Algoritmi za djelomično podudaranje znakovnih nizova

Ružić, Sanjin

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Sanjin Ružić

ALGORITMI ZA DJELOMIČNO PODUDARANJE ZNAKOVNIH NIZOVA

Diplomski rad

Voditelj rada:
doc. dr. sc. Goranka Nogo

Zagreb, rujan 2015.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Svojoj obitelji i Ivi

Sadržaj

Uvod

Tekst predstavlja jedno od osnovnih sredstava razmjene i pohrane informacija među ljudima, stoga ne čudi da su se razvili mnogobrojni algoritmi koji rješavaju probleme vezane uz tekst, odnosno znakovne nizove.

Klasični predstavnici su algoritmi za sortiranje znakovnih nizova koji se suštinski ne razlikuju od „običnih" algoritama za sortiranje, no ipak mogu iskoristiti poznavanje domene znakovnih nizova (npr., ukoliko je potrebno sortirati tekst na hrvatskom jeziku koristimo se činjenicom da hrvatska abeceda ima 30 slova). Svoju primjenu nalaze u svim situacijama kada je sortirani poredak nužan, na primjer kod ispisa adresara, telefonskog imenika ili kataloga proizvoda.

Ogromna količina podataka koja se svakodnevno stvara, zahtijeva i efikasnu pohranu. Stoga postoje algoritmi čiji je glavni cilj kompresija podataka kako bi se zauzimalo čim manje prostora, te omogućio brži prijenos istih.

Mi ćemo se u ovome radu detaljno baviti trećom skupinom algoritama, tzv. algoritmima za podudaranje znakovnih nizova. Njihove primjene su brojne, a mi ćemo istaknuti samo neke od najznačajnijih. Primjerice, u programima za uređivanje teksta vrlo često se moraju pronaći sva pojavljivanja uzorka u tekstu. Tipično, tekst je dokument koji se uređuje, a uzorak je riječ ili rečenica koju zadaje korisnik. Efikasni algoritmi koji rješavaju taj zadatak mogu uvelike pomoći u responzivnosti programa. Nadalje, od suvremenijih primjena mogli bi spomenuti pretragu zadanog uzorka u DNK nizu te pomoć internetskim tražilicama kod prikaza relevantih rezultata za zadani upit.

Rad je organiziran na sljedeći način.

U prvom poglavlju ćemo uvesti oznake i dati osnovne definicije potrebne za razumijevanje rada. One uključuju područja interpretacije, složenosti te oblikovanja i analize algoritama.

U drugom poglavlju ćemo klasificirati algoritme za podudaranje znakovnih nizova te dati formalnu definiciju problema kojeg oni rješavaju. Za svaki problem prikazat ćemo jedan konkretan primjer i očekivane izlaze algoritma koji ga rješava. Uz to opisat ćemo kratki povijesni pregled razvoja algoritama.

Svako od posljednja tri poglavlja bit će posvećeno pojedinoj klasi algoritama za podudaranje znakovnih nizova. U njima ćemo detaljno ćemo obraditi jedan ili više naj-

značajnijih algoritama za dotičnu klasu. Osim prikaza pseudokoda i rada algoritma, dokazat ćemo korektnost i složenost algoritma.

Poglavlje 0

Oznake i osnovne definicije

0.1 Znak, abeceda, niz, jezik

Znak je apstraktni pojam koji se ne definira formalno kao što se ne definira ni točka u geometriji. Kao primjer znaka obično se uzimaju brojke i slova. Ako se drugačije ne napomene, znak ćemo označavati malim slovima x i y.

Abeceda je konačni skup znakova. Na primjer, brojke 0 i 1 čine binarnu abecedu $B = \{0, 1\}$. U radu će oznaka Σ predstavljati proizvoljnu konačnu abecedu.

Niz je konačni slijed znakova abecede pozicioniranih jedan do drugoga. Niz znakova se često kraće naziva i *string*. Na primjer, nizovi 100, 0011 i 10101 zadani su nad binarnom abecedom $B = \{0, 1\}$. U radu će oznake u i v predstavljati proizvoljne nizove. Također, niz T će biti niz znakova *teksta*, a niz P niz znakova uzorka (eng. *pattern*).

Duljina niza jednaka je broju znakova od kojih se sastoji niz. Primjerice, niz $w = 00110$ ima duljinu $|w| = 5$ jer se sastoji od pet znakova binarne abecede $B = \{0, 1\}$. Niz duljine jedan ćemo poistovjetiti s jedinim znakom tog niza. Koristit ćemo i *indeksiranje od jedinice* pa tako u našem slučaju imamo $w \equiv w[1..5]$, odnosno $w[1] = 0$ i $w[3] = 1$.

Prazni niz označava se znakom ε. To je niz koji ne sadrži niti jedan znak te zato za njega vrijedi $|\varepsilon| = 0$. U slučaju $i > j$ znakovni niz $u[i..j]$ je zapravo prazni niz ε.

Prefiks niza u dobije se odbacivanjem niti jednog, jednog ili više posljednjih znakova niza u. (Na primjer, niz $v =$ `mate` je prefiks niza $u =$ `matematika`.) *Sufiks niza w* dobije se odbacivanjem niti jednog, jednog ili više početnih znakova niza w. (Primjerice, niz $v =$ `jeka` je sufiks niza $u =$ `rijeka`.) *Podniz niza u* dobije se odbacivanjem prefiksa i sufiksa niza u. (Na primjer, niza $v =$ `osa` je podniz niza $u =$ `posao`.) *Nadovezivanje niza u i niza v* dobije se dodavanjem znakova niza v iza znakova niza u, a rezultat označavamo kao $w = uv$. (Na primjer, nadovezivanjem nizova $u =$ `na` i $v =$ `rasti` nastaje niz $w =$ `narasti`.) Primijetimo da je prazni niz ε neutralni element za nadovezivanje nizova jer vrijedi $\varepsilon u =$

