	6674	15781	855	506	6154	7766	6435	7882
10	17422	22025	9576	15019	25184	14137	7758	26997	8075	12934
50	37692	25122	8798	14634	51246	39417	9816	57486	8580	22843
100	44095	24828	8541	15895	53909	42464	10021	51474	9301	21705
1000	40225	23025	7888	14663	53867	45535	8597	52696	8026	20473
10000	40770	22684	7185	13625	50872	43188	7496	52595	7043	21450

Tabela 4: Primerjava povprečnega iskalnega časa pri naključnih vzorcih določenih dolžin

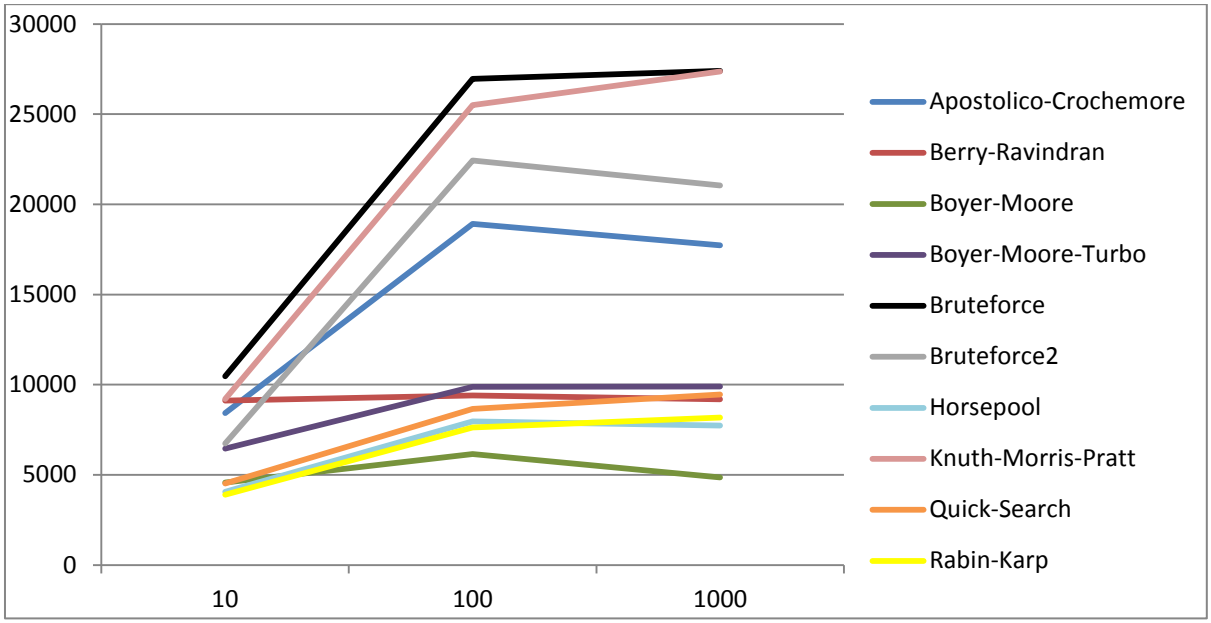
Tudi tukaj se situacija ponovi: bruteforce algoritma sta na kratkem vzorcu boljša od ostalih, ker nimata pripravljalne faze, medtem ko ostali algoritmi izgubijo nekaj časa pri pripravi vzorca za iskanje.

Pri daljših vzorcih opazimo, da pripravljalna faza nima več takega deleža pri samem izvajanju, medtem ko podobne čase pri različnih dolžinah vzorcev pripišemo sposobnosti preskakovanja delov besedila algoritmov, kjer se izračuna, da iskani vzorec ne more biti vsebovan. Daljši kot je vzorec, več besedila lahko preskočimo. Pri daljših vzorcih sta najslabša bruteforce algoritma, kjer se preverja vsak znak, medtem ko so algoritmi, ki delujejo po principu Boyer-Moore analize, najhitrejši.

