UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Tomaž Hočevar

Vzporedni algoritmi za iskanje nizov

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Borut Robič

Ljubljana, 2012

Rezultati diplomskega dela so intelektualna lastnina Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljanje ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje Fakultete za računalništvo in informatiko ter mentorja.

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: 01854/2012
Datum: 03.09.2012

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

Kandidat: **TOMAŽ HOČEVAR**

Naslov: **VZPOREDNI ALGORITMI ZA ISKANJE NIZOV**
PARALLEL STRING MATCHING ALGORITHMS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Predstavite algoritme za iskanje nizov. Algoritme opišite, predstavite njihove časovne zahtevnosti in jih implementirajte. Primerjajte učinkovitost algoritmov na različnih primerih podatkov. Preglejte možnosti za vzporedno iskanje nizov. Eksperimentalno preverite učinkovitost izbranih vzporednih algoritmov.

Mentor:

prof. dr. Borut Robič

Dekan:

prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU
diplomskega dela

Spodaj podpisani Tomaž Hočevar,

z vpisno številko 63070118,

sem avtor diplomskega dela z naslovom:

Vzporedni algoritmi za iskanje nizov

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 28.08.2012 Podpis avtorja:

Zahvala

Hvala mentorju prof. dr. Borutu Robiču za nasvete in usmerjanje pri izdelavi diplomske naloge.

Zahvaljujem se tudi Institutu Jožef Stefan, ki mi je omogočil izvajanje meritev na svoji računalniški opremi.

Posebej bi se zahvalil še svoji družini za vso podporo v času študija.

Kazalo

1 Uvod **1**

2 Zaporedni algoritmi **3**

2.1 Naivno iskanje . . . 3
2.2 Knuth-Morris-Pratt . . . 4
2.2.1 Delna ujemanja . . . 4
2.2.2 Faza iskanja . . . 5
2.3 Boyer-Moore . . . 7
2.3.1 Poravnava znaka . . . 8
2.3.2 Poravnava pripone . . . 9
2.4 Rabin-Karp . . . 13
2.5 Primerjava algoritmov . . . 14
2.6 Aho-Corasick . . . 16

3 Vzporedni algoritmi **21**

3.1 Delitev besedila . . . 22
3.2 Razširitev abecede . . . 23
3.3 Meritve . . . 26
3.4 Vishkinov algoritem . . . 27
3.4.1 Neperiodičen vzorec . . . 29
3.4.2 Izračun prič . . . 30
3.4.3 Periodičen vzorec . . . 31

4 Sklepne ugotovitve **33**

A Implementacije algoritmov **35**

Seznam slik **44**
Seznam tabel **45**
Stvarno kazalo **46**

Literatura **47**

Povzetek

V diplomski nalogi so predstavljeni algoritmi za iskanje nizov v besedilu. Ta problem se uvršča med najbolj osnovne probleme s področja obdelave besedil ali zaporedij znakov, zopet pa je postal aktualen z razvojem bioinformatike in potrebami po analizi nizov DNK. Poleg tega predstavljajo opisani algoritmi osnovo za reševanje kompleksnejših problemov s področja analize nizov. Predstavili smo klasične algoritme z linearno časovno zahtevnostjo, kot sta npr. Knuth-Morris-Pratt in Rabin-Karp, nato pa smo se posvetili še možnostim za vzporedno izvajanje. Opisali smo enostaven postopek, ki reši problem iskanja niza v besedilu dolžine N v času $O(\sqrt{N})$, če imamo na voljo $O(\sqrt{N})$ procesorjev. V nadaljevanju smo se posvetili še kompleksnejšemu Vishkinovemu algoritmu, ki reši problem v času $O(\log N)$. Algoritme smo primerjali tako na praktičnih kot tudi na bolj izrojenih testnih podatkih. Implementirali smo jih v programskem jeziku C++, za vzporedno izvajanje pa smo uporabili knjižnico OpenMP. Ugotovili smo, da se v večini praktičnih primerov najosnovnejši algoritmi obnesejo zelo dobro. Z algoritmi, ki so namenjeni zaporednemu izvajanju, lahko učinkovito rešimo tudi bolj ekstremne primere. Pri vzporednem izvajanju pa smo prišli do zaključka, da se današnji večjedrni računalniki še preveč razlikujejo od teoretičnega računskega modela, da bi lahko dosegli vse teoretične pohitritve.

Ključne besede:

niz, podniz, iskanje, algoritem, vzporedno, prstni odtis

Abstract

This thesis presents different string searching algorithms. The string searching or string matching problem is one of the most basic problems on strings. It resurfaced with the development of bioinformatics and the need for DNA sequence analysis. It also presents a foundation for solving other, more complex string problems. First, we describe classical algoritms with linear time complexity such as Knuth-Morris-Pratt and Rabin-Karp. Then we turned our attention to the possibilites of parallelization. We present a simple parallelization scheme which finds the pattern in a string of size N in $O(\sqrt{N})$ time on $O(\sqrt{N})$ processors and a more complex Vishkin algoritm which solves it in $O(\log N)$ time. The algorithms were compared on real as well as on degenerate test cases. They were implemented in C++ and with the use of OpenMP library for parallelization. We determined that the basic algorithms were sufficient for most practical purposes. The degenerate cases can also be solved efficiently with sequential linear time algorithms. However, the experiments on parallelization showed that multicore computers these days differ too much from computation models to reach the expected theoretic speedup.

Key words:

string, substring, search, algorithm, parallel, hash

Poglavje 1

Uvod

Ena najbolj naravnih predstavitev informacij je v obliki besedila, zato so se že zgodaj razvili številni algoritmi za njihovo obdelavo. Iskanje nizov je en izmed klasičnih problemov na tem področju, kjer nas zanima, na katerih mestih v besedilu se pojavi strnjeno zaporedje znakov, ki je podano z iskanim nizom. Večini uporabnikov računalnika bo najbolj očitna uporaba algoritmov za iskanje nizov v urejevalnikih besedil ali pa za preiskovanje ogromnih količin besedil, ki so objavljena na internetu v obliki spletnih strani. Učinkoviti algoritmi pa pridejo do izraza predvsem na drugih področjih, kjer so iskani nizi daljši kot besede v naravnem jeziku. Tak primer sta kompresija podatkov in iskanje virusov v programih. V zadnjih letih pa je vse bolj pomembno področje uporabe bioinformatika, kjer imamo opravka z nizi DNK. Problem iskanja nizov je dobro raziskan, saj je osnova za reševanje številnih bolj kompleksnih problemov. Tak primer je iskanje niza, pri čemer dovolimo tudi manjša odstopanja ali pa iskanje pogostih nizov določene dolžine v besedilu. Poleg tega lahko problem posplošimo na več dimenzij, če se lotimo iskanja podobnih območij v slikah.

Pomemben del algoritmov za iskanje nizov predstavljajo tudi metode, ki pred iskanjem indeksirajo ali kako drugače obdelajo besedilo. Do izraza pridejo predvsem, kadar večkrat izvajamo iskanje v istem besedilu. Tak primer so drevesa pripon, ki jih je možno zgraditi celo v linearnem času [11]. Slabost takih pristopov se pokaže v dodatni porabi prostora in času, ki je potreben za predobdelavo besedila. V nadaljevanju so obravnavani le algoritmi, ki zahtevajo kvečjemu predhodno obdelavo iskanih vzorcev, ki pa so tipično precej krajši od besedila.

Že v začetku 70. let so Knuth, Morris in Pratt razvili algoritem z linearno časovno zahtevnostjo [9] in s tem dosegli teoretično spodnjo mejo. Zaradi velike potrebe po učinkovitih algoritmih pa to ni ustavilo razvoja, saj sta Boyer in

Moore [2] leta 1977 objavila algoritem, ki je še izboljšal povprečno časovno zahtevnost. V tem času so bili razviti tudi številni drugi algoritmi, vendar po svojih lastnostih niso posebej odstopali od prej omenjenih. Leta 1987 sta Rabin in Karp [8] prestavila drugačen pristop k iskanju nizov, ki ni temeljil zgolj na primerjanju znakov. Algoritmi, ki so namenjeni vzporednemu izvajanju, so se začeli razvijati sredi 80. let. Vishkin [12] je leta 1985 objavil algoritem, ki ob zadostnem številu procesorjev reši problem v logaritmičnem času.

V tem delu so predstavljeni osnovni algoritmi za iskanje nizov, ki jih obravnavamo v prvem poglavju. Ker so vsi izrazito zaporedne narave, se v drugem poglavju posvetimo tehnikam in algoritmom za vzporedno iskanje nizov, ki v zadnjem času pridobivajo na pomembnosti zaradi rasti količine podatkov in vedno večje dostopnosti večprocesorskih računalnikov.

Poglavje 2

Zaporedni algoritmi

Definicija problema: Podana sta niza znakov: vzorec $V = v_0 v_1 \ldots v_{m-1}$ in besedilo $B = b_0 b_1 \ldots b_{n-1}$. Poiskati je potrebno vsa mesta i, kjer velja enakost $V = b_i b_{i+1} \ldots b_{i+(m-1)}$.

Algoritmi so ilustrirani tudi z izvorno kodo v programskem jeziku C. Tabeli znakov v in b predstavljata iskani vzorec in besedilo, števili m in n pa sta njuni dolžini. V tabelo r zapisujemo rezultate iskanja. Če v besedilu najdemu vzorec, ki se začenja na mestu i, to označimo z zapisom števila 1 v r_i.

2.1 Naivno iskanje

Problem najlažje rešimo tako, da za vsa možna mesta $i = 0 \ldots n-m$ preverimo, ali se vzorec ujema z besedilom (Alg. 2.1). Da preverimo, ali se vzorec ujema z besedilom na izbranem mestu, potrebujemo $O(m)$ primerjav znakov, preveriti pa je potrebno $O(n)$ mest. Tako naivno iskanje ima torej časovno zahtevnost $O(n * m)$.

Kljub kvadratni časovni zahtevnosti pa se algoritem dobro obnese v primerih, kjer izvajamo iskanje nad besedilom v naravnem jeziku. Ker je nabor znakov dokaj velik, pride na večini mest do razlike med vzorcem in besedilom že po nekaj primerjavah. Za ugotavljanje ujemanja torej v večini primerov potrebujemo precej manj kot m primerjav. Kadar pa se ujema prvih nekaj znakov, obstaja velika verjetnost, da se bo ujemal tudi preostanek.

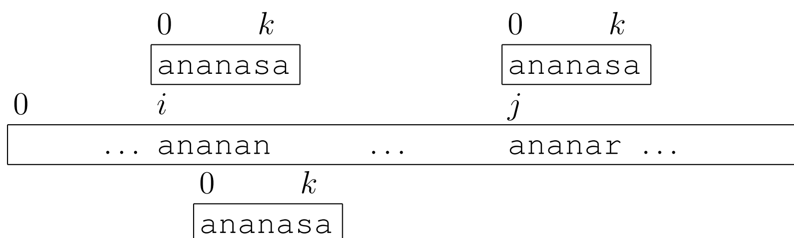
Algoritem 2.1: Naivno iskanje

```
1 void naive(char v[], char b[], char r[], int m, int n) {
2    for (int i=0; i<=n-m; i++) {
3       int ok=1;
4       for (int j=0; j<m; j++)
5          if (v[j]!=b[i+j]) { ok=0; break; }
6       if (ok) r[i]=1;
7    }
8 }
```

2.2 Knuth-Morris-Pratt

Algoritem KMP je pravzaprav nadgradnja naivnega iskanja. Za lažji opis ideje algoritma si oglejmo primer (Slika 2.1).

Slika 2.1: Ideja algoritma KMP



V besedilu se na mestu i ujema prvih k znakov vzorca. Naivni algoritem bi v naslednjem koraku preveril ujemanje na mestu $i+1$, vendar bi tam prišlo do razlike že pri prvem znaku. Mesti $i+2$ in $i+4$ sta edini, kjer bi se znaki ujemali vsaj do mesta $i+(k-1)$, kot pri iskanju ujemanja na mestu i. Če bi si vnaprej pripravili seznam neobetavnih mest, bi lahko preskočili preverjanje ujemanja na številnih mestih, saj lahko do podobne situacije pride tudi drugje v besedilu, kot npr. na mestu j. Kadar pa preverjamo ujemanje na obetavnem mestu, npr. $i+2$, lahko preverjamo ujemanje šele od $i+k$ naprej, saj se predhodni znaki zagotovo ujemajo, sicer tega mesta sploh ne bi preverjali.