$u\varepsilon = u$. Nadovezivanje se jednostavnije označava potencijama na sljedeći način:

$$u^0 = \varepsilon$$
$$u^i = w^{i-1}w.$$

Formalni jezik je skup nizova nad abecedom. Na temelju dane formalne definicije kao primjeri jezika mogu se navesti: prazni skup $L_1 = \{\}$; skup kojem je jedini element prazni niz $L_2 = \{\varepsilon\}$; skup nizova za koje vrijedi da je broj nula i broj jedinica paran broj $L_3 = \{00, 11, 0011, 0101, 1001, 1100, \ldots\}$. Primijetimo da iako je jezik L_3 zadan nad konačnom abecedom $\{0, 1\}$ on sam nije konačan. *Unija jezika L i N*, u oznaci $L \cup N$, se definira s

$$L \cup N = \{u : u \in L \text{ ili } u \in N\}.$$

Nadovezivanje jezika L i N, u oznaci LN, se definira s

$$LN = \{uv : u \in L \text{ i } v \in N\}.$$

Kleenov operator L^* se definira s

$$L^* = \bigcup_{i=0}^{\infty} L^i,$$

gdje je s L^i označeno nadovezivanje jezika $L^0 = \{\varepsilon\}$, $L^i = L^{i-1}L$.

0.2 Konačni automati

Automat jest model diskretnog matematičkog sustava koji čitanjem znak po znak odlučuje je li pročitani niz element zadanog jezika. Automat se sastoji od *stanja*. Prije čitanja prvog znaka automat se nalazi u *početnom* stanju. Čitanjem znakova niza automat mijenja stanja. Stanja se dijele na *prihvatljiva* i *neprihvatljiva*. Ako se nakon posljednjeg pročitanog znaka automat nalazi u jednom od prihvatljivih stanja, onda se niz prihvaća. Skup svih nizova koje prihvaća automat M je jezik za koji se kaže da ga prihvaća automat M. Jezik koji prihvaća automat M označava se s $L(M)$.

Deterministički konačni automat sastoji se od konačnog skupa stanja i funkcije prijelaza. Funkcija prijelaza jednoznačno je određena znakom na ulazu i stanjem u kojem se nalazi automat. Za pojedini znak i stanje postoji samo jedan prijelaz. Deterministički konačni automat D formalno se zadaje kao uređena petorka:

$$D = (Q, \Sigma, \delta, q_0, F) \tag{0.1}$$

gdje je

Q	konačni skup stanja
Σ	konačni skup ulaznih znakova
δ	funkcija prijelaza $Q \times \Sigma \to Q$
$q_0 \in Q$	početno stanje
$F \subseteq Q$	skup prihvatljivih stanja.

Definicija deterministočkog konačnog automata proširuje se novom funkcijom prijelaza. Za razliku od funkcije prijelaza deterministočkog konačnog automata koja za jedno stanje i jedan znak određuje prijelaz u jedinstveno stanje $\delta(q, a) = p$, nova funkcija prijelaza određuje prijelaz u skup stanja $\delta(q, a) = \{p_1, p_2, \ldots\}$. Skup stanja može biti prazan. Konačni automat koji se zasniva na novoj funkciji prijelaza naziva se *nedeterministički konačni automat*. Nedeterminizam pritom ne označava pojavu slučajnih događaja već činjenicu da za neki niz u nedeterministički konačni automat N može imati više od jednog slijeda prijelaza. Tijekom postupka utvrđivanja prihvaća li N niz u, provjeravaju se svi slijedovi prijelaza. Postoji li barem jedan slijed prijelaza iz početnog stanja u jedno od prihvatljivih stanja, tada N prihvaća niz u. Nedeterministički konačni automat N formalno se zadaje kao uređena petorka:

$$D = (Q, \Sigma, \delta, q_o, F)$$

pri čemu je jedina razlika u odnosu na definiciju (0.1) u funkciji prijelaza $\delta\colon\ Q\times\Sigma \to \mathcal{P}(Q)$.

0.3 Složenost algoritama

Određeni problem se uvijek može riješiti sa mnogo algoritama. Postavlja se pitanje postoji li među svim algoritmima koji predstavljaju rješenje problema „najbolji" i po kojim kriterijima ga odrediti. U tu svrhu promatramo količinu *resursa* koji su potrebni algoritmu u izračunavanju. Uglavnom su nam najzanimljivija dva resursa – prostor i vrijeme.

Vremenska složenost izražava potrebno vrijeme za izvođenje algoritma u nekim osnovnim jedinicama. Najčešće jedinice su ili standardne vremenske (sekunde, minute i sl.) ili strojne (osnovni ciklus računala ili prosječno trajanje osnovnih instrukcija). Odabir mjere vremenske složenosti treba biti takav da omogućava objektivno uspoređivanje *algoritama*, a ne strojeva.

Prostornom složenošću mjerimo potrebnu memoriju za izvođenje algoritma, također u nekim osnovnim jedinicama (bit, jedan integer ili sl.). Prostorna složenost, osim ulaza i izlaza, uključuje i sve međurezultate koji se javljaju prilikom izvođenja algoritma. Pritom je dozvoljeno više puta koristiti istu memoriju za razne podatke u razna vremena.

U oba slučaja razlikujemo *najgoru* i *prosječnu složenost*. Kod najgore složenosti za danu veličinu problema uzimamo najveće moguće vrijeme (ili neki drugi resurs) za *sve* zadaće te veličine. Ona predstavlja najgori mogući slučaj, odnosno garanciju da izračun

ne troši više resursa od toga. S druge strane, prosječna složenost uzima prosječno trajanje po svim zadaćama iste veličine. Ta mjera je mnogo realističnija, no zbog težeg teoretskog izračuna od najgore složenosti, ona se rjeđe koristi.