8.3 Zapis gena bakterije E.Coli

V tem besedilu, ki izgleda za človeka precej naključno, je algoritem v 200 ponovitvah iskal vzorce dolžin 10, 100 in 1000, ki so bili izbrani naključno v celotnem besedilu.

Primerjava povprečnih časov iskanja naključnega vzorca dolžin 10, 100 in 1000:



Slika 13: Primerjava povprečnih časov iskanja naključnega vzorca dolžin 10, 100 in 1000

Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
10	8434	9116	4575	6456	10466	6748	4067	9205	4520	3903
100	18919	9403	6159	9889	26953	22430	7960	25510	8652	7621
1000	17723	9186	4853	9899	27400	21039	7734	27375	9462	8171

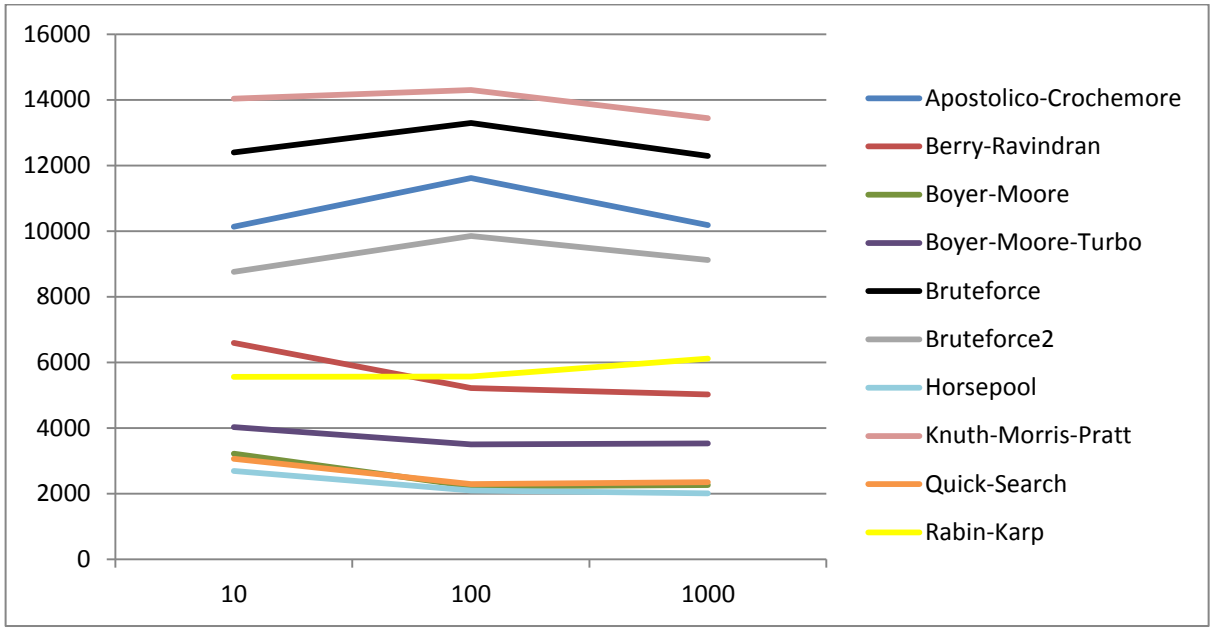
Tabela 5: Primerjava povprečnih časov iskanja naključnega vzorca dolžin 10, 100 in 1000

Pričakovani rezultati se zopet ponovijo. Najslabše čase sem dobil pri bruteforce algoritmih in Knuth-Morris-Pratt algoritmu, ki je načeloma izboljšani bruteforce s preskakovanjem delov besedila, kjer se izračuna, da vzorec ne more biti vsebovan.

Pomembnejši algoritmi se nahajajo v spodnjem delu grafa, kar je pričakovano.