2.2.1 Delna ujemanja

Za vsako predpono vzorca $p_i = v_0 v_1 \dots v_{i-1}$ moramo izračunati vrednost $f(i)$, ki predstavlja dolžino najdaljše predpone-pripone niza p_i (Tabela 2.1). Z izrazom *predpona-pripona* poimenujemo predpono niza, ki je hkrati tudi pripona

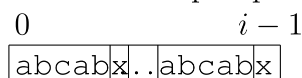
istega niza. Funkcijo f bomo imenovali *funkcija delnega ujemanja.* Če se v besedilu na mestu i ujema k znakov, bomo naslednje ujemanje v besedilu iskali šele na mestu $i + k - f(k)$.

Tabela 2.1: Funkcija delnega ujemanja

p_i	i	$f(i)$
ϵ	0	/
a	1	0
an	2	0
an<u>a</u>	3	1
an<u>an</u>	4	2
ana<u>na</u>	5	3
ananas	6	0
ananas<u>a</u>	7	1

Te vrednosti je trivialno izračunati v času $O(m^2)$, vendar se bomo posvetili učinkovitejšemu algoritmu. Vrednosti bomo računali od krajših predpon proti daljšim in pri tem izkoristili lastnost, da ima niz p_{i-1} predpono-pripono, ki je dolga vsaj $f(i) - 1$ znakov. Nizu p_i lahko odrežemo zadnji znak in ohranimo predpono-pripono dolžine $f(i) - 1$ (Slika 2.2).

Slika 2.2: Rekurzivna struktura predpon-pripon niza p_i



Vrednost $f(i)$ lahko torej izračunamo tako, da poiščemo najdaljšo predpono, ki je hkrati tudi pripona niza p_{i-1} in jo je možno podaljšati z znakom v_{i-1}. Najdaljša predpona-pripona niza p_{i-1} je dolga ravno $f(i-1)$, druga najdaljša je dolga $f(f(i-1))$ in tako naprej. V danem primeru (Slika 2.2) sta to niza "abcab" in "ab". Za primer implementacije glej Alg. 2.2.

2.2.2 Faza iskanja

Ko imamo izračunane vrednosti funkcije delnega ujemanja, lahko začnemo z iskanjem, in sicer na mestu $i = 0$ in z velikostjo delnega ujemanja $j = 0$. Spremenljivka i predstavlja kazalec na zadnji obravnavani znak v besedilu.

Algoritem 2.2: Izračun vrednosti funkcije delnega ujemanja

```
1 f[0]=0; f[1]=0;
2 int i=2, j=0;
3 while (i<=m) {
4    if (v[j]==v[i-1]) { j++; f[i]=j; i++; } // j=f[i-1]
5    else if (j==0) { f[i]=0; i++; }
6    else { j=f[j]; }
7 }
```

Ujemanje, ki ustreza trenutno obravnavanemu znaku, se torej začenja v besedilu na mestu $i - j$. Na vsakem koraku poskusimo razširiti delno ujemanje z i-tim znakom v besedilu. To lahko storimo, če je v_j enak b_i. Če nismo uspešni, se s funkcijo delnega ujemanja premaknemo na naslednje obetavno mesto oz. skrajšamo dolžino ujemanja. Če delno ujemanje doseže dolžino vzorca, smo našli pojavitev (Alg. 2.3).

Algoritem 2.3: Faza iskanja pri algoritmu KMP

```
1 int i=0, j=0;
2 while (i<n) {
3    if (v[j]==b[i]) { // uspesno razsirimo delno ujemanje
4       j++; i++;
5       if (j==m) { j=f[j]; r[i-m]=1; } // nasli smo vzorec
6    } else if (j==0) { i++; } // ne ujema se niti prva crka
7    else { j=f[j]; } // poskusimo krajse delno ujemanje
8 }
```

Časovna zahtevnost izračuna funkcije delnih ujemanj je $O(m)$, faze iskanja pa $O(n)$, kar je skupaj $O(n + m)$. Da so časovne zahtevnosti linearne, ni očitno, zato si oglejmo primer faze iskanja vzorca "ananas" v besedilu "anananasa"(Tabela 2.2). Utemeljitev linearne časovne zahtevnosti izračuna funkcije delnih ujemanj je namreč povsem podobna kot pri fazi iskanja. Algoritem se po besedilu premika z dvema kazalcema $i - j$ in i, ki označujeta začetek in konec delnega ujemanja vzorca z besedilom. Na vsakem koraku algoritma se eden izmed njiju premakne proti desni, ko oba dosežeta desni rob, pa se algoritem ustavi. Skupaj torej naredita največ $2n$ premikov. V izvorni kodi (Alg. 2.3) se lepo vidi, da se v vsaki ponovitvi zanke poveča vrednost izraza $2i - j$ vsaj za 1, vendar nikoli ne preseže $2n$.

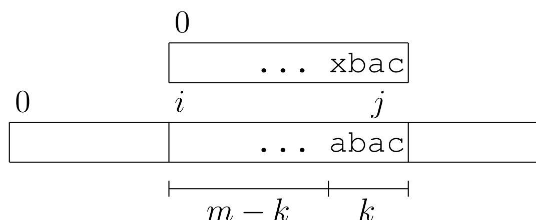
Tabela 2.2: Primer faze iskanja z algoritmom KMP

anananasa	i	j	i-j	f_j
	0	0	0	
a	1	1	0	
an	2	2	0	
ana	3	3	0	
anan	4	4	0	
anana	5	5	0	3
ana	5	3	2	
anan	6	4	2	
anana	7	5	2	
ananas	8	**6**	2	0
	8	0	8	
a	9	1	8	

2.3 Boyer-Moore

Večina algoritmov za iskanje nizov stremi k temu, da bi vsak znak besedila obdelala zgolj enkrat. Algoritem Boyer-Moore gre še korak dlje. Njegov cilj je v celoti preskočiti čim več znakov in v ta namen preverja ujemanje vzorca na nekem mestu v besedilu od zadaj naprej. Pri ugotavljanju, ali se vzorec ujema z besedilom na mestu i, bo algoritem najprej preveril, ali se zadnji znak vzorca ujema z znakom b_{i+m-1}, nato bo preveril predzadnjega itd. Če v tem procesu preverjanja najde razliko po k preverjenih znakih, izvleče iz obdelanih znakov $b_{i+m-k}, \ldots, b_{i+m-1}$ čim več informacije o tem, na katerem mestu $j > i$ bi se še lahko nahajal vzorec. Če velja npr. $j \geq i + m - k$, bo algoritem povsem preskočil $m - k$ znakov $b_i, b_{i+1}, \ldots, b_{i+m-k-1}$ (Slika 2.3). Za iskanje naslednjega mesta, kjer bi se lahko nahajal vzorec, se uporabljata dve različni metodi. Vsako od teh metod bi lahko uporabljali tudi samostojno, saj nobena izmed njiju ne preskoči kakšne pojavitve vzorca v besedilu. Lahko pa metodi združimo in na vsakem koraku upoštevamo bolj desno izmed naslednjih mest, ki ju predlagata metodi.

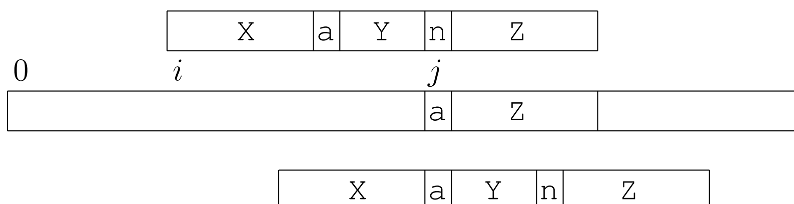
Slika 2.3: Preskakovanje znakov



2.3.1 Poravnava znaka

Prva metoda izkorišča velikost abecede. Večja kot je abeceda, bolj bo v povprečju metoda uspešna. Algoritem, ki uporablja samo ta del algoritma BM, se imenuje Horspoolov algoritem. Ideja je dokaj enostavna in zelo učinkovita pri iskanju v besedilih, ki so napisana v naravnem jeziku, zato se pogosto uporablja v urejevalnikih besedil. Recimo, da preverjamo, ali se vzorec pojavi na i-tem mestu v besedilu, pri tem pa pride do razlike na mestu j (Slika 2.4). Če podniza Y in Z ne vsebujeta črke 'a', je smiselno premakniti iskani vzorec za toliko mest, da se bo znak b_j ujemal z znakom v vzorcu. V ta namen si pripravimo tabelo C, ki za vsak znak iz abecede vsebuje najbolj desno mesto v vzorcu, kjer se ta znak pojavi (Tabela 2.3). Izjema je zadnji znak v vzorcu, ki ga ne upoštevamo. Vrednosti izračunamo tako, da se od predzadnjega znaka premikamo proti začetku vzorca in za vsak znak, ki še nima vrednosti v tabeli, vpišemo trenutno mesto. Znaki, ki se ne pojavijo v vzorcu, imajo v tabeli vrednost -1. Če bi se v danem primeru (Slika 2.4) znak 'a' pojavil tudi v podnizu Z, bi bilo potrebno vzorec premakniti v levo, česar pa ne želimo. Novo mesto iskanja vzorca je torej enako $i + max(1, j - i - C(b_j))$.

Slika 2.4: Poravnava znaka



Pri majhnih abecedah se vsak znak iz abecede verjetno pojavi zelo blizu konca vzorca, zato so skoki precej kratki. Idejo o poravnavi znaka pa lahko razširimo s tem, da namesto zadnje pojavitve vsakega znaka hranimo tabelo,

Tabela 2.3: Tabela zadnjih pojavitev znakov v iskanem vzorcu

banana	mesto
n	4
a	3
b	0
ostali znaki	-1

ki za vsako mesto i in vsak znak x hrani mesto $t_{i,x} = max(j < i | v_j = x)$ predhodnje pojavitve tega znaka v vzorcu. Taka tabela bi zahtevala $O(m * \Sigma)$ prostora, kar ni sprejemljivo. Namesto celotne tabele pa si lahko za vsak znak pripravimo urejen seznam mest, kjer se ta znak pojavi v vzorcu. Pri iskanju predhodnjega mesta nekega znaka se moramo sprehoditi čez seznam mest za ta znak, zaradi česar ima lahko posamezna poizvedba časovno zahtevnost $O(m)$. Če pa analiziramo amortizirano časovno zahtevnost, bomo ugotovili, da je izvedeno število operacij največ toliko, kot je primerjav znakov v celotnem algoritmu in torej ne vpliva na časovno zahtevnost algoritma. Število operacij se kvečjemu podvoji, v večini primerov pa prihranki odtehtajo dodatno število operacij. Za iskanje po urejenem seznamu mest se lahko poslužimo tudi dvojiškega iskanja, uporabnost pa je odvisna od strukture besedila in vzorca.