Pretpostavljamo da je složenost f nekog problema funkcija oblika

$$f\colon \mathcal{D} \to \mathbb{R}$$

gdje je $\mathcal{D}$ odozgo neograničen skup (na primjer, u $\mathbb{R}_0$ ili $\mathbb{N}_0$). Očekujemo da veća zadaća predstavlja teži problem, tj. da je f monotono rastuća funkcija. No kako to ne vrijedi za sve probleme, ne uvodimo dodatne pretpostavke na f. Uspoređivanje složenosti algoritama uvodimo sljedećom definicijom.

Definicija 0.3.1. Neka su $f, g\colon \mathcal{D} \to \mathbb{R}$ dvije funkcije na odozgo neograničenom podskupu $\mathcal{D} \subseteq \mathbb{R}$.

(a) *f nije većeg reda veličine od g*, ili *f ne raste brže od g*, u oznaci

$$f(x) = \mathcal{O}(g(x)) \quad (x \to \infty)$$

ako postoje $c \in \mathbb{R}$ i $x_0 \in \mathcal{D}$ takvi da za svaki $x > x_0$ vrijedi

$$|f(x)| < c|g(x)|.$$

(b) *f je istog reda veličine kao i g*, ili *f raste istom brzinom kao i g*, u oznaci

$$f(x) = \Theta(g(x)) \quad (x \to \infty)$$

ako postoje $c_1, c_2 > 0$ ($c_1, c_2 \in \mathbb{R}$) i $x_0 \in \mathcal{D}$ takvi da za svaki $x > x_0$ vrijedi

$$c_1|g(x) < |f(x)| < c_2|g(x)|.$$

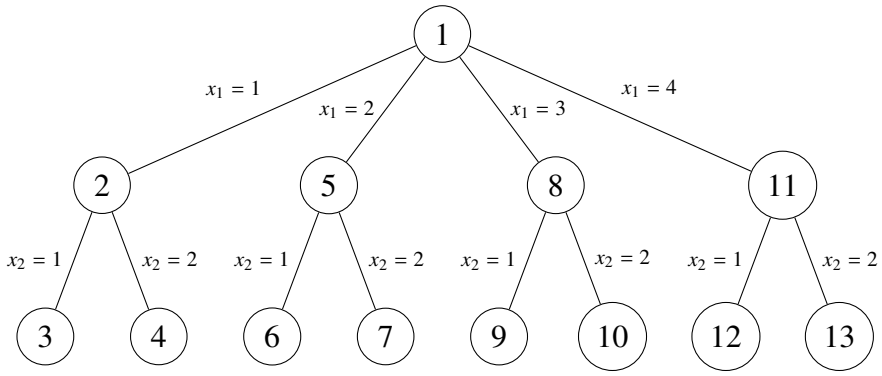
0.4 Oblikovanje algoritama

Dinamičko programiranje

Dinamičko programiranje je metoda koja zahtijeva da se svi potproblemi redom riješe, te da se rješenja spreme u odgovarajuću tablicu. Kad god nam treba neko rješenje, tada ga ne računamo ponovo već ga samo pročitamo iz tablice. Za razliku od algoritama tipa „podijeli pa vladaj" koji idu „s vrha prema dolje" (od većeg problema prema manjima), algoritmi dinamičkog programiranja idu s „dna prema gore" (od manjih prema većem). Znači, prvo se u tablicu unose rješenja za probleme najmanje veličine, zatim rješenja za malo veće probleme, itd., sve dok se ne dosegne veličina zadanog problema. Važan je redoslijed ispunjavanja tablice. Postupak dinamičkog programiranja koji put zahtijeva da riješimo i neke potprobleme koji nam na kraju neće biti potrebni za rješenje zadanog problema. Ipak, to se još uvijek više isplati nego rješavanje istih potproblema mnogo puta.

Metoda *backtrackinga*

Backtracking je vrlo općenita metoda koja se primjenjuje za teške kombinatorne probleme. Rješenje problema se traži sistematskim ispitivanjem svih mogućnosti za konstrukciju tog rješenja. Metoda zahtijeva da se traženo rješenje izrazi kao n-torka oblika $(x_1, x_2, \ldots, x_n)$, gdje je x_i element nekog konačnog skupa S_i. Kartezijev produkt $S_1 \times S_2 \times \cdots \times S_n$ zove se prostor rješenja. Da bi neka konkretna n-torka iz prostora rješenja zaista predstavljala rješenje, ona mora zadovoljiti još i neka dodatna ograničenja (ovisna o samom problemu). Prostor rješenja treba zamišljati kao uređeno stablo rješenja. Korijen tog stabla predstavlja sve moguće n-torke. Dijete korijena predstavlja sve n-torke gdje prva komponenta x_1 ima neku određenu vrijednost. Unuk korijena predstavlja sve n-torke gdje su prve dvije komponente x_1 i x_2 fiksirane na određeni način, itd. List stabla predstavlja jednu konkretnu n-torku. Backtracking je u osnovi rekurzivni algoritam koji se sastoji od simultanog ge-



Slika 0.1: Stablo rješenja.

neriranja i ispitivanja čvorova u stablu rješenja. Čvorovi se stavljaju na stog. Jedan korak algoritma sastoji se od toga da se uzme čvor s vrha stoga, te da se provjeri da li taj čvor predstavlja rješenje problema. Ukoliko čvor predstavlja rješenje, tada se poduzme odgovarajuća akcija. Ukoliko čvor nije rješenje, tada se pokušaju generirati njegova djeca te se ona stavljaju na stog. Algoritam počinje tako da na stogu bude samo korijen stabla; završetak je onda kad nađemo rješenje ili kad se isprazni stog. Redoslijed obrađivanja čvorova (skidanja sa stoga) vidi se na slici 0.1. Veličina prostora rješenja raste eksponencijalno s veličinom problema n. Zbog toga dobar backtracking algoritam nikad ne generira cijelo stablo rješenja, već „reže" one grane (podstabla) za koje uspije utvrditi da ne vode do rješenja. Naime, u samom postupku generiranja čvorova provjeravaju se ograničenja koja n-torka $(x_1, \ldots, x_n)$ mora zadovoljiti da bi zaista bila rješenje. Čvor na i-tom nivou predstavlja n-torke gdje je prvih i komponenti $x_1, \ldots, x_i$ fiksirano na određeni način. Ukoliko se već na osnovu vrijednosti tih i komponenti može utvrditi da ograničenja nisu zadovoljena, tada dotični čvor ne treba generirati jer ni on ni njegova djeca neće dati rješenje. Npr. ako u

kontekstu prethodne slike vrijedi ograničenje da komponenta x_1 mora biti paran broj, tada se neće generirati prvo i treće dijete korijena stabla rješenja pa ukupan broj generiranih čvorova pada na 7.