8.4 Zapis sekvence proteina človeškega gena

Primerjava povprečnih časov iskanja naključnega vzorca dolžin 10, 100 in 1000:



Slika 14: Primerjava povprečnih časov iskanja naključnega vzorca dolžin 10, 100 in 1000

Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
10	10132	6593	3215	4023	12395	8758	2693	14035	3063	5557
100	11612	5221	2219	3504	13289	9854	2095	14294	2289	5567
1000	10181	5022	2258	3530	12285	9119	2007	13444	2353	6113

Tabela 6: Primerjava povprečnih časov iskanja naključnega vzorca dolžin 10, 100 in 1000

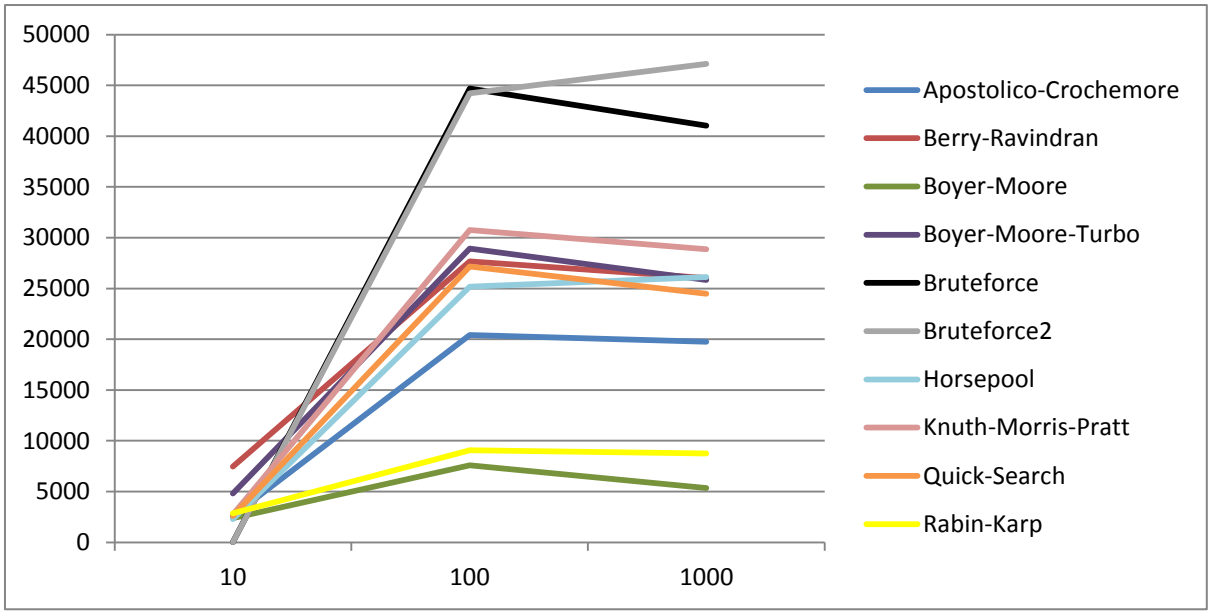
V primerjavi s prejšnjim testom opazimo, da so časi že pri vzorcu dolžine 10 precej razkropljeni. To se je zgodilo zato, ker ima besedilo zapisa sekvence proteina človeškega gena precej več različnih znakov kot pa zapis gena bakterije E.Coli.

Ostala opažanja ostajajo ista kot pri prejšnjem testu.

8.5 Besedila z naključnim ponavljanjem znakov

V teh besedilih zopet iščemo tri različno dolge vzorce, ki jih algoritem v predpripravi izbere naključno. Razlika v besedilih je v tem, da uporabljajo različno število istih znakov, ki se naključno ponavljajo. Vseh pet besedil je iste dolžine.