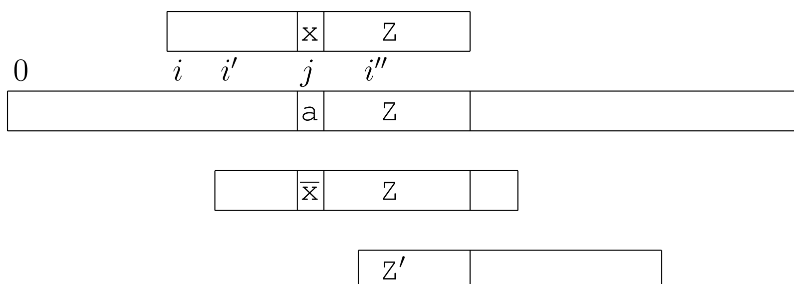
2.3.2 Poravnava pripone

Drugi del algoritma BM se osredotoča na pripono, ki se ujema pri trenutni poravnavi vzorca in besedila. Zanima nas, za koliko mest lahko premaknemo vzorec, pri čemer ne želimo pokvariti ujemanja trenutno ujemajoče se pripone vzorca. Recimo, da pride do razlike med vzorcem in besedilom pri k-tem znaku s konca vzorca, kar odgovarja mestu j v besedilu. Besedilo torej vsebuje podniz oblike $\overline{v_{m-k}} v_{m-k+1} \ldots v_{m-1}$. $\overline{x}$ predstavlja katerikoli znak, ki ni enak znaku x. Iščemo minimalen premik v desno $i' > i$, ki bo konsistenten z omenjenim podnizom: na j-tem mestu se mora znak v vzorcu razlikovati od tistega, ki je povzročil neujemanje z besedilom, naslednjih $k-1$ znakov pa se mora ujemati s pripono vzorca (Slika 2.5). Pri tem moramo paziti na primer, ko je premik tolikšen, da se vzorec začne na nekem mestu i'', ki je večje od j.

$$L(i) = max(j-1 \Big| S(j) = m-i) \tag{2.1}$$

$$l(i) = max(j-1 \Big| j \leq m-i \wedge S(j) = j) \tag{2.2}$$

Slika 2.5: Poravnava pripone



Ker moramo obravnavati dva ločena prej omenjena primera, si moramo pripraviti dve tabeli. Njun izračun v linearnem času je vse prej kot enostaven. $L(i)$ naj predstavlja končno mesto zadnje pojavitve niza $v_i \ldots v_{m-1}$ v vzorcu v, pred katerim se ne nahaja znak v_{i-1}. $l(i)$ pa definiramo kot najdaljšo predpono vzorca, ki se ujema s pripono niza $s_i \equiv v_i \ldots v_{m-1}$ (Tabela 2.4). Predpostavimo, da imamo na voljo še funkcijo S, vrednost $S(i)$ pa predstavlja dolžino najdaljše skupne pripone nizov p_i in v. Iz enakosti 2.1 in 2.2 lahko v linearnem času izračunamo pomožni tabeli, če poznamo vrednosti funkcije S. Zaradi preglednosti pa bomo tudi to funkcijo tabelirali posredno preko tabele P dolžin najdaljših skupnih predpon obrnjenega vzorca v^R in njegovih pripon.

Recimo, da algoritem preverja, ali se vzorec pojavi v besedilu na mestu $k-m+1$. Zadnji znak vzorca je torej poravnan s k-tim znakom besedila. Algoritem bo po vrsti preverjal znake s konca vzorca in ugotovil, da pride do razlike pri primerjavi znakov v_i in b_h. Če je vrednost $L(i+1)$ definirana, lahko vzorec premaknemo v desno za $(m-1) - L(i+1)$ mest, sicer pa izvedemo premik v desno za $m - l(i+1)$ mest.

Problem izračuna pomožnih tabel algoritma BM smo torej poenostavili na izračun funkcije $P(i)$ najdaljših skupnih predpon danega niza x in vseh njegovih pripon: $P(i) = max(k \Big| x_0 \ldots x_{k-1} = x_{i-1} \ldots x_{i+k})$. Vrednosti $P(i)$ bomo računali od $i = 1$ proti m. To je možno storiti v linearnem času ob pametni uporabi informacij, ki jih dobimo od izračunu predhodnjih vrednosti. Reševanja se lahko lotimo naivno tako, da za vsako pripono preverimo, koliko znakov se ujema z nizom. Pri računanju i-te vrednosti označimo s s mesto, pri

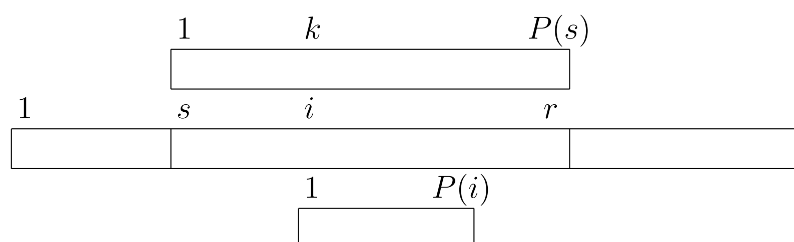
Tabela 2.4: Primer tabel $L(i)$ in $l(i)$

aabbabab	i	$L(i)$
~~b~~	8	6
~~a~~b	7	3
~~b~~ab	6	2
~~a~~bab	5	5
~~b~~abab	4	-1
~~b~~babab	3	-1
~~a~~bbabab	2	-1
~~a~~abbabab	1	-1

aabaaba	i	$l(i)$
ϵ	7	0
a	6	1
ba	5	1
aba	4	1
aaba	3	4
baaba	2	4
abaaba	1	4

katerem smo do sedaj primerjali najbolj oddaljeni znak v nizu: $s = \underset{j<i}{\mathrm{argmax}}(j + P(j))$. Argmax predstavlja vrednost, pri kateri podana funkcija doseže svoj maksimum. Če se ujemanje na i-tem mestu ne razteza preko mesta $r = s + P(s) - 1$, imamo vrednost $P(i)$ že izračunano, saj je enaka $P(k), k = i - s$ (Slika 2.6). V nasprotnem primeru pa moramo primerjati samo znake v nizu od mesta $r + 1$ naprej, saj se predhodni zagotovo ujemajo. Taka izboljšava obravnava vsak znak natanko enkrat in vodi k algoritmu s časovno zahtevnostjo $O(m)$.

Slika 2.6: Izračun najdaljših skupnih predpon niza in njegovih pripon

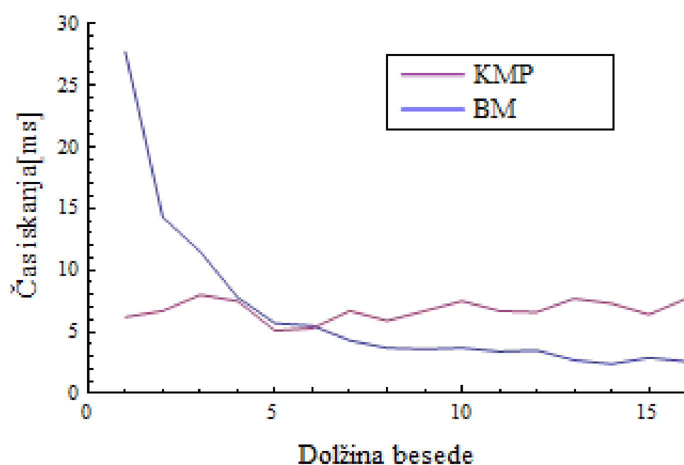


Algoritem BM ima v najboljšem primeru časovno zahtevnost $O(n/m)$. Tak primer bi bil npr. iskanje vzorca samih 'a'-jev v besedilu, ki ne vsebuje črke 'a'. Leta 1991 je Cole[5] pokazal, da algoritem najde prvo pojavitev vzorca v besedilu v največ $3n$ primerjavah znakov. Če želimo poiskati vse pojavitve vzorca v besedilu, pa je časovna zahtevnost lahko še vedno $O(n*m)$. Do težave pride pri periodičnih vzorcih, ki se v besedilu pojavijo na $O(n)$ mestih. Ta

problem se odpravi z enostavno izboljšavo, ki je poznana kot Galilovo pravilo. To pravilo pravi, da ni potrebno primerjati znakov, za katere vemo, da se bodo zagotovo ujemali, ko se v besedilu premaknemo s pravilom za poravnavo pripon. S to izboljšavo ima algoritem BM v vseh primerih časovno zahtevnost $O(n+m)$, pri obravnavi besedil v naravnih jezikih pa je izrazito hitrejši. Leta 2007 sta Sustik in Moore[10] predstavila nov algoritem, osnovan na algoritmu BM, ki izboljša čas iskanja v besedilih z majhno abecedo (kot npr. pri nizih DNK), vendar na račun večjega časa, ki je potreben za predobdelavo iskanega vzorca.

Pri iskanju v besedilih, ki so napisana v naravnem jeziku, ima algoritem v povprečnem primeru časovno zahtevnost $O(n/m)$. To smo preverili z iskanjem različno dolgih besed v literarnem delu Vojna in mir.[1] Za vzorce, ki smo jih iskali v besedilu, smo uporabili najpogostejše besede posameznih dolžin od 1 do 16. Na grafu 2.7 se lepo vidi, kako čas iskanja pada z dolžino iskane besede. Do manjšega odstopanja pride le pri besedah dolžin 13 (involuntarily) in 14 (responsibility). Iskanje teh besed je bilo še posebej hitro, saj se končata na črko 'y', ki je bolj redka in je v večini primerov prišlo do neujemanja že pri preverjanju zadnje črke. Za primerjavo so prikazani še časi iskanja z algoritmom KMP, ki so bolj odvisni od strukture iskanih besed kot pa od njihovih dolžin.

Slika 2.7: Iskanje najpogosteših besed v knjigi Vojna in mir



[1]Projekt Gutenberg: http://www.gutenberg.org/ebooks/2600

2.4 Rabin-Karp

Algoritem RK se razlikuje od ostalih algoritmov za iskanje nizov po tem, da ne temelji na primerjavah znakov, temveč izkorišča prilagojenost procesorjev za izvajanje aritmetičnih operacij. Algoritem izračuna t. i. prstni odtis h_i dolžine m znakov na vseh mestih v besedilu in jih primerja s prstnim odtisom iskanega vzorca. *Prstni odtis* predstavimo s številom, ki ga izračunamo iz zaporedja znakov in ki to zaporedje čim bolj natančno določa. Enaki zaporedji znakov imata vedno enak prstni odtis, različni zaporedji pa imata z veliko verjetnostjo različna odtisa. Rešitev lahko prilagodimo tudi za sočasno iskanje več vzorcev v besedilu. Z razpršeno tabelo, ki vsebuje prstne odtise vseh iskanih vzorcev, lahko učinkovito ugotovimo, ali prstni odtis podniza v besedilu ustreza kateremu od vzorcev.

Če abeceda Σ vsebuje k znakov, jih lahko oštevilčimo z vrednostmi med 0 in $k-1$ ter pretvorimo besedilo v zaporedje števil b_i. Pomembna lastnost prstnega odtisa je, da lahko učinkovito izračunamo odtis h_{i+1}, če poznamo vrednost h_i (Enačba 2.5). To je mogoče doseči, ker se podniza, katerih prstne odtise računamo, v veliki meri prekrivata. Tehnika je znana pod imenom "rolling hash".

$$h_i = k^{m-1} * b_i + k^{m-2} * b_{i+1} + \ldots + k * b_{i+k-2} + b_{i+m-1} \tag{2.3}$$

$$h_{i+1} = k^{m-1} * b_{i+1} + \ldots + k^2 * b_{i+k-2} + k * b_{i+m-1} + b_{i+m} \tag{2.4}$$

$$h_{i+1} = (h_i - k^{m-1} * b_i) * k + b_{i+m} \tag{2.5}$$

Izračunati moramo $O(n)$ prstnih odtisov, vsakega pa izračunamo iz predhodnjega v konstantnem času (Tabela 2.5). Časovna zahtevnost algoritma je torej $O(n)$. Dolžina iskanega vzorca in velikost abecede pa sta ponavadi prevelika, da bi lahko učinkovito računali vrednosti h_i, zato se omejimo na računanje prstnih odtisov v kolobarju $\mathbb{Z}_w$ ostankov po modulu w. Pri takem računanju prstnih odtisov pa se lahko zgodi, da imata različna niza enak prstni odtis.

Na tem mestu je potrebno sprejeti kompromis med učinkovitostjo in pravilnostjo. Lahko se sprijaznimo z dejstvom, da bomo v zelo majhnem odstotku primerov dobili napačen rezultat. Tak pristop se imenuje Monte Carlo metoda. Verjetnost napake lahko zmanjšamo z računanjem več različnih prstnih odtisov − z drugačno izbiro delitelja w.