Poglavlje 1

Opis problema

Osim klasičnog egzaktnog podudaranja, postoje još dvije varijante problema: aproksimativno podudaranje znakovnih nizova te podudaranje znakovnih nizova koji sadrže zamjenske znakove. U tekstu kada ne specificiramo podudaranje, onda mislimo na egzaktno podudaranje znakovnih nizova.

1.1 Egzaktno podudaranje znakovnih nizova

Definicija 1.1.1. Neka je *tekst* niz znakova T duljine n te neka je *uzorak* niz znakova P duljine $m \leqslant n$. Elementi nizova T i P su znakovi iz konačne abecede Σ.

Kažemo da se uzorak P *pojavljuje u tekstu* T *s pomakom* s ako vrijedi $0 \leqslant s \leqslant n - m$ i

$$T[s + 1..s + m] = P[1..m],$$

odnosno, $T[s + j] = P[j]$ za svaki $1 \leqslant j \leqslant m$. Ako se P javlja u T s pomakom s, tada kažemo da je *s valjan pomak*; u protivnom je *s nevaljan pomak*.

Sada se *problem (egzaktnog) podudaranja znakovnih nizova* svodi na pronalaženje svih valjanih pomaka s kojima se dani uzorak P pojavljuje u danom tekstu T.

Primjer 1.1.2. Neka abeceda dana sa $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$ predstavlja nukleotide adenin, citozin, gvanin i timin. Neka je tekst (DNK molekula) dan s $T = \texttt{TTAGACGTAG}$. Za dani (DNK) uzorak $P = \texttt{TAG}$, algoritam za (egzaktno) podudaranje znakovnih nizova mora vratiti sljedeća pojavljivanja: TTAGACGTAG i TTAGACGTAG, tj. mora vratiti valjane pomake 2 i 8, respektivno.

1.2 Aproksimativno podudaranje znakovnih nizova

Definicija 1.2.1. Neka su A i B znakovni nizovi na konačnoj abecedi Σ. *Edit-udaljenost* (eng. *edit* – urediti) *od stringa A do stringa B*, u oznaci $\delta(A, B)$, je jednaka sumi težina niza dozvoljenih operacija uređivanja koje transformiraju A u B.

Jedan od najjednostavnijih skupova operacija uređivanja je definirao Levenshtein 1966. godine:

(1) *ubacivanje* jednog znaka: ako je $A = uv$, tada ubacivanjem znaka x dobivamo $A = uxv$

(2) *brisanje* jednog znaka: ako je $A = uxv$, tada brisanjem znaka x dobivamo $A = uv$

(3) *zamjena* jednog znaka drugim: ako je $A = uxv$, tada zamjenom znaka x znakom y dobivamo $A = uyv$.

Pretpostavit ćemo da svaka od Levenshteinovih operacija uređivanja ima jediničnu težinu (osim zamjene znaka istim tim znakom koja ima težinu nula). Tada se *edit*-udaljenost stringova svodi na minimalan broj operacija potrebnih da se string A transformira u string B.

Primjer 1.2.2. Neka je $A =$ `slovo` te $B =$ `sivko`. Tada je $\delta(A, B) = 3$ jer string A možemo transformirati u string B u najmanje 3 koraka:

1. `slovo` → `slov`k`o` (ubacivanje znaka `k`)

2. `s`l`ovko` → `sovko` (brisanje znaka `l`)

3. `s`o`vko` → `s`i`vko` (zamjena znaka `o` znakom `i`).

U nastavku dajemo jednu moguću definiciju problema aproksimativnog podudaranja znakovnih nizova.

Definicija 1.2.3. Neka su dani stringovi T i P na konačnoj abecedi Σ. *Problem aproksimativnog podudaranja znakovnih nizova* je za dani uzorak P pronaći sve znakovne podnizove S teksta T takve da za sve znakovne podnizove S' teksta T vrijedi

$$\delta(P, S) \leqslant \delta(P, S').$$

Drugim riječima, potrebno je pronaći sve podnizove teksta T koji se razlikuju što je manje moguće od stringa P s obzirom na *edit*-udaljenost.

1.3 Podudaranje znakovnih nizova sa zamjenskim znakovima

Definicija 1.3.1. Neka je Σ konačna abeceda. *Zamjenski znakovi* **?** i $\star$ su rezervirani znakovi takvi da je $\Sigma \cap \{\mathbf{?}, \star\} = \emptyset$. Znak **?** označava točno jedan znak, dok znak $\star$ označava nula ili više proizvoljnih znakova abecede Σ.

Napomena 1.3.2. Za potrebe egzaktnog podudaranja stringova, smatrat ćemo da se znak **?** podudara sa svakim znakom abecede.

Definicija 1.3.3. Neka je tekst niz znakova T duljine n s elementima iz konačne abecede Σ, te neka je uzorak niz znakova P duljine $m \leqslant n$ s elementima iz $\Sigma \cup \{\mathbf{?}, \star\}$. *Problem podudaranja znakovnih nizova sa zamjenskim znakovima* je odrediti podudara li se dani tekst T s danim uzorkom P.