Iskanje vzorca v besedilu s ponavljajočima se dvema različnima znakoma:



Slika 15: Besedilo s ponavljajočima se dvema različnima znakoma

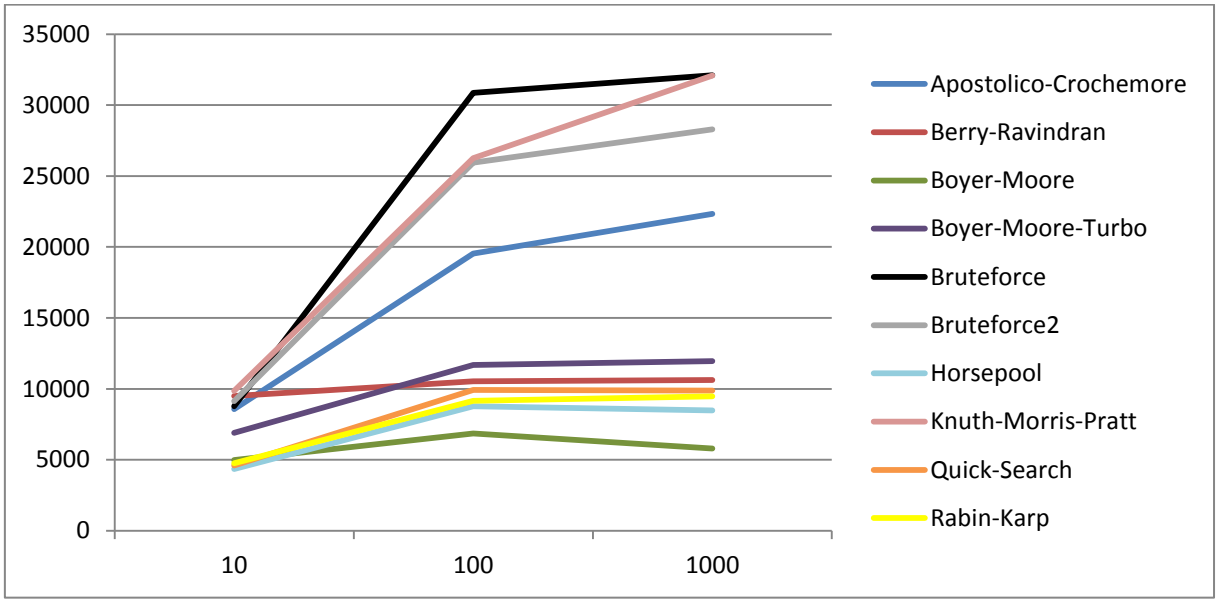
Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
10	2621	7453	2372	4817	20	20	2307	2727	2613	2861
100	20420	27666	7584	28938	44696	44211	25165	30768	27165	9070
1000	19761	26042	5355	25829	41046	47120	26115	28861	24475	8754

Tabela 7: Besedilo s ponavljajočima se dvema različnima znakoma

Rabin-Karp algoritem ima drugi najboljši čas zato, ker njegova pripravljalna faza zgoščevanja vzorca ne traja dolgo zaradi preprostega vzorca. Kasneje bomo videli, da se s kompliciranjem vzorca podaljšuje tudi pripravljalna faza tega algoritma in ga zato drugi algoritmi prehitijo.

Po drugi strani pa je razlika med Boyer-Moore in Boyer-Moore-Turbo algoritmoma to, da Boyer-Moore-Turbo uporablja še dodatni zamik, vendar se v tem primeru to ne izplača, ker imamo samo dva različna znaka in na koncu izračun tega dodatnega zamika samo doda k povprečnemu času.

Besedilo s ponavljajočimi se štirimi različnimi znaki:



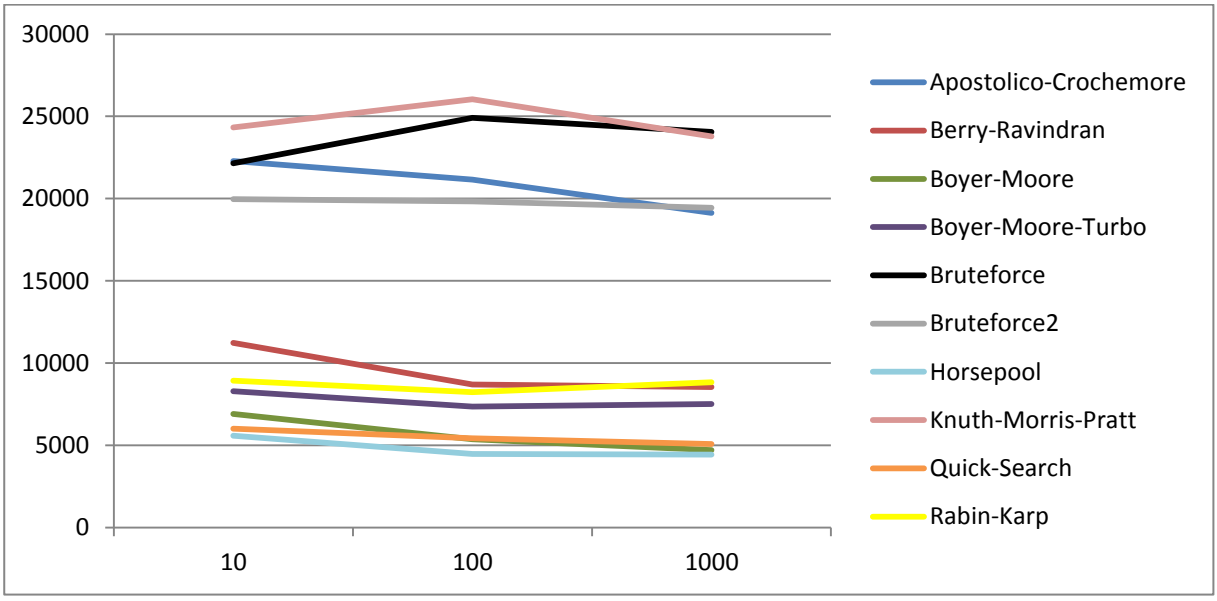
Slika 16: Besedilo s ponavljajočimi se štirimi različnimi znaki

Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
10	8561	9488	4981	6895	8771	9139	4342	9854	4603	4745
100	19526	10525	6850	11683	30862	25934	8771	26257	9913	9150
1000	22321	10616	5803	11952	32110	28293	8482	32083	9871	9462

Tabela 8: Besedilo s ponavljajočimi se štirimi različnimi znaki

Pri štirih različnih znakih se že pokaže, da Boyer-Moore-Turbo algoritem z dodano funkcijo dodatnega zamika doprinese k hitrejšemu iskanju, prav tako pa vidimo, da se Rabin-Karp-ova metoda zgoščevanja že zapleta in časovno obremeni izvajanje samega algoritma.

Besedilo s ponavljajočimi se osmimi različnimi znaki:



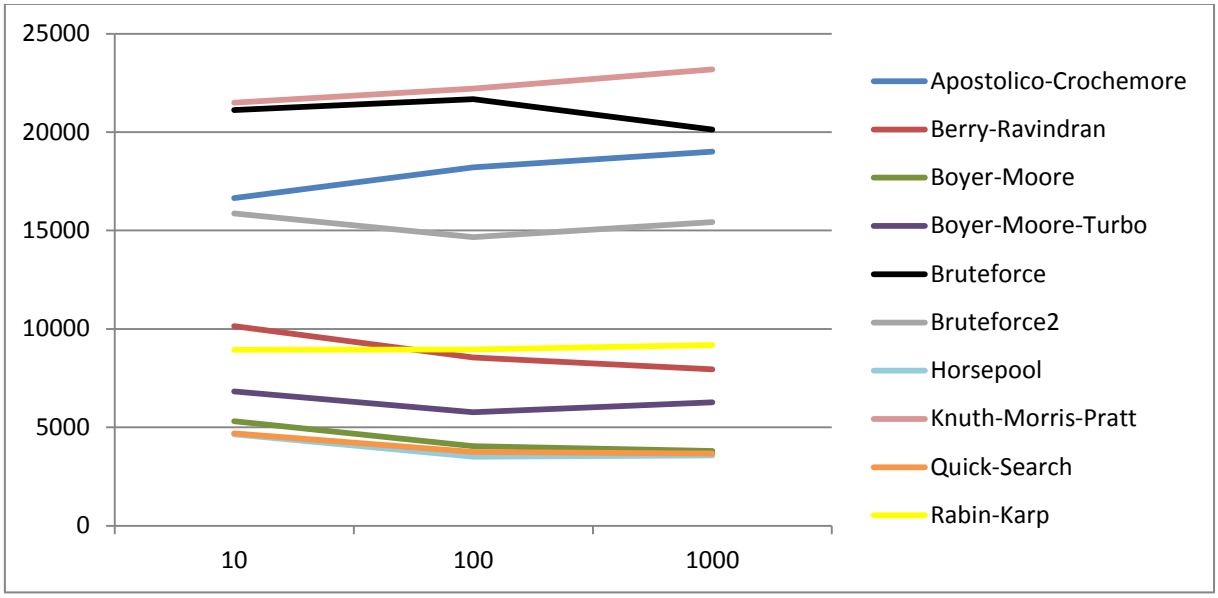
Slika 17: Besedilo s ponavljajočimi se osmimi različnimi znaki

Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
10	22285	11237	6918	8287	22143	19968	5586	24334	6022	8927
100	21159	8710	5366	7362	24920	19838	4471	26036	5423	8240
1000	19135	8554	4712	7523	24057	19443	4445	23782	5086	8831

Tabela 9: Besedilo s ponavljajočimi se osmimi različnimi znaki

Pri testu z osmimi različnimi znaki opazimo, da je iskanje vzorca dolžine 10 daljše kot ostala dva vzorca. To se zgodi zaradi preskokov besedila, v katerem se iskani vzorec ne more nahajati. Daljši je vzorec, več besedila lahko preskočimo.

Besedilo s ponavljajočimi se 16 različnimi znaki:



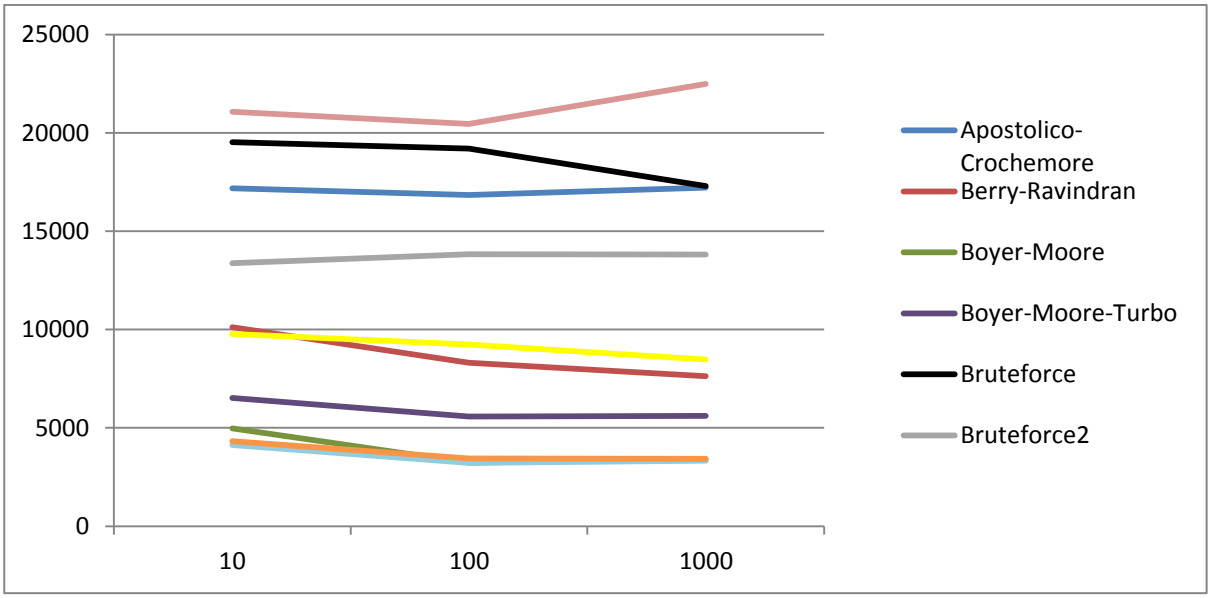
Slika 18: Besedilo s ponavljajočimi se 16 različnimi znaki

Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
10	16649	10150	5312	6818	21124	15868	4640	21497	4695	8934
100	18215	8553	4042	5774	21673	14656	3506	22219	3749	8955
1000	19008	7941	3799	6270	20123	15436	3570	23186	3674	9189

Tabela 10: Besedilo s ponavljajočimi se 16 različnimi znaki

Dobimo podobne rezultate kot pri prejšnjem testu, opazimo pa lahko, da je kompleksnost vzorca s 16 različnimi znaki precej vplivna na rezultate Rabin-Karp algoritma in njegovega zgoščevanja.

Besedilo s ponavljajočimi se 32 različnimi znaki:



Slika 19: Besedilo s ponavljajočimi se 32 različnimi znaki

Pattern Length	Apostolico-Crochemore	Berry-Ravindran	Boyer-Moore	Boyer-Moore-Turbo	Bruteforce	Bruteforce2	Horsepool	Knuth-Morris-Pratt	Quick-Search	Rabin-Karp
10	17185	10110	4978	6529	19519	13374	4130	21077	4321	9774
100	16839	8312	3233	5584	19200	13821	3211	20455	3438	9238
1000	17207	7622	3411	5607	17302	13806	3330	22485	3430	8472

Tabela 11: Besedilo s ponavljajočimi se 32 različnimi znaki

Pri najbolj kompleksnem besedilu vidimo, da je Rabin-Karp najpočasnejši od optimiziranih algoritmov zaradi zahtevnosti funkcije zgoščevanja.

Poglavje 9 Analiza in ugotovitve

Posamezen zagon algoritmov na vseh testih v 200 ponovitvah je povprečno trajal 21 minut na računalniku z naslednjimi komponentami:

- 16 GB 1600 MHz RAM
- SSD disk
- Windows 7 64bit SP1
- Intel Core i7 860 @ 3.04 GHz (vključena Intel Hyperthreading funkcionalnost)
- Java SDK 8

Od tega je proces Java porabil povprečno 3 GB RAMa in zasedel približno 15 % šest jeder procesorja. Ob vsakem zagonu testov sem vedno izklopil prvo jedro in njegovo virtualno jedro, ker na tem jedru potekajo procesi operacijskega sistema, ki bi lahko motili izvajanje mojih testov.

Pričakovano so se algoritmi, ki temeljijo na Boyer-Moore analizi iskanja nizov, vedno uvrščali med najhitrejše. Najslabši algoritmi so bili algoritmi na osnovi iskanja na silo oziroma bruteforca.

Presenetljivo je bil algoritem Knuth-Morris-Pratt vedno med najpočasnejšimi, če ne kar najpočasnejši, vendar razloga za to nisem uspel najti. Najprej sem raziskal, če je moja implementacija samega algoritma mogoče problematična, vendar sem pri treh različnih izvedenkah algoritma, ki so jih objavili avtorji na internetu, potrdil, da je moja implementacija ustrezna. Tudi ko sem zagnal algoritem z uporabo lastne funkcije za iskanje, torej brez podpore sistema ALGator, se je iskanje vedno izvedlo pravilno in bilo časovno primerljivo z algoritmi bruteforce. Na internetu so sicer navedene informacije, da je Knuth-Morris-Pratt algoritem hitrejši od bruteforce algoritmov, vendar v mojih testih tega nisem uspel potrditi.

Najboljši algoritem je Horsepool, ki je izvedenka Boyer-Moore algoritma, vendar v pripravljalni fazi uporabi samo zamik slabega znaka, kar zelo pohitri delovanje samega algoritma, vendar to deluje samo v primerih, ko je abeceda, torej raznolikost znakov, visoka, kot je, recimo, v primeru iskanja v besedilih, v katerih so uporabljeni vsi ASCII znaki. V najslabših primerih je algoritem precej počasnejši od njegovega predhodnika.