Druga možnost je, da rezultate naknadno preverimo in izločimo odvečne rešitve, ki nastanejo zaradi trka prstnih odtisov. Do trka pride, kadar se

$$\Sigma = \{\text{a, c, g, t}\}, k = 4$$
$$h(\text{agca}) = 64 * 0 + 16 * 2 + 4 * 1 + 0 = 36$$

Tabela 2.5: Primer iskanja z algoritmom RK

tagcagcata	i	h_i
tagc	0	201
agca	1	**36**
gcag	2	146
cagc	3	73
agca	4	**36**
gcat	5	147
cata	6	76

različni zaporedji znakov preslikata v enak prstni odtis zaradi računanja v kolobarju $\mathbb{Z}_w$. To ima lahko velik vpliv na časovno zahtevnost algoritma. V najslabšem primeru moramo še enkrat preveriti $O(n)$ mest, zaradi česar je lahko končna časovna zahtevnost tudi $O(n * m)$. Algoritmi, ki imajo v majhnem odstotku primerov izrazito večjo časovno zahtevnost, so poznani kot Las Vegas algoritmi.

2.5 Primerjava algoritmov

Tabela 2.6: Povzetek zaporednih algoritmov

algoritem	časovna zahtevnost	najboljša časovna zahtevnost	prostorska zahtevnost
Naivno iskanje	$O(n * m)$	$O(n)$	$O(1)$
Knuth-Morris-Pratt	$O(n + m)$	$O(n + m)$	$O(m)$
Boyer-Moore	$O(n + m)$	$O(n/m + m)$	$O(m)$
Rabin-Karp	$O(n + m)$	$O(n + m)$	$O(1)$

Predstavljene algoritme smo testirali na treh teoretičnih primerih, ki so vsebovali le črki a in b, razlikovali pa so se v strukturi iskanega vzorca in besedila. Poleg tega smo izvedli testiranje tudi nad dvema bolj praktičnima primeroma. V besedilu knjige Vojna in mir smo iskali besedo Bolkonski, ki je primek enega od glavnih junakov. V genomu cianobakterije[2] pa smo iskali podniz, ki po dolžini približno ustreza dolžini enega gena.

Algoritme smo najprej testirali na povsem naključnem besedilu dolžine 10 milijonov znakov in z naključnim vzorcem dolžine 20. Ker je takih vzorcev $2^{20} \approx 10^6$, smo pričakovali približno 10 zadetkov. Rezultati so prikazani na grafu (a), od koder je razvidno, da je naivni algoritem pri naključnih podatkih povsem sprejemljiv. Naivni algoritem se je v splošnem obnesel precej dobro, z izjemo posebnih primerov, kot npr. pri periodični strukturi vzorca in besedila (b), kjer s stališča učinkovitosti povsem odpove. Taki primeri so motivacija za uporabo algoritma z linearno asimptotično časovno zahtevnostjo, kot je algoritem KMP, ki je bil na tem primeru med hitrejšimi. Boljša časovna zahtevnost pa ne pomeni vedno tudi hitrejšega izvajanja. To je lepo razvidno v primeru (c), kjer algoritma KMP in BM porabita veliko časa za analizo iskanega vzorca, ki je izjemno dolg. Pri iskanju po nizu DNK (e) pa je do izraza prišel algoritem BM, ki lahko zaradi dolgega vzorca povsem preskoči nekatera mesta v besedilu. Zaradi večje abecede bi še boljše rezultate pričakovali v primeru (d), vendar je iskana beseda prekratka, da bi algoritem BM odstopal od ostalih. V povprečju je bil najhitrejši algoritem RK, vendar je potrebno omeniti, da smo uporabljali Monte Carlo različico, tako da ne moremo biti povsem prepričani v pravilnost najdenih pojavitev.

Zaključili bi lahko, da je za večino uporab v praksi povsem primeren že naivni algoritem. Če želimo pohitriti iskanje, je najpreprostejša in zelo učinkovita rešitev uporaba prstnih odtisov z algoritmom RK. Kadar imamo opravka z daljšimi vzorci in večjimi abecedami, pa je prava izbira algoritem BM, ki je kot nalašč za uporabo v bioinformatiki. Njegova edina pomanjklivost je zahtevnost implementacije, zato se velikokrat uporablja Horspoolov algoritem, ki je pravzaprav poenostavljena različica algoritma BM. Algoritem KMP je zanimiv predvsem s teoretičnega vidika kot prvi algoritem z linearno časovno zahtevnostjo in kot osnova algoritma Aho-Corasick, ki je predstavljen v poglavju 2.6.

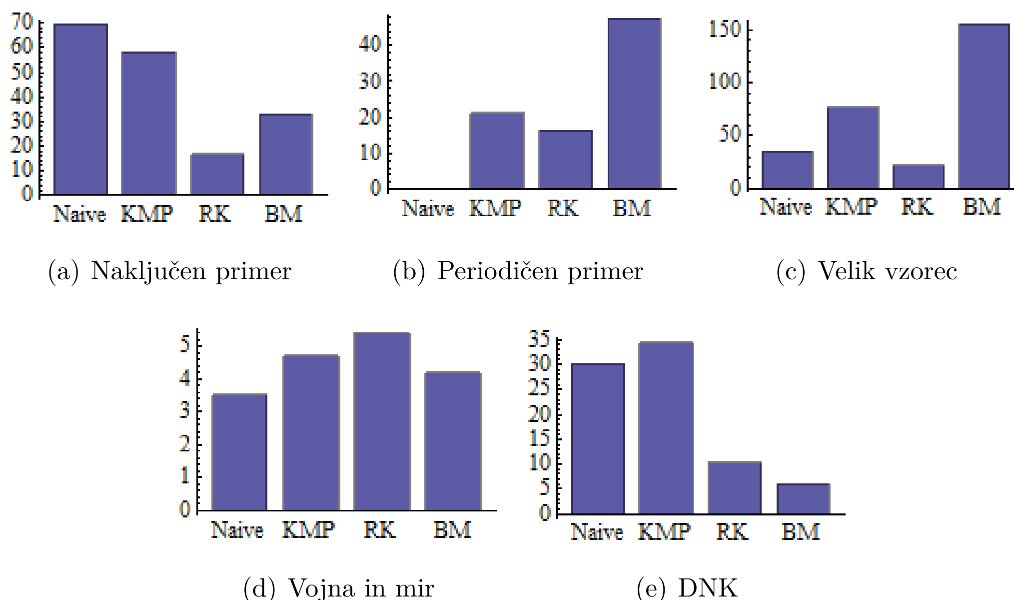
---

[2]Acaryochloris marina, http://www.genome.jp/dbget-bin/www_bget?refseq:NC_009925

Tabela 2.7: Struktura testnih primerov

primer	m	n	besedilo	Σ
(a)	20	10^7	$\{a,b\}^*$	$\{a,b\}$
(b)	1000	10^7	$(ab)^*$	$\{a,b\}$
(c)	$n/2$	10^7	$\{a,b\}^*$	$\{a,b\}$
(d)	9	$3.4 * 10^6$	Vojna in mir	ASCII
(e)	300	$6.5 * 10^6$	genom	$\{a,c,g,t\}$

Slika 2.8: Primerjava časov iskanja



(a) Naključen primer (b) Periodičen primer (c) Velik vzorec

(d) Vojna in mir (e) DNK

2.6 Aho-Corasick

Algoritem Aho-Corasick se nahaja nekje med zaporednimi in vzporednimi algoritmi. Izvajanje algoritma je zaporedno, vendar algoritem sočasno išče pojavitve cele množice vzorcev v besedilu. Alfred V. Aho in Margaret J. Corasick sta leta 1975 razvila algoritem, ki za vsak vzorec iz dane množice poišče vse njegove pojavitve v besedilu, čas iskanja pa je neodvisen od velikosti te množice vzorcev, ki jih iščemo v besedilu. Množica vzorcev se namreč obdela še

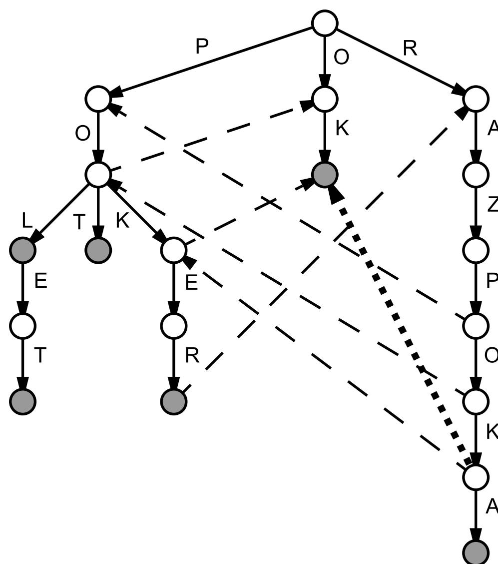
pred fazo iskanja, rezultat te obdelave pa lahko uporabimo na različnih besedilih. Dober primer uporabe bi bil razvrščanje besedil glede na to, katero izmed množice ključnih besed vsebujejo. Uporablja se tudi pri iskanju vzorcev z nedoločenimi znaki. Če želimo v besedilu poiskati vse pojavitve vzorca "avt*", lahko ta vzorec razširimo v množico, kjer se bodo nahajali vzorci "avto", "avta" ... Nedoločeni znaki v vzorcih predstavljajo veliko oviro klasičnim algoritmom za iskanje nizov z linearno časovno zahtevnostjo, saj se le-ti opirajo na tranzitivnost ujemanja nizov, ki pa ne velja, ko vpeljemo nedoločene znake: avto = avt∗, avt∗ = avta, avto ≠ avta.

Problem iskanja k vzorcev s skupno dolžino m v besedilu dolžine n lahko rešimo tako, da poiščemo v besedilu vsak vzorec posebej, kar ima časovno zahtevnost $O(m+k*n)$. Algoritem AC to doseže v času $O(m+n+p)$, kjer je p število najdenih pojavitev vzorcev v besedilu. Omeniti je potrebno, da je lahko p reda $O(k*n)$. V tem primeru algoritem AC ni nič boljši od ločenih iskanj vzorcev. Tak primer bi bil iskanje množice vzorcev $\{a^i \big| 1 \leq i \leq x\}$ v besedilu oblike a^*. Večinoma pa je število pojavitev precej manjše. Algoritem AC je posplošitev algoritma KMP za sočasno iskanje več vzorcev. Če uporabljamo algoritem AC z enim samim vzorcem, je enakovreden algoritmu KMP.

V prvem koraku algoritma si pripravimo slovar vseh vzorcev v obliki drevesa (angl. trie). Vsaka povezava v drevesu ustreza enemu znaku, pot od korena do lista pa opiše enega od vzorcev. Vzorci se lahko pojavijo tudi sredi poti, zato so vozlišča, kjer se konča nek vzorec, posebej označena. Slika 2.9 prikazuje drevo, ki ga dobimo, če vanj vstavimo množico vzorcev {ok, poker, pol, polet, pot, razpoka}. Trenutno so relevantne le navadne povezave, pomen črtkanih in pikastih povezav bomo pojasnili kasneje. Vzorec vstavimo v drevo tako, da od korena sledimo povezavam, ki ustrezajo trenutnemu znaku v vzorcu. Če take povezave ni, jo dodamo. Vsako vozlišče v drevesu mora vsebovati povezave na naslednja vozlišča. Te bi lahko hranili kar v seznamu, bolj učinkovita rešitev pa bi bila z uporabo razpršenih tabel. Ključ v tabeli bi bil znak iz abecede, vrednost pa vozlišče, do katerega pridemo po povezavi. Drevo lahko torej zgradimo v času $O(m)$.

Vsako vozlišče v drevesu določa niz, ki je sestavljen iz črk na povezavah od korena do vozlišča. Vsak tak niz predstavlja predpono enega ali več vzorcev. S črtkanimi črtami so označene povezave, ki vodijo do najdaljše pripone niza, ki ga določa vozlišče. Podobno kot pri algoritmu KMP jih bomo poimenovali *povezave delnih ujemanj*. Če vozlišče določa niz "primer", bi dodali črtkano povezavo do niza "rimer", če ta v drevesu ne obstaja pa do niza "imer" itd. To lahko storimo hitreje, kot da za vsako pripono preverjamo, ali obstaja v drevesu. Niz skrajšamo za en znak in sledimo povezavam delnih ujemanj,

Slika 2.9: Iskalno drevo algoritma Aho-Corasick



dokler ne prispemo do vozlišča, iz katerega vodi povezava, ki ustreza zadnji črki niza. Sedaj lahko dodamo povezavo do vozlišča, h kateremu vodi ta povezava. Če takega vozlišča ni, dodamo povezavo do korena drevesa. Te povezave do korena na sliki 2.9 niso prikazane zaradi boljše preglednosti. Povezave delnih ujemanj lahko računamo po naraščajočih globinah vozlišč, iz katerih izhajajo. Iz vozlišč na globini 1 vodijo te povezave do korena. Nato izračunamo delna ujemanja za vozlišča na globini 2, itd. Tak način dodajanja povezav delnih ujemanj ima linearno časovno zahtevnost.