Primijetimo da je ovako definiran problem podudaranja stringova sa zamjenskim znakovima zapravo problem odluke. Algoritam koji rješava navedeni problem mora za ulazne podatke T i P mora „vratiti" istina ili laž ovisno o tome podudaraju li se T i P ili ne.

Primjer 1.3.4. Neka je abeceda $\Sigma = \{0, 1\}$ te neka je A algoritam koji rješava problem podudaranja stringova sa zamjenskim znakovima. Za uzorak $P_1 = \mathbf{?}1\mathbf{?}$ algoritam A mora vratiti istinu ako i samo ako je tekst T_1 binarni niz duljine 3 kojemu je srednji znak 1. Uzorak $P_2 = \star 0$ označava parne brojeve zapisane u binarnom prikazu, tj. algoritam A mora vratiti istinu ako i samo ako je tekst T_2 binarni niz koji završava znakom 0.

1.4 Kratki pregled razvoja algoritama

Algoritmi kojima ćemo se baviti imaju zanimljivu povijest koju ćemo u nastavku iznijeti.

Postoji jednostavni, naivni algoritam koji „grubom silom" rješava problem egzaktnog podudaranja znakovnih nizova. Iako je njegova vremenska složenost $O(mn)$, stringovi koji se javljaju u praksi najčešće vode k optimalnoj vremenskoj složenosti, odnosno $O(m + n)$.

S. Cook je 1970. godine teoretski dokazao egzistenciju algoritma koji rješava problem podudaranja znakovnih nizova u vremenskoj složenosti jednakoj $O(m + n)$ u najgorem slučaju. Iako njegov dokaz nije bio konstruktivne naravi, D. E. Knuth i V. R. Pratt su ga rafinirali u relativno jednostavan i praktičan algoritam. Međutim, ispostavilo se da je i J. H. Morris otkrio praktički identičan algoritam rješavajući problem koji mu se javljao dok je razvijao uređivač teksta. Činjenica da se iz dva različita pristupa razvio isti algoritam dala mu je na značaju kao fundamentalnom rješenju problema.

Tek 1977. godine Knuth, Morris i Pratt su zajednički objavili svoj rad, a u međuvremenu su R. S. Boyer i J. S. Moore (te, nezavisno, R. W. Gosper) otkrili algoritam koji je puno

brži u praktičnim primjenama te ga stoga najčešće koriste programi za uređivanje tekstova kako bi smanjili vrijeme odaziva prilikom pretrage dokumenta.

1980. godine M. O. Rabin i R. M. Karp su razvili algoritam zasnovan na *hashiranju* koji je tek nešto kompliciraniji od naivnog algoritma. Iako je njegova vremenska složenost jednaka $O(mn)$, s velikom vjerojatnošću možemo očekivati vremensku složenost proporcionalnu sa $m + n$.

Dali smo pregled nekoliko najpoznatijih predstavnika, iako ih ima još na desetke s vrlo sličnim, ili pak potpuno različitim idejama rješavanja. Stoga se i smatra da postoje algoritmi koje tek treba otkriti, makar problem podudaranja znakovnih nizova predstavlja klasičan problem u računarstvu.

Poglavlje 2

Algoritmi za egzaktno podudaranje

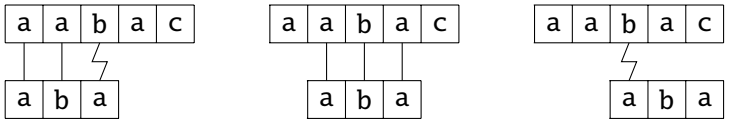
2.1 Naivni algoritam

Očiti način kako bi mogli riješiti problem egzaktnog podudaranja stringova je da za svaku moguću poziciju u tekstu gdje bi se mogao pojaviti uzorak, provjerimo javlja li se on uistinu.

Formalno rečeno, naivni algoritam pronalazi sve valjane pomake koristeći petlju koja provjerava uvjet $P[1..m] = T[s+1..s+m]$ za svaku od $n-m-1$ mogućih vrijednosti za s. Pseudokod je dan u nastavku.

Algoritam 1 NaivniAlgoritam *(T, P)*

```
1: n ← T.length
2: m ← P.length
3: for s = 0 to n − m do
4:     if P[1..m] == T[s + 1..s + m] then
5:         print „Uzorak se javlja s pomakom” s
```



Slika 2.1: Grafički prikaz rada naivnog algoritma za T = aabac i P = aba. Slike redom prikazuju provjere pomaka 0, 1 i 2. Ravnom crtom je označeno podudaranje, a znakom „munje” nepodudaranje odgovarajućih znakova teksta i uzorka. Ukoliko nema nepodudaranja, tada je promatrani pomak valjan. U našem slučaju to je jedino $s = 1$.

Već smo spomenuli da je vremenska složenost naivnog algoritma jednaka $O(mn)$, tj. sada bi preciznije mogli reći $O((n-m-1)m)$. U najgorem slučaju slučaju ona se i postiže.

Propozicija 2.1.1. *Vremenska složenost naivnog algoritma za egzaktno podudaranje znakovnih nizova u najgorem slučaju je jednaka* $\Theta((n-m-1)m)$.

Dokaz. Tvrdnju ćemo dokazati primjerom. Neka je tekst $T = \mathtt{a}^n$, tj. tekst se sastoji od n uzastopnih znakova a. Neka je $P = \mathtt{a}^m$ gdje je $m \leqslant n$. Očito je svaka od $n-m+1$ mogućih vrijednosti za pomak s valjana. Stoga će se za svaki $0 \leqslant s \leqslant n-m$ implicitna petlja iz 4. linije u potpunosti izvršiti, odnosno algoritam će napraviti m usporedbi znakova da bi provjerio da je pomak s uistinu valjan. □

2.2 Rabin-Karp algoritam

Algoritam koji su razvili M. O. Rabin i R. A. Karp zasniva se na *hashiranju*. Najprije izračunamo hash vrijednost uzorka, a zatim koristeći istu hash funkciju izračunamo hash vrijednost svakog podniza teksta duljine m. Ukoliko pronađemo podudaranje hash vrijednosti uzorka s hash vrijednosti nekog podniza, tek tada provjeravamo radi li se o istim znakovnim nizovima.