Sledijo mu ostali algoritmi na osnovi Boyer-Moore analize, izkaže pa se, da je oglaševani Boyer-Moore-Turbo algoritem slabši od originala. Zopet sem preveril implementacije tega algoritma na internetu in pri vseh se izkaže, da ta algoritem ne uporablja zamika dobre pripone, namesto tega uporablja zamik, ki je vzet kar neposredno iz tabele zamika slabega znaka. Prav tako uporablja funkcijo primerjave nizov, ki je zelo potratna.

Druge rešitve, ki bi to funkcijo zaobšla, na žalost nisem našel, tako da sem se moral zadovoljiti s to implementacijo, čeprav se izkaže, da pričakovani rezultati tega algoritma niso pravilni.

Analiziral sem še en precej slabo razložen in implementiran algoritem, Apostolico-Crochemore. Glede na to, da je algoritem izpeljanka Knuth-Morris-Pratt algoritma, so pričakovani časi tudi v okvirih časov, ki jih je dosegel Knuth-Morris-Pratt algoritem. Tudi tukaj sem raziskal, ali je implementacija ustrezna glede na druge objavljene, vendar sem lahko to preveril samo na dveh primerih, pri čemer sem obakrat potrdil pravilnost moje implementacije algoritma.

Algoritmi na osnovi bruteforce logike so bili najpočasnejši v večini primerov, razen takrat, ko smo opazovali iskanje vzorcev, ki se pojavijo takoj na začetku besedila. Ker nimajo pripravljalne faze, se v teh primerih ujemanje najde takoj po zagonu testa in so zato iskalni časi zelo kratki, vendar je to v stvarnem okolju iskanja v besedilih zelo majhna verjetnost.

Najlažjo implementacijo imata Bruteforce in Bruteforce2 algoritma, vendar sta najpočasnejša, sledi jima Rabin-Karp algoritem, kjer je zahtevnost implementacije odvisna od implementacije zgoščevalne funkcije. Algoritem se izkaže za zelo uspešnega, kadar imamo abecedo zelo majhno, ko pa se abeceda poveča, se algoritem začne uvrščati v sredino časovne zahtevnosti.

Glede na zahtevnost implementacije je Horsepool algoritem zopet v prednosti pred ostalimi, če upoštevamo še časovno kompleksnost in s tem ne izbiramo med bruteforce algoritmi. Ker uporablja samo del Boyer-Moore logike, je hitreje implementiran, vendar ima svoje omejitve, ko pride do obsežnosti abecede.

Knuth-Morris-Pratt in Apostolico-Chrochemore algoritma sta v vseh testih najslabša algoritma, če poleg časovnih zahtevnosti upoštevamo še zahtevnost implementacije.

Literatura

[1] T. Dobravec, „GitHub ALGator,“ 2014. [Elektronski]. Available: https://github.com/ALGatorDevel/Algator.

[2] T. Lecroq in C. Charras, „EXACT STRING MATCHING ALGORITHMS,“ Faculté des Sciences et des Techniques, 1997. [Elektronski]. Available: http://www-igm.univ-mlv.fr/~lecroq/string/index.html.

[3] Wikipedia users, „Wikipedia – String Searching Algorithms,“ 2014. [Elektronski]. Available: http://en.wikipedia.org/wiki/String_searching_algorithm.

[4] Wikipedia users, „Wikipedia,“ 2014. [Elektronski]. Available: http://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm.

[5] T. Dobravec, „GitHub ALGator Documentation,“ 2014. [Elektronski]. Available: https://github.com/ALGatorDevel/Algator/blob/master/doc/ALGator.docx.

[6] S. Faro in T. Lecroq, „SMART – String MAtching Research Tool,“ University of Catania, 1999. [Elektronski]. Available: http://www.dmi.unict.it/~faro/smart/index.php.