V fazi iskanja začnemo na začetku besedila in v korenu drevesa. Na vsakem koraku se poskušamo premakniti navzdol po drevesu po povezavi, ki ustreza trenutni črki v besedilu. Če taka povezava ne obstaja, se premaknemo po povezavi delnega ujemanja in poskusimo znova. Če se na tak način vrnemo do korena in tudi tam ne obstaja povezava s pravim znakom, ostanemo v korenu in nadaljujemo z naslednjim znakom v besedilu. Kadar dosežemo vozlišče v drevesu, ki ustreza koncu katerega od iskanih vzorcev, smo našli njegovo pojavitev v besedilu. Vendar to še ni dovolj. Če je nek vzorec podniz drugega, bi lahko zgrešili kakšno pojavitev. Recimo, da v prikazanem primeru (Slika 2.9) v drevesu prispemo do stanja "razpok". Kljub temu, da to ni končno vozlišče katerega od vzorcev, pa smo našli vzorec "ok". V splošnem bi morali vsakič

preveriti, ali obstaja pripona niza, ki ga določa trenutno vozlišče, ki je enaka kateremu od vzorcev. To je enakovredno postopku, da sledimo povezavam delnih ujemanj in izpišemo vsa končna vozlišča, ki jih dosežemo na poti. Da nam ni treba vsakič slediti celotni poti, si pomagamo s pikastimi povezavami, ki jih bomo poimenovali *končne povezave*. Končna povezava kaže do prvega končnega vozlišča na poti po povezavah delnih ujemanj. Tako se lahko premikamo samo po končnih vozliščih in jih sproti izpisujemo. Končne povezave lahko izračunamo sočasno s povezavami delnih ujemanj v času $O(m)$. Za podrobnosti glej primer implementacije (Alg. A.4).

Med procesiranjem besedila se v drevesu n-krat premaknemo po povezavi navzdol (stran od korena). Zaradi sledenja povezavam delnih ujemanj pa je nekaj premikov tudi v obratni smeri. Ker nas vsaka taka povezava pripelje bližje korenu, je lahko skupaj takih premikov kvečjemu n. V nasprotnem primeru bi morali končati višje od korena, kar pa ni mogoče. Upoštevati pa je treba še sledenje končnim povezavam za izpis vseh pojavitev. Če smo našli v besedilu p pojavitev vzorcev, potem število operacij zaradi premikov po končnih povezavah zagotovo ne presega števila p. Tako lahko povzamemo, da je časovna zahtevnost celotnega algoritma $O(m + n + p)$.

Poglavje 3

Vzporedni algoritmi

Doslej predstavljeni algoritmi so namenjeni zaporednemu izvajanju na enem procesorju. Danes pa so vse pogostejši računalniki z več procesorji. Trenutno je namreč lažje in predvsem ceneje izdelati dva enaka procesorja, kot pa enega dvakrat bolj zmogljivega. Ker so programi, ki se izvajajo na računalniku med seboj neodvisni, jih lahko izvajamo vzporedno. To seveda ni povsem trivialno, saj kljub vsemu dostopajo do istih virov računalnika. V tem poglavju bomo predstavili možnosti, kako lahko izkoristimo tako arhitekturo računalnikov za pohitritev iskanja nizov.

Najprej se moramo vprašati, koliko sploh lahko pridobimo na hitrosti z vzporednim izvajanjem. Na to nam odgovori Amdahlov zakon. Če predpostavimo, da je delež ukazov, ki jih je možno paralelizirati, enak P, preostalih $1 - P$ pa moramo izvesti zaporedno, lahko izračunamo faktor pohitritve pri uporabi N procesorjev.

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}} \tag{3.1}$$

$$\lim_{N\to\infty} S(N) = \frac{1}{1-P} \tag{3.2}$$

Največja možna pohitritev je torej $\frac{1}{1-P}$. Izboljšamo pa jo lahko samo z drugačno zasnovo algoritma, ki bo imela večji delež ukazov, ki jih lahko izvajamo vzporedno. Ta izračun seveda ne upošteva številnih težav, ki jih prinese vzporedno izvajanje. Komunikacija in sinhronizacija vzporednih procesov namreč prinašata precej dodatnega dela (t. i. režije).

Kot model računanja bomo uporabili PRAM (Parallel Random Access Machine), ki je najbolj podoben današnjim večjedrnim računalnikom. Model PRAM predpostavlja, da imajo vsi procesorji dostop do skupnega pomnilnika v

konstantnem času. Pri večjem številu procesorjev je ta poenostavitev nekoliko problematična, saj dostop do spomina predstavlja ozko grlo. Model PRAM se deli naprej glede na sočasnost dostopa do pomnilnika pri branju in pisanju. Uporabljali bomo Concurrent Read Exclusive Write model (CREW), ki omogoča sočasno branje, ne pa tudi sočasnega zapisovanja na isto mesto v pomnilniku. Ostali modeli so še EREW, ERCW in najbolj splošen CRCW. Omenimo še, da najšibkejši model (EREW) lahko simulira najmočnejšega (CRCW) v času, ki je kvečjemu $O(\log p)$-krat večji od časa, ki ga rabi CRCW. Tu je p število uporabljenih procesorjev v EREW oz. CRCW.

Pri vzporednih algoritmih sta pomembna predvsem stopnja paralelizacije in skupna količina opravljenega dela. Stopnja paralelizacije predstavlja mejo, do katere lahko razbijemo nek problem in ga rešujemo vzporedno. Razbitje problema običajno ni optimalno, kar pomeni, da vsi procesorji skupaj opravijo več dela, kot bi ga opravil zaporeden algoritem na enem procesorju.

V nadaljevanju je predstavljenih nekaj možnosti paralelizacije iskanja nizov v besedilih, ki so uporabne v različnih primerih podatkov, s katerimi imamo opravka. Opisi paralelizacije so opremljeni s primeri implementacije z uporabo knjižnice za večprocesno programiranje OpenMP. Uporabljena je le najbolj osnovna funkcionalnost knjižnice. Vsa paralelizacija je implementirana na sledeč način:

Algoritem 3.1: Paralelizacija z OpenMP

```
1 omp_set_num_threads(p);
2 #pragma omp parallel for
3 for (int proc=0; proc<p; proc++) {
4     /* Delo, ki naj ga opravi nit proc */
5 }
```

Z ukazom omp_set_num_threads(p) nastavimo število niti, ki jih bomo uporabljali za vzporedno izvajanje. “#pragma omp parallel for” pa pove prevajalniku, naj razdeli iteracije, ki jih opravi sledeča zanka, med niti. Lahko bi določili tudi način delitve iteracij med niti. Ker pa je število iteracij enako številu niti in zato precej majhno, lahko ostanemo kar pri privzetem načinu.

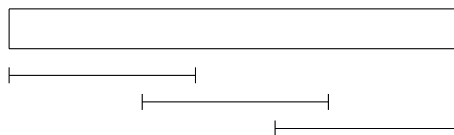
3.1 Delitev besedila

Najbolj naraven pristop je, da besedilo razdelimo na več enako velikih kosov in v vsakem kosu vzporedno iščemo pojavitve vzorca. Če se za začetek posvetimo naivnemu algoritmu za iskanje nizov, gremo lahko tako daleč, da vsako primerjavo znakov izvajamo na svojem procesorju. Na CRCW PRAM modelu bi

to pomenilo, da lahko z $n * m$ procesorji rešimo problem v konstantnem času. Taka paralelizacija je s stališča opravljenega dela optimalna, saj vsi procesorji skupaj opravijo točno toliko dela, kot bi ga en sam. Naivni algoritem je torej mogoče optimalno paralelizirati. Taka rešitev je v večini praktičnih primerov, ko v dolgih besedilih iščemo kratke vzorce, povsem sprejemljiva.

Tehniko delitve besedila na kose lahko uporabimo tudi za pohitritev linearnih algoritmov kot npr. KMP. Vzorec se lahko začne v besedilu na $n-m+1$ različnih mestih. Preverjanje, ali se vzorec res pojavi na nekem mestu, pa bomo enakomerno razdelili med p procesorjev. Da bi ugotovili, na katerih mestih med vključno i_1 in i_2 se začne vzorec, mora algoritem KMP pregledati še dodatnih $m-1$ znakov za mestom i_2. Vsak procesor bo obdelal $\frac{n-m+1}{p}+(m-1)$ znakov. Časovna zahtevnost pri izvajanju na p procesorjih je torej $O(\frac{n}{p}+m)$. Primer implementacije je na voljo v dodatku (Alg. A.5).

Slika 3.1: Delitev besedila med 3 procesorje



Analiza iz prejšnjega odstavka razkrije glavne probleme paralelizacije z delitvijo besedila. Pri neskončno procesorjih bi bila časovna zahtevnost faze iskanja $O(m)$. Pred razbitjem besedila ne smemo pozabiti na fazo inicializacije, kjer izračunamo funkcijo delnih ujemanj. Ta izračun je povsem zaporedne narave in ga v taki obliki ne moremo paralelizirati. S skice 3.1 je tudi razvidno, da nekatere znake v besedilu obdela več procesorjev. Vsi skupaj opravijo več dela, kot bi ga zaporeden algoritem na enem procesorju, zato taka paralelizacija ni optimalna. Kljub težavam pri izjemno dolgih vzorcih in neoptimalnosti paralelizacije, predstavlja ta pristop velik napredek v primerjavi s paralelizacijo naivnega algoritma.

3.2 Razširitev abecede

Ideja za razširitvijo abecede leži v tem, da bi morda lahko dosegli dvakratno pohitritev iskanja, če bi primerjali po dva znaka hkrati. Za primer vzemimo, da v besedilu b = 00**1010**11**0100**0 iščemo vzorec v = **1010**. Zaporedne pare znakov lahko predstavimo s črkami (00 = A, 01 = B, 10 = C, 11 = D) in

tako dobimo besedilo $b' =$ ACCDBA in vzorec $v' =$ CC. Iz primera je očitno, da na ta način zgrešimo vse pojavitve vzorca na lihih mestih $(b_1, b_3 \ldots)$. To rešimo tako, da pripravimo še eno strnjeno kopijo besedila, ki se začne šele pri znaku b_1 namesto pri b_0 in dobimo $b'' =$ BBBCC. Na prvi pogled nismo pridobili nič, saj moramo iskati vzorec v dveh besedilih b' in b'', ki sta pol krajši od originalnega. Hitrosti iskanja res nismo zmanjšali, se pa sedaj ponuja možnost za učinkovito vzporedno iskanje, ki bo neodvisno od dolžine vzorca. Vzporedno iskanje na dveh procesorjih z metodo delitve besedila, ki je opisana v prejšnjem poglavju, razdeli morebitne pojavitve vzorca v besedilu na prvo in drugo polovico, postopek z razširitvijo abecede pa bi jih razdelil na liha in soda mesta.

Če želimo izvajati iskanje na p procesorjih, bi morali v nov znak združiti zaporedja p znakov. Pri tem pravzaprav izkoriščamo paralelnost aritmetično logične enote, ki je sposobna v eni operaciji primerjati več kot zgolj en bit. Kljub temu pa hitro dosežemo točko, kjer postane število znakov v razširjeni abecedi večje od števil, ki jih je procesor sposoben primerjati v eni operaciji. Namesto identifikacijskih številk znakov v razširjeni abecedi lahko izračunamo njihove prstne odtise na podoben način kot pri algoritmu Rabin-Karp z uporabo "rolling hash" prstnih odtisov (Alg. 3.2). V besedilu moramo torej izračunati $n-p+1$ prstnih odtisov $h_i = hash(h_i h_{i+1} \ldots h_{i+p-1})$. Vsakemu procesorju dodelimo približno enako velik strnjen interval prstnih odtisov, ki naj jih izračuna. Pri tem naletimo na podoben problem kot pri razbitju besedila za vzporedno iskanje z linearnimi algoritmi. Pride namreč do prekrivanja med znaki, ki jih obravnavajo različni procesorji. Na ta način lahko prstne odtise izračunamo v času $O(n/p + p)$.