Iako je ideja algoritma relativno jednostavna, naivna implementacija bi vodila na lošiju složenost od naivnog algoritma iz prethodnog odjeljka (jer je računanje hash vrijednosti podniza znakova skuplja operacija od jednostavnog uspoređivanja znakova s uzorkom). Stoga hash funkcija mora zadovoljavati neka dodatna svojstva:

(1) efikasno izračunavanje

(2) raspršenost

(3) $\text{hash}(T[j+1..j+m])$ se mora moći jednostavno izračunati koristeći $hash(T[j..j+m-1])$ i $T[j+m]$.

Pokazuje se da tražena svojstva zadovoljava sljedeća hash funkcija:

$$\text{hash}(T[j+1..j+m]) = T[j+1]R^{m-1} + T[j+2]R^{m-2} + \cdots + T[j+m]R^0 \pmod{Q}, \quad (2.1)$$

gdje su $R, Q \in \mathbb{N}$. R je najčešće veličina abecede Σ, dok je Q veliki prosti broj.

Formulu (2.1) možemo shvatiti kao polinom kojemu su koeficijenti upravo znakovi (tj. njihove numeričke vrijednosti), odnosno ona na svaki znakovni podniz duljine m gleda kao na broj u bazi R. Svojstvo (1) navedene hash funkcije se svodi na primjenu Hornerova algoritma. Svojstvo (2) slijedi iz odabira parametara R i Q, dok svojstvo (3) vrijedi zbog

$$\text{hash}(T[j+1..j+m]) = \left(\text{hash}(T[j..j+m-1] - T[j]R^{m-1}\right)R + T[j+m] \pmod{Q}. \quad (2.2)$$

Primjer 2.2.1. Najjednostavniji primjer je kada promatramo znakovne nizove na abecedi $\Sigma = \{\texttt{0},\texttt{1},\ldots,\texttt{9}\}$. Tada imamo prirodno preslikavanje između znaka te njegove numeričke vrijednosti. (Općenito, znakove abecede Σ možemo poredati te dobiti $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_{k-1}\}$, te nakon toga znak σ_i promatrati kao znamenku i u bazi k.) Neka je $R = 10$, a $Q = 97$. Sada je hash vrijednost podniza `279` jednaka

$$\begin{aligned}\text{hash}(\texttt{279}) &= 2 \cdot 10^2 + 7 \cdot 10^1 + 9 \cdot 10^0 \pmod{97} \\ &= 279 \pmod{97} \\ &= 85 \pmod{97}.\end{aligned}$$

Da ilustriramo svojstvo (3), izračunajmo hash vrijednost podniza `795`:

$$\begin{aligned}\text{hash}(\texttt{795}) &= \left(85 - 2 \cdot 10^2\right) \cdot 10 + 5 \pmod{97} \\ &= -1145 \pmod{97} \\ &= 19 \quad (= 795 \pmod{97}).\end{aligned}$$

U nastavku dajemo pseudokod Rabin-Karp algoritma te ilustraciju njegova izvođenja na jednom primjeru.

Algoritam 2 RabinKarp (T, P, R, Q)

```
1:  n ← T.length
2:  m ← P.length
3:  h ← R^(m−1)
4:  p = 0
5:  t_0 = 0
6:  for i = 1 to m do
7:      p = R · p + P[i] (mod Q)
8:      t_0 = R · t_0 + T[i] (mod Q)
9:  for s = 0 to n − m do
10:     if p == t_s then
11:         if P[1..m] == T[s + 1..s + m] then
12:             print „Uzorak se javlja s pomakom” s
13:     if s < n − m then
14:         t_{s+1} = (t_s − T[s + 1]h) R + T[s + m + 1] (mod Q)
```

Algoritam svaki znak promatra kao znamenku u bazi R. Na početku se izračunaju hash vrijednosti uzorka te početnog podniza teksta duljine m (linije 6-8) što je vremenske složenosti $\Theta(m)$. Osim toga, samo jednom se izračuna vrijednost R^{m-1} (linija 3) što se

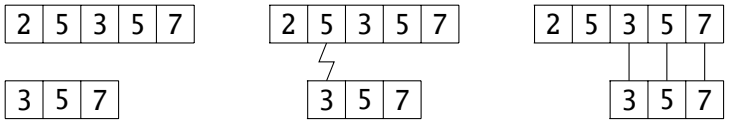
brzim potenciranjem može izvesti u $\Theta(\log m)$. Dakle, ukupno pretprocesiranje je složenosti $\Theta(m)$.

Zatim **for**-petlja (linije 9-14) prolazi svim mogućim pomacima s održavajući sljedeću invarijantu:

$$\textit{svaki put kada se linija 10 izvrši, vrijedi } t_s = hash(T[s+1..s+m]).$$

Pomak s može biti valjan samo u slučaju da postoji podudaranje hash vrijednosti uzorka te promatranog podniza $T[s+1..s+m]$ (linija 10). Stoga u tom slučaju dodatno provjeravamo radi li se uistinu o valjanom pomaku (linija 11). Konačno, linije 13-14 osvježavaju hash vrijednosti u skladu s formulom (2.2).

Cijela ideja algoritma se zasniva na hashiranju koje osigurava da provjeravamo samo nekoliko kandidata za valjani pomak s. Ipak, u najgorem slučaju vremenska složenost algoritma je $\Theta((n-m+1)m)$, a postiže se na istom primjeru kao i za naivni algoritam (vidi propoziciju 2.1.1).



Slika 2.2: Grafički prikaz rada Rabin-Karp algoritma za $T = 25357$ i $P = 357$, te $R = 10$, $Q = 89$. Za pomak $s = 0$ imamo da je $253 \not\equiv 357 \pmod{89}$, pa ni ne provjeravamo jednakost odgovarajućih stringova. Za pomak $s = 1$ vrijedi $535 \equiv 357 \pmod{89}$, pa krećemo u provjeru radi li se uistinu o podudaranju. Ipak, već na prvom znaku provjera rezultira nepodudaranjem. Konačno, za $s = 3$ imamo $357 \equiv 357 \pmod{89}$ te daljnja provjera otkriva da se radi o istim znakovnim podnizovima.

2.3 Knuth-Morris-Pratt algoritam

Glavna ideja iza algoritma kojeg su otkrili Knuth, Morris i Pratt je činjenica da svaki put kada naiđemo na nepodudaranje odgovarajućeg znaka uzorka sa znakom teksta, već znamo neke znakove u tekstu (zbog toga što su znakovi prije nepodudaranja bili jednaki). Tu informaciju možemo iskoristiti tako da izbjegnemo provjeravanje pomaka za koje unaprijed znamo da nisu valjani.

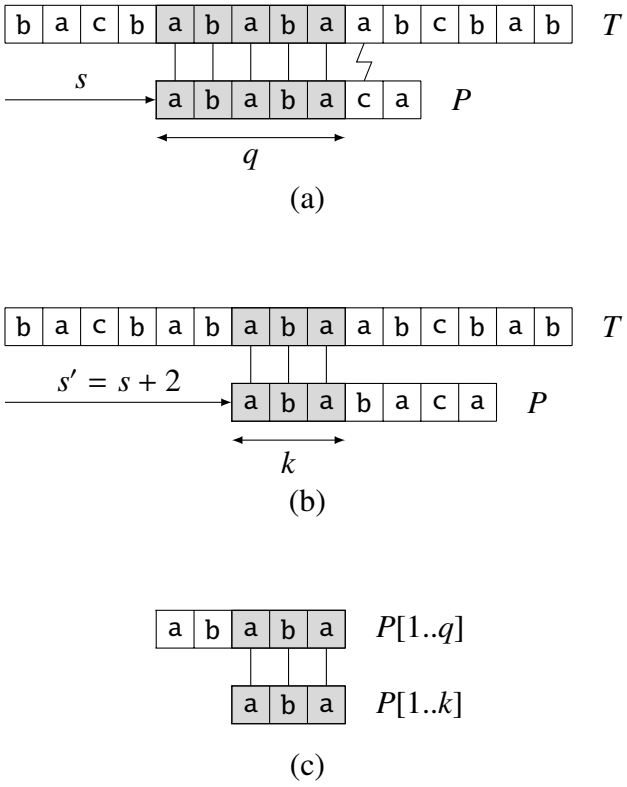
Sada ćemo ilustrirati rečeno na jednostavnom primjeru. Slika 2.3 (a) pokazuje određeni pomak s zajedno s tekstom T te uzorkom $P = \texttt{ababaca}$. Vidimo da se prvih $q = 5$ znakova uzorka i teksta podudaraju, no šesti znak uzorka je različit od odgovarajućeg znaka teksta. Poznavajući tih q znakova teksta, možemo odmah zaključiti da je pomak $s + 1$ nužno nevaljan jer bi njime prvi znak uzorka (a) bio poravnat sa znakom teksta za kojeg znamo da je različit od a. S druge strane, pomak $s' = s+2$ poravnava prva tri znaka s odgovarajuća tri

znaka teksta (slika 2.3 (b)) te, uzimajući u obzir sve što dosad znamo o tekstu, predstavlja potencijalno valjan pomak. Općenito, željeli bi znati odgovor na sljedeće pitanje:

za dani podniz znakova uzorka $P[1..q]$ *koji se podudara s* $T[s+1..s+q]$*, koji je najmanji pomak* $s' > s$ *takav da za neki* $k < q$ *vrijedi*

$$P[1..k] = T[s'+1..s'+k], \tag{2.3}$$

gdje je $s' + k = s + q$*?*



Slika 2.3: Prefiks funkcija π. **(a)** Uzorka P je poravnat s tekstom T tako da se prvih $q = 5$ znakova podudaraju (označeno osjenčano). **(b)** Korištenjem isključivo znanja o podudaranju 5 znakova, možemo zaključiti da je pomak $s + 1$ nevaljan, ali i da je pomak $s + 2$ konzistentan sa svime što dosad znamo o tekstu T te stoga predstavlja potencijalno valjan pomak. **(c)** Uspoređivanjem uzorka sa samim sobom možemo unaprijed izračunati korisne informacije za ovakva zaključivanja. Na slici vidimo da je $P[1..3]$ najdulji prefiks od P koji je ujedno i sufiks od $P[1..5]$. Tu unaprijed izračunatu informaciju bilježimo u nizu π kao $\pi[5] = 3$.

U najboljem slučaju će biti $k = 0$, te ćemo zbog $s' = s + q$ moći odmah zanemariti pomake $s + 1, s + 2, \ldots, s + q - 1$. U svakom slučaju, kod novog pomaka s' ne trebamo

uspoređivati prvih k znakova uzorka s odgovarajućim znakovima teksta jer nam jednadžba (2.3) garantira da se oni podudaraju.

Potrebne informacije dobivamo uspoređivanjem uzorka sa samim sobom kako pokazuje slika 2.3 (c). Jednadžbu (2.3) možemo interpretirati na način da se pitamo koji je najveći k takav da je $P[1..k]$ sufiks od $P[1..q]$. Tada će novi pomak $s' = s + (q - k)$ predstavljati potencijalni valjani pomak. Time, za dani uzorak $P[1..m]$, dobivamo *prefiks funkciju* $\pi\colon \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ definiranu s

$$\pi(q) = \max\{k : k < q,\ P[1..k] \text{ je prefiks od } P[1..q]\}.$$

U idućem pseudokodu pretpostavljamo da je postoji pomoćna funkcija IzračunajPrefiksFunkciju koja računa π u vremenskoj složenosti $\Theta(m)$ (vidi [2], str. 1006.).