Algoritem 3.2: Prstni odtisi znakov v razširjeni abecedi

```
1  int hf=37;
2  int pos=n-p+1;
3  #pragma omp parallel for
4  for (int proc=0; proc<p; proc++) {
5     int start=proc*pos/p, end=(proc+1)*pos/p;
6     int hash=0, hp=1;
7     for (int i=1; i<=p-1; i++) hp*=hf;
8     for (int i=0; i<p-1; i++) hash=hash*hf+b[start+i];
9     for (int i=start; i<end; i++) {
10       hash=hash*hf+b[i+p-1];
11       h[i]=hash;
12       hash-=b[i]*hp;
13    }
14 }
```

Izračun prstnih odtisov vzorca $vh_i = hash(v_{i*p}v_{i*p+1} \ldots v_{i*p+p-1})$ je še enostavnejši, saj potrebujemo le prstne odtise na mestih, ki so večkratniki števila procesorjev. V povezavi z vzorcem pa je potrebno rešiti neko drugo težavo. Dolžina vzorca bo le redko deljiva s številom procesorjev. Vzorcu lahko odrežemo pripono, ki je po dolžini enaka ostanku pri deljenju dolžine vzorca m s številom procesorjev, in pojavitve te pripone poiščemo kar z metodo delitve besedila. Dolžina te pripone bo krajša od števila procesorjev, zato bo časovna zahtevnost iskanja $O(n/p+p)$. Ko najdemo pojavitve predpone, ki je deljiva s številom procesorjev, in kratke pripone, moramo rezultate še združiti. Poiskati moramo vsa mesta, kjer se pripona in predpona pojavita na pravilni oddaljenosti. V primeru implementacije (Alg. 3.3) se pojavitve pripone shranjujejo v pomožno tabelo *tmp*.

Algoritem 3.3: Ločeno iskanje pripone

```
1  int m1=m-m%p, m2=m%p;
2  parallel(p,v,b,r,m1,n,vh,h,0);
3  kmp_parallel(p,v+m1,b,tmp,m2,n);
4
5  #pragma omp parallel for
6  for (int proc=0; proc<p; proc++) {
7     int start=proc*n/p, end=(proc+1)*n/p;
8     for (int i=start; i<end; i++) {
9        if (i+m1<n) r[i]&=tmp[i+m1];
10       else r[i]=0;
11    }
12 }
```

Ko imamo izračunane prstne odtise, lahko pričnemo z iskanjem. i-ti procesor bo iskal vzorec vh v besedilu oz. zaporedju, ki je sestavljeno iz vsakega p-tega prstnega odtisa $(h_i, h_{i+p}, h_{i+2*p}, \ldots)$. Zaporedje prstnih odtisov lahko preuredimo tako, da bo vsak procesor iskal v strnjenem intervalu. Druga možnost pa je, da popravimo zaporedni algoritem za iskanje nizov, tako da se bo v zaporedju prstnih odtisih premikal po p mest naenkrat (Alg. 3.4).

Algoritem 3.4: Vzporedno iskanje z razširjeno abecedo

```
1  #pragma omp parallel for
2  for (int proc=0; proc<p; proc++) {
3     kmp_serial(vh, h+proc, r+proc, m/p, n-p+1, p);
4  }
```

To metodo paralelizacije iskanja nizov lahko uporabimo na poljubnem algoritmu za iskanje nizov z neomejeno abecedo. Metoda je enostavna in ne povzroči veliko dodatnega dela. Glavna slabost se pokaže v količini dodatnega

pomnilnika, ki je potreben za shranjevanje prstnih odtisov vzorca vh in besedila h. S takim načinom paralelizacije lahko rešimo problem iskanja niza v času $O(\sqrt{n})$, če imamo na voljo $O(\sqrt{n})$ procesorjev. Ker različni procesorji nikoli ne zapisujejo vrednosti na isto mesto, je algoritem primeren za izvajanje na modelu CREW. Poleg tega pa z istega mesta bereta kvečjemu dva procesorja. Če torej pred začetkom izvajanja algoritma pripravimo dodatno kopijo besedila in primerno razporedimo procesorje, ki berejo z istih mest, med obe kopiji besedila, bi algoritem deloval tudi na najosnovnejšem računskem modelu EREW.

3.3 Meritve

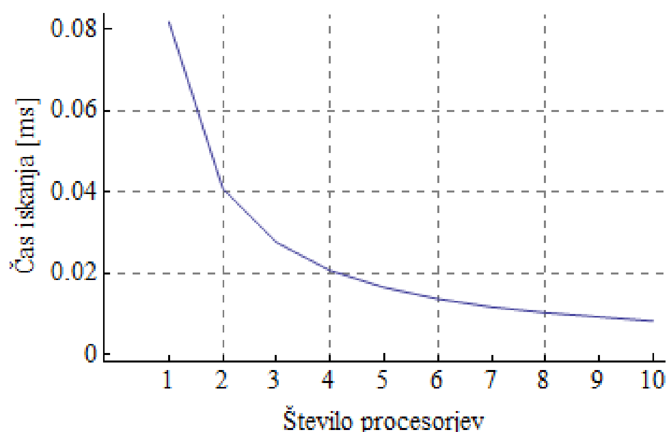
Obe metodi paralelizacije smo preizkusili tudi v praksi. Meritve časa iskanja z algoritmom KMP smo izvedli na računalniku s 4 procesorji Intel Xeon Processor X7460 in 128 GB delovnega spomina. To nam je omogočilo eksperimentiranje z do 24 vzporednimi nitmi izvajanja.

Uporaba večjedrnega procesorja kot približka modela CREW PRAM je v tem primeru malce problematična. Čas, ki se porabi za ustvarjanje več niti izvajanja, ni zanemarljiv, zato bi potrebovali čim večje testne podatke. Po drugi strani pa so algoritmi za iskanje nizov dokaj nezahtevni s strani aritmetičnologičnih operacij. Večinoma je ozko grlo v hitrosti dostopa do pomnilnika. Da bi se izognili upočasnitvi zaradi dostopov do skupnega pomnilnika, bi morali imeti čim manjše podatke, da jih lahko posamezne procesne enote naložijo v predpomnilnik in ne obremenjujejo skupnega pomnilnika.

Kot kompromis med zgoraj opisanima problemoma smo postopek paralelizacije z delitvijo besedila testirali na naključnih nizih iz znakov ’a’ in ’b’. Dolžina besedila je bila 10^7, dolžina vzorca pa 100. Ker je čas izvajanja pri taki velikosti podatkov izjemno majhen, smo iskanje ponovili večkrat in izračunali povprečen čas izvajanja (Slika 3.2). Iz rezultatov je razvidno, da pri kratkem vzorcu čas iskanja enakomerno pada s številom procesorjev.

Pri izjemno dolgih vzorcih pa je zgodba drugačna. V enako dolgem besedilu smo iskali še vzorec dolžine $5.4 * 10^6$ in primerjali hitrost padanja časov iskanja pri obeh načinih paralelizacije. Ker nas zanima padanje časov pri večanju števila procesorjev in ne absolutni časi iskanja, smo izbrali tako dolžino vzorca, ki je deljiva s števili od 1 do 10. V teh primerih ima metoda z razširitvijo abecede namreč manj dodatnega dela in so časi iskanja bolj primerljivi z metodo delitve besedila (Slika 3.3). Zaradi velikega vzorca je paralelizacija z delitvijo besedila neučinkovita. Pravzaprav se čas izvajanja še podaljša zaradi dodatnega dela

Slika 3.2: Paralelizacija z delitvijo besedila

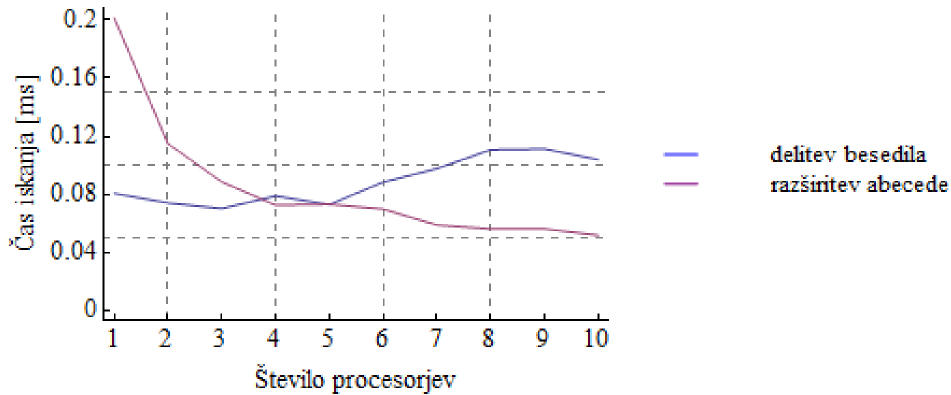


z ustvarjanjem vzporednih niti. Metoda razširitve abecede je neodvisna od dolžine vzorca. Sicer opravi nekaj več dela, vendar jo lahko uspešno paraleliziramo z do $O(\sqrt{n})$ procesorji. Na današnjih večjedrnih računalnikih je ta meja precej nižja, z bolj namensko zasnovano arhitekturo računalnika pa bi lahko dosegli tudi teoretično mejo.

3.4 Vishkinov algoritem

V poglavju 3.2 smo pokazali, kako lahko rešimo problem iskanja niza v času $O(\sqrt{n})$ pri uporabi računskega modela EREW PRAM. Pri tem se postavi vprašanje, kakšna je meja vzporedne časovne zahtevnosti, ki je seveda ne moremo preseči ne glede na število procesorjev. Problem lahko enostavno rešimo v času $O(1)$ z $O(n * m)$ procesorji, vendar le pri modelu CRCW, ki omogoča sočasno zapisovanje vrednosti na isto pomnilniško lokacijo. Pri računskem modelu CREW pa je ta meja pri času $O(\log n)$. To lahko dokažemo s prevedbo problema računanja logične konjunkcije na problem iskanja niza v besedilu. Konjunkcija n elementov ima vrednost 1 natanko takrat, ko so vsi elementi enaki 1. Konjunkcijo lahko torej izračunamo tudi z iskanjem zaporedja n enic v zaporedju n elementov. Če bi lahko problem iskanja nizov rešili hitreje od $O(\log n)$, bi lahko ta algoritem uporabili za hitrejši izračun konjunkcije. Ker pa konjunkcije z uporabo modela CREW ni mogoče izračunati hitreje kot v času $O(\log n)$ [6], velja ta spodnja meja tudi za problem iskanja nizov. Galil[7]

Slika 3.3: Paralelizacija z razširitvijo abecede



je opisal algoritem, ki doseže čas $O(\log n)$ na modelu CRCW za nize z omejeno abecedo, Vishkin[4, 12] pa ga je nato posplošil na neomejeno abecedo (Tabela 3.1). Njegov algoritem bomo podrobneje opisali v nadaljevanju poglavja. Karp in Rabin[4, 8] sta objavila Monte Carlo algoritem, ki pristop z uporabo prstnih odtisov uspešno prilagodi vzporednemu izvajanju. Leta 1990 sta Breslauer in Galil[3] dosegla teoretično spodnjo mejo časovne zahtevnosti $O(\log \log n)$. Istega leta je Vishkin[13] objavil algoritem s še boljšo časovno zahtevnostjo, ki pa za razliko od prej omenjenih algoritmov ni vključevala časa predobdelave vzorca.

Tabela 3.1: Razvoj vzporednih algoritmov

avtor	leto	časovna zahtevnost
Galil	1984	$O(\log n)$
Vishkin	1985	$O(\log n)$
Karp, Rabin	1987	$O(\log n)$
Breslauer, Galil	1990	$O(\log \log n)$
Vishkin	1990	$O(\log^* n)$

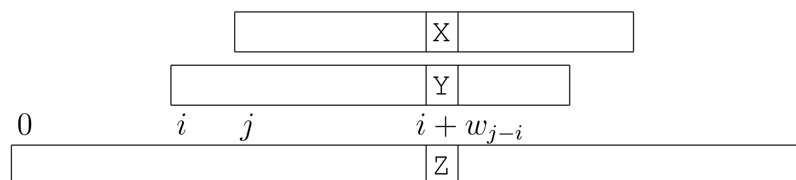
Recimo, da se vzorec začne s znakom 'a', ki pa se v besedilu pojavi le na k mestih. Če je število mest k dovolj majhno ($k \leq \frac{n}{m}$), lahko rešimo

problem v konstantnem času z $k * m = O(n)$ procesorji. Število mest, kjer se v besedilu nahaja vzorec, pa je lahko zelo veliko. Vzorec in besedilo sta lahko sestavljena npr. iz samo enega različnega znaka. V tem primeru je število pojavitev vzorca reda $O(n)$. Kadar se vzorec pojavi v besedilu več kot $\frac{n}{m}$-krat, se določene pojavitve med seboj zagotovo prekrivajo, torej mora biti vzorec periodične oblike. Problem je precej bolj obvladljiv v primeru neperiodičnih vzorcev, kjer lahko omejimo število pojavitev vzorca v besedilu. Zato bomo periodične in neperiodične vzorce obravnavali ločeno.

Niz u je *perioda* niza v natanko takrat, ko je v predpona niza u^i. Pri obravnavi vzorca v nas bo vedno zanimala njegova najkrajša perioda u z dolžino z. Vzorcu bomo rekli periodičen, če je več kot dvakrat daljši od svoje najkrajše periode ($m > 2z$).

3.4.1 Neperiodičen vzorec

Slika 3.4: Dvoboj



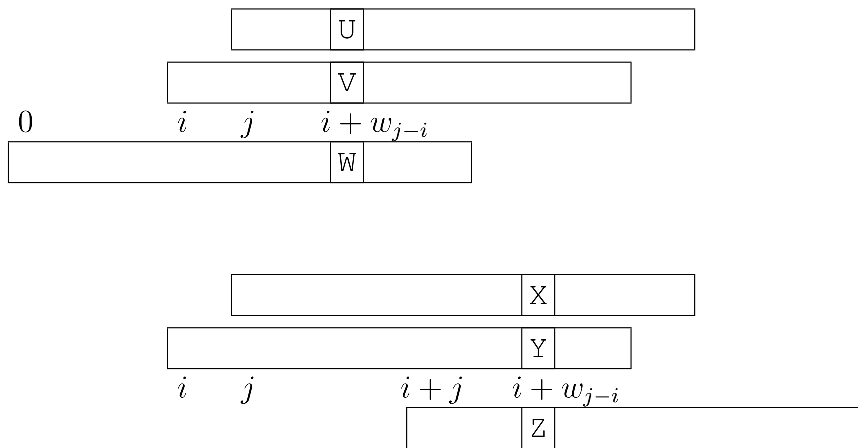
Recimo, da primerjamo vzorca, ki sta med seboj zamaknjena za $k < z$ znakov. Ker je dolžina najkrajše periode vzorca enaka z, bo pri taki poravnavi zagotovo prišlo do neujemanja. Mestu, kjer se za k znakov zamaknjena vzorca razlikujeta, pravimo *priča* razdalje k in jo označimo z w_k. Vishkinov algoritem v prvi fazi izračuna priče za vse razdalje, ki so krajše od periode vzorca. V drugi fazi pa uporabi to informacijo za sistematično izločanje potencialnih mest v besedilu, kjer bi se lahko pojavil vzorec. Če se v besedilu pojavita dve potencialni mesti i in j na razdalji manj kot z, lahko eno od njiju zagotovo izločimo z *dvobojem*: ker za tako poravnavo vzorcev obstaja priča w_{j-i}, se znaka X in Y v vzorcu zagotovo razlikujeta in zato se kvečjemu en od njiju ujema s pripadajočim znakom Z v besedilu (Slika 3.4).

Faza izločanja potencialnih mest v besedilu poteka v $\log m$ korakih. V i-tem koraku je besedilo razdeljeno na intervale velikost 2^{i-1}, vsak od njih pa vsebuje le eno potencialno mesto. V posameznem koraku združimo sosednje pare intervalov $2j$ in $2j + 1$ z dvobojem potencialnih mest. To lahko ponavljamo, dokler velikost intervalov ne preseže polovice periode. Takrat nam

ostane $\frac{n}{z/2}$ potencialnih mest, ki jih lahko preverimo z naivnim algoritmom in pri tem opravimo le $\frac{n}{z/2} * m = O(n)$ primerjav znakov. Pri tem smo upoštevali, da je vzorec neperiodičen.

3.4.2 Izračun prič

Slika 3.5: Priče



Preden lahko začnemo z izločanjem mest pa moramo izračunati priče za vse zamike, ki so krajši od periode vzorca. Izkoristili bomo lastnost, ki pravi, da če poznamo pričo zamika $j - i$, lahko izračunamo pričo za zamik i ali pa za j. Poleg vzorca si lahko predstavljamo še dve kopiji, ki sta v primerjavi z njim zamaknjeni za i in j mest. Glej zgornji primer na sliki 3.5. Ker smo predpostavili, da obstaja priča w_{j-i}, se znaka U in V razlikujeta in se zato kvečjemu en ujema z znakom W.

$$V \neq W \Rightarrow w_i = i + w_{j-i} \tag{3.3}$$

$$U \neq W \Rightarrow w_j = i + w_{j-i} \tag{3.4}$$

Težava pa nastane, če se priča pojavi preveč proti koncu vzorca ($i + w_{j-i} \geq m$). Vzorec lahko premaknemo za $i + j$ znakov v desno glede na obe kopiji vzorca, kar je prikazano na spodnjem delu slike 3.5. Pri tem pride do spremembe samo v tem, katera od kopij je zamaknjena za i in katera za j znakov glede

na vzorec.

$$X \neq Z \Rightarrow w_i = i + w_{j-i} - j \quad (3.5)$$
$$Y \neq Z \Rightarrow w_j = w_{j-i} \quad (3.6)$$

Izračun prič poteka na podoben način kot kasnejše izločanje potencialnih mest v $\log m$ korakih. V i-tem koraku je vzorec razdeljen na intervale velikosti 2^{i-1}, vsak od njih pa vsebuje le eno manjkajočo pričo. Prvi interval ima ves čas neizračunano pričo w_0. Neizračunano pričo v drugem intervalu lahko poiščemo z naivnim algoritmom v konstantnem času z $O(m)$ procesorji. Če ta priča ne obstaja, smo našli periodo vzorca in lahko zaključimo s fazo računanja prič. V nasprotnem primeru pa imamo sedaj izračunane priče $w_1, w_2, \ldots, w_{2^i-1}$, ki jih lahko uporabimo za dvoboje med ostalimi neizračunanimi pričami v sosednjih parih intervalov. Te korake ponavljamo, dokler velikost intervalov ne preseže polovice dolžine vzorca. Vrednosti $w_i, i > \frac{m}{2}$ namreč ne potrebujemo za kasnejše izločanje potencialnih mest v besedilu. S tem se tudi ognemo težavam, ki se pojavijo, kadar vzorec nima dolžine enake potenci števila 2.

3.4.3 Periodičen vzorec

Sedaj se posvetimo še periodičnim vzorcem, ki so oblike $v = u^k t$ ($k \geq 2$, t je predpona periode u). Niz u^2 ima enako najmanjšo periodo kot vzorec v. Oba imata periodo dolžine z, dokažemo pa lahko, da če bi imel u^2 krajšo periodo, bi jo imel tudi v. S postopkom za iskanje neperiodičnih vzorcev poiščemo vsa mesta, kjer se v besedilu pojavi u^2. Ker je teh mest kvečjemu $\frac{n}{z}$ in so med seboj vsaj z mest narazen, lahko z naivnim algoritmom preverimo katerim izmed njih sledi t. Tako najdemo vse pojavitve niza $u^2 t$. Sedaj moramo le še prešteti zaporedne pojavitve. Pojavitvi niza sta zaporedni, če se nahajata na mestih i in $i+z$. Če se jih zapored pojavi r, obstaja na tem območju $r-k+2$ pojavitev vzorca v.

Ena izmed možnosti, kako prešteti zaporedne pojavitve, je z izgradnjo polnega binarnega drevesa nad besedilom. V vsaki točki drevesa hranimo informacijo o dolžini predpone in pripone, ki sta sestavljeni iz zaporednih pojavitev niza $u^2 t$ v intervalu, ki ga pokriva poddrevo. Poleg dolžin hranimo tudi odmik najbolj stranske pojavitve od robu intervala. Ta odmik bo vedno manjši od z. Podatke lahko računamo od listov proti vrhu drevesa. Kadar ima levo poddrevo pripono x mest od desnega roba in desno poddrevo predpono $z-1-x$ mest od levega roba, ju lahko združimo in s tem dobimo strnjen interval zaporednih pojavitev. Podobno moramo obravnavati primere, kjer se pojavitve raztezajo čez celoten interval.

Poglavje 4

Sklepne ugotovitve

V tem delu smo predstavili različne algoritme za iskanje nizov in možnosti za vzporedno reševanje tega problema. Meritve so pokazale, da se enostavni pristopi v praksi obnesejo dobro. Zelo se je potrebno potruditi, da najdemo izrojene primere, kjer enostavni pristopi odpovejo. V okviru zaporednih algoritmov lahko s kompleksnejšimi pristopi učinkovito rešimo tudi izrojene primere. Pri vzporednem reševanju pa so se izrojeni primeri izkazali za precej trši oreh. Teoretično lahko tudi te rešimo optimalno, meritve pa so pokazale drugačno sliko. Tak izid meritev lahko delno pripišemo dodatnem delu, ki je potrebno pri kompleksnejših načinih paralelizacije. Glavna krivda pa se skriva v arhitekturi večjedrnih računalnikov, ki ni povsem primerljiva z modelom računanja, ki ga uporabljamo v teoretičnih izračunih.

Predstavljeni algoritmi in načini paralelizacije so osnova, ki bi jo lahko poskusili razširiti na druge probleme s področja obdelave nizov, kot npr. ujemanje nizov z nekaj dovoljenimi napakami, iskanje najdaljšega skupnega podniza itd. Poleg tega bi lahko poizkuse izvedli na grafičnih karticah, ki so namenjene izračunom z visoko stopnjo vzporednosti. Na njih se običajno izvajajo računsko zahtevne operacije za prikaz slike, morda pa bi arhitektura koristila tudi vzporednim algoritmom za iskanje nizov.