Algoritam 3 KnuthMorrisPratt (T, P)

```
1:  n ← T.length
2:  m ← P.length
3:  π ← IzračunajPrefiksFunkciju(P)
4:  q = 0
5:  for i = 1 to n do
6:      while q > 0 and P[q + 1] ≠ T[i] do
7:          q ← π(q)
8:      if P[q + 1] == T[i] then
9:          q ← q + 1
10:     if q == m then
11:         print „Uzorak se javlja s pomakom” i − m
12:         q ← π(q)
```

Sada ćemo pokazati da je vremenska složenost danog algoritma jednaka $O(m + n)$. Jedino što zapravo treba pokazati je da se **while**-petlja (linije 6-7) izvršava $O(n)$ puta sveukupno. Koristimo se činjenicom da je $\pi(q) < q$, za svaki $q \in \{1, \ldots, m\}$. Primijetimo da je početna vrijednost za q jednaka 0, te jedini način da se q poveća je operacija inkrementa u liniji 9 koja se izvrši najviše jednom po iteraciji **for**-petlje (linije 5-12). Zbog toga što q nikada ne postaje negativan, slijedi da je ukupan broj smanjivanja vrijednosti q u **while**-petlji odozgo ograničen s ukupnim brojem povećavanja, odnosno s n. Dakle, **while**-petlja se najviše izvrši n puta, odnosno ukoliko vremenskoj složenosti pretprocesiranja (linije 1-4), koja iznosi $\Theta(m)$, nadodamo vremensku složenost **for**-petlje, dobivamo da je vremenska složenost cjelokupnog algoritma jednaka $O(m + n)$.

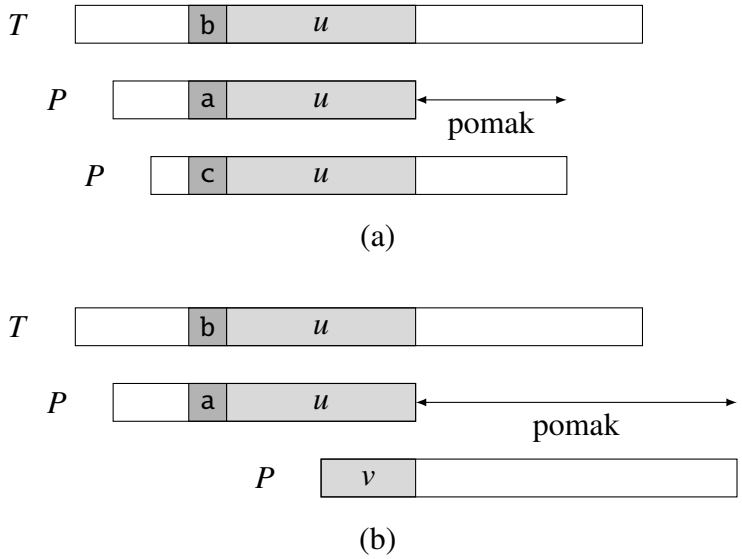
Formalni dokaz korektnosti algoritma je ponešto kompliciraniji, a može se pronaći u [2].

2.4 Boyer-Moore algoritam

Boyer-Moore algoritam se smatra najefikasnijim algoritmom za podudaranje stringova u stvarnim primjenama te se stoga upravo on, ili njegova pojednostavljena verzija, najčešće koristi u programima za uređivanje tekstova u naredbama pretrage ili zamjene teksta.

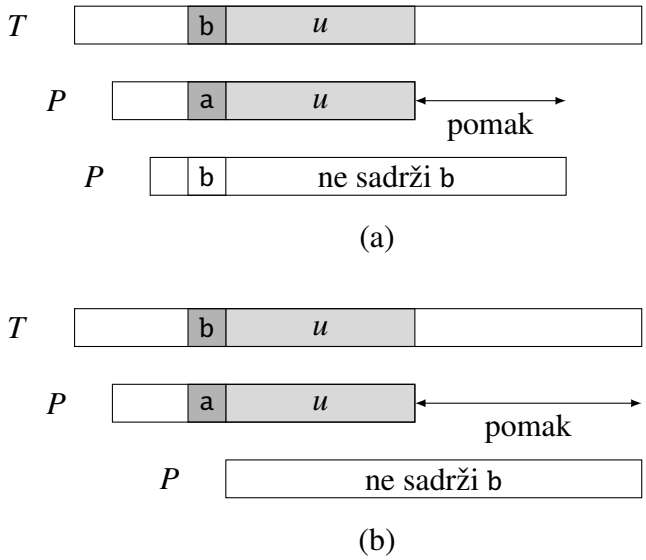
Algoritam „skenira" znakove uzorka zdesna nalijevo, te u slučaju nailaska na nepodudaranje odgovarajućih znakova koristi se dvjema unaprijed izračunatima funkcijama za ažuriranje pomaka koje se nazivaju *pomak dobrog sufiksa* (eng. *good-suffix shift*) te *pomak lošeg znaka* (eng. *bad-character shift*). Opišimo malo detaljnije navedene funkcije.

Pretpostavimo da se nepodudaranje dogodilo između znaka uzorka $P[i] = a$ te znaka teksta $P[i + j] = b$. Tada vrijedi $P[i + 1..i + m] = T[i + j + 1..i + j + m]$ i $P[i] \neq T[i + j]$. Pomak dobrog sufiksa nastoji poravnati segment $P[i + 1..i + m]$ s njegovim „najdesnijim" pojavljivanjem u uzorku kojemu prethodi znak različit od $P[i]$ (vidi 2.4 (a)). Ako takav segment ne postoji, tada se navedeni pomak sastoji u poravnanju najduljeg sufiksa v od $P[i + 1..i + m]$ koji