Dodatek A

Implementacije algoritmov

Algoritem A.1: Algoritem Knuth-Morris-Pratt

```
1  void KnuthMorrisPratt(char v[], char b[], char r[], int m, int n)
     {
2     int i,j;
3     int f[m+1]; // funkcija delnega ujemanja
4     f[0]=0; f[1]=0;
5     i=2; j=0;
6     while (i<=m) {
7        if (v[j]==v[i-1]) { j++; f[i]=j; i++; }
8        else if (j==0) { f[i]=0; i++; }
9        else { j=f[j]; }
10    }
11    // faza iskanja
12    i=0; j=0;
13    while (i<n) {
14       if (v[j]==b[i]) {
15          j++; i++;
16          if (j==m) { j=f[j]; r[i-m]=1; }
17       } else if (j==0) { i++; }
18       else { j=f[j]; }
19    }
20 }
```

Algoritem A.2: Algoritem Boyer-Moore

```
1  void BoyerMoore(char v[], char b[], char r[], int m, int n) {
2     /** poravnava znaka */
3     vector<int> t[256];
4     for (int i=m-1;i>=0;i--) t[v[i]].push_back(i);
5     for (int i=0;i<256;i++) t[i].push_back(-1);
6
7     /** poravnava pripone */
8     for (int i=0;i<m-1-i;i++) swap(v[i],v[m-1-i]); // obrni vzorec
9     int P[m]; // tabela skupnih predpon
10    P[0]=0;
11    int s=0;
12    for (int i=1;i<m;i++) {
13       int k=i-s;
14       if (k<i && i+P[k]<s+P[s]) P[i]=P[k];
15       else {
16          int x=max(0,s+P[s]-i);
17          while (i+x<m && v[x]==v[i+x]) x++;
18          P[i]=x;
19          s=i;
20       }
21    }
22    for (int i=0;i<m-1-i;i++) swap(v[i],v[m-1-i]); // obrni vzorec
23
24    int S[m]; // tabela skupnih pripon
25    for (int i=1;i<m;i++) S[m-i]=P[i];
26
27    int L[m+1]; // pomozna tabela L
28    for (int i=1;i<=m;i++) L[i]=-1;
29    for (int j=1;j<m;j++) L[m-S[j]]=j-1;
30
31    int l[m+1]; // pomozna tabela l
32    l[m]=0;
33    for (int i=m-1;i>=1;i--) {
34       l[i]=l[i+1];
35       int j=m-i;
36       if (S[j]==j) l[i]=j;
37    }
38
39    /** iskanje */
40    int k=m-1; // zadnje mesto vzorca v besedilu
41    int gk=-1, gn=0;
42    while (k<n) {
43       int i=m-1, h=k; // kazalca na znak v vzorcu in besedilu
44       while (i>=0 && v[i]==b[h]) {
45          if (h==gk && gn>0) { i-=gn; h-=gn; } // Galilovo pravilo
46          else { i--; h--; }
```

```
47          }
48          gk=k; gn=min(k-h,m);
49          if (i<0) { // ujemanje vzorca in besedila
50             r[k-m+1]=1;
51             k=k+m-l[1];
52          } else { // razlika pri znaku v[i]!=b[h]
53             int k1=k+1, k2=k+1;
54             // pravilo poravnave znaka
55             int j=0;
56             while (t[b[h]][j]>=i) { j++; }
57             k1=k+i-t[b[h]][j];
58             // pravilo poravnave pripone
59             if (L[i+1]!=-1) k2=k+(m-1)-L[i+1];
60             else k2=k+m-l[i+1];
61             // naredimo vecji premik
62             k = max(k1,k2);
63          }
64       }
65    }
```

Algoritem A.3: Algoritem Rabin-Karp

```
1  void RabinKarp(char v[], char b[], char r[], int m, int n) {
2     int k=37; // w=2^32
3     int hv=v[0], h=b[0], f=1; // f=k^(m-1)
4     for (int i=1;i<m;i++) {
5        hv=hv*k+v[i];
6        h=h*k+b[i];
7        f=f*k;
8     }
9     if (hv==h) r[0]=1;
10    for (int i=0;i+1<=n-m;i++) {
11       h=(h-f*b[i])*k+b[i+m];
12       if (h==hv) r[i+1]=1;
13    }
14 }
```

Algoritem A.4: Algoritem Aho-Corasick

```
1  class node {
2  public:
3     int id, len, final;
4     node *go[26], *fail, *out;
5     node() {
6        for (int c=0;c<26;c++) go[c]=0;
7        fail=0; out=0; final=0; id=-1; len=0;
8     }
9  };
10
11 vector<vector<int> > AhoCorasick(vector<string> &v, string &b) {
12    // vstavi besede v slovar
13    node *root = new node();
14    for (int i=0;i<v.size();i++) {
15       node *n = root;
16       for (int j=0;j<v[i].size();j++) {
17          int c=v[i][j]-'A';
18          if (!n->go[c]) {
19             n->go[c] = new node();
20             n->go[c]->len = n->len+1;
21          }
22          n=n->go[c];
23       }
24       n->final = 1;
25       n->id = i;
26    }
27    // izracunaj povezave delnih ujemanj
28    queue<node*> q;
29    for (int c=0;c<26;c++) {
30       if (!root->go[c]) root->go[c]=root;
31       else {
32          root->go[c]->fail=root;
33          q.push(root->go[c]);
34       }
35    }
36    while (!q.empty()) {
37       node *n=q.front(); q.pop();
38       for (int c=0;c<26;c++) if (n->go[c]) {
39          q.push(n->go[c]);
40          node *v = n->fail;
41          while (!v->go[c]) v=v->fail;
42          n->go[c]->fail = v->go[c];
43          if (v->go[c]->final) n->go[c]->out = v->go[c];
44          else n->go[c]->out = v->go[c]->out;
45       }
46    }
```

```
47     // iskanje
48     vector<vector<int> > r = vector<vector<int> >(v.size());
49     node *s=root;
50     for (int i=0;i<b.size();i++) {
51        int c=b[i]-'A';
52        while (!s->go[c]) s=s->fail;
53        s=s->go[c];
54        if (s->final) r[s->id].push_back(i-s->len+1);
55        for (node *o=s->out; o; o=o->out)
              r[o->id].push_back(i-o->len+1);
56     }
57     return r;
58  }
```

Algoritem A.5: Vzporedno iskanje z algoritmom KMP

```
1
2  int* kmp_init(char v[], int m) {
3     int *f = new int[m+1];
4     f[0]=0; f[1]=0;
5     int i=2, j=0;
6     while (i<=m) {
7        if (v[j]==v[i-1]) { j++; f[i]=j; i++; }
8        else if (j==0) { f[i]=0; i++; }
9        else { j=f[j]; }
10    }
11    return f;
12 }
13
14 void kmp_search(char v[], char b[], char r[], int m, int n, int
       *f) {
15    int i=0, j=0;
16    while (i<n) {
17       if (v[j]==b[i]) {
18          j++; i++;
19          if (j==m) { j=f[j]; r[i-m]=1; }
20       } else if (j==0) { i++; }
21       else { j=f[j]; }
22    }
23 }
24
25 void kmp_parallel(int p, char v[], char b[], char r[], int m, int
       n) {
26    omp_set_num_threads(p);
27    int *f = kmp_init(v,m);
28    int pos=n-m+1;
29    #pragma omp parallel for
30    for (int proc=0;proc<p;proc++) {
31       int start=proc*pos/p;
32       int end=(proc+1)*pos/p;
33       kmp_search(v,b+start,r+start,m,end-start+m-1,f);
34    }
35 }
```

Algoritem A.6: Vzporedno iskanje z razširitvijo abecede

```
1
2  void parallel(int p, int v[], int b[], char r[], int m, int n,
       int *vh, int *h, char *tmp) {
3     omp_set_num_threads(p);
4
5     if (m<p) {
6        kmp_parallel(p,v,b,r,m,n);
7
8     } else if (m%p!=0) {
9        int m1=m-m%p, m2=m%p;
10       parallel(p,v,b,r,m1,n,vh,h,0);
11       kmp_parallel(p,v+m1,b,tmp,m2,n);
12
13       #pragma omp parallel for
14       for (int proc=0;proc<p;proc++) {
15          int start=proc*n/p;
16          int end=(proc+1)*n/p;
17          for (int i=start;i<end;i++) {
18             if (i+m1<n) r[i]&=tmp[i+m1];
19             else r[i]=0;
20          }
21       }
22
23    } else {
24       int hf=37;
25       // pattern hash
26       int mp=m/p;
27       #pragma omp parallel for
28       for (int i=0;i<mp;i++) {
29          int hash=0, start=i*p;
30          for (int j=0;j<p;j++) hash=hash*hf+v[start+j];
31          vh[i]=hash;
32       }
33       // text hash
34       int pos=n-p+1;
35       #pragma omp parallel for
36       for (int proc=0;proc<p;proc++) {
37          int start=proc*pos/p;
38          int end=(proc+1)*pos/p;
39          int hash=0, hp=1;
40          for (int i=1;i<=p-1;i++) hp*=hf;
41          for (int i=0;i<p-1;i++) hash=hash*hf+b[start+i];
42          for (int i=start; i<end; i++) {
43             hash=hash*hf+b[i+p-1];
44             h[i]=hash;
45             hash-=b[i]*hp;
```

```
46            }
47         }
48         // search
49         #pragma omp parallel for
50         for (int proc=0;proc<p;proc++) {
51            kmp_serial(vh, h+proc, r+proc, m/p, n-p+1, p);
52         }
53      }
54   }
```

Slike

2.1 Ideja algoritma KMP 4
2.2 Rekurzivna struktura predpon-pripon niza p_i 5
2.3 Preskakovanje znakov 8
2.4 Poravnava znaka 8
2.5 Poravnava pripone 10
2.6 Izračun najdaljših skupnih predpon niza in njegovih pripon . . . 11
2.7 Iskanje najpogostejših besed v knjigi Vojna in mir 12
2.8 Primerjava časov iskanja 16
2.9 Iskalno drevo algoritma Aho-Corasick 18

3.1 Delitev besedila med 3 procesorje 23
3.2 Paralelizacija z delitvijo besedila 27
3.3 Paralelizacija z razširitvijo abecede 28
3.4 Dvoboj 29
3.5 Priče 30

Tabele

2.1 Funkcija delnega ujemanja 5
2.2 Primer faze iskanja z algoritmom KMP 7
2.3 Tabela zadnjih pojavitev znakov v iskanem vzorcu 9
2.4 Primer tabel $L(i)$ in $l(i)$ 11
2.5 Primer iskanja z algoritmom RK 14
2.6 Povzetek zaporednih algoritmov 14
2.7 Struktura testnih primerov 16

3.1 Razvoj vzporednih algoritmov 28

Stvarno kazalo

Ahmdalov zakon, 21
Aho-Corasick, 16

bioinformatika, 1, 15
Boyer-Moore, 7

DNK, 1, 12, 15
drevo, 17
    binarno, 31
    pripon, 1
dvoboj, 29
dvojiško iskanje, 9

funkcija delnega ujemanja, 5

Galilovo pravilo, 11

Horspool, 8

Knuth-Morris-Pratt, 4, 23, 26
kolobar ostankov, 13

Las Vegas, 14

Monte Carlo, 13, 28

Naivno iskanje, 3
nedoločeni znaki, 17

OpenMP, 22

PRAM, 21
    CRCW, 28
    CREW, 22
predpona-pripona, 4
priča, 29
prstni odtis, 13, 24

Rabin-Karp, 13, 24
razpršena tabela, 13, 17
rolling hash, 13

Sustik-Moore, 12

trie, 17

virus, 1

Literatura

[1] A. V. Aho, M. J. Corasick, ”Efficient string matching: an aid to bibliographic search,” *Communications of the ACM*, št. 18, zv. 6, 1975.

[2] R. S. Boyer, J. S. Moore, ”A fast string searching algorithm,” *Communications of the ACM*, št. 20, zv. 10, 1977.

[3] D. Breslauer, Z. Galil, ”An Optimal O(log log n) Time Parallel String Matching Algorithm,” *Siam Journal on Computing*, št. 19, zv. 6, 1990.

[4] D. Breslauer, Z. Galil, ”Parallel String Matching Algorithms,” v zborniku *Sequences ’91 Workshop*, 1991, str. 121−142.

[5] R. Cole, ”Tight bounds on the complexity of the Boyer-Moore string matching algorithm,” v zborniku *Second Annual ACM Symposium on Discrete Algorithms*, 1991, str. 224−233.

[6] S. Cook, C. Dwork, R. Reischuk, ”Upper and lower time bounds for parallel random access machines without simultaneous writes,” *Siam Journal on Computing*, št. 15, zv. 1, 1986.

[7] Z. Galil, ”Optimal parallel algorithms for string matching,” *Information and Control*, št. 67, zv. 1−3, 1986.

[8] R. M. Karp, M. O. Rabin, ”Efficient randomized pattern-matching algorithms,” *IBM Journal of Research and Development*, št. 31, zv. 2, 1987.

[9] D. E. Knuth, J. H. Morris, V. R. Pratt, ”Fast Pattern Matching in Strings,” *Siam Journal on Computing*, št. 6, zv. 2, 1977.

[10] M. Sustik, J. S. Moore, ”String Searching over Small Alphabets,” *TR-07-62, Department of Computer Sciences, University of Texas at Austin*, 2007.

[11] E. Ukkonen, ”On-Line Construction of Suffix Trees,” *Algorithmica*, št. 14, zv. 3, 1995.

[12] U. Vishkin, ”Optimal parallel pattern matching in strings,” *Information and Control*, št. 67, zv. 1–3, 1985.

[13] U. Vishkin, ”Deterministic Sampling - A New Technique for Fast Pattern Matching,” *Siam Journal on Computing*, št. 20, zv. 1, 1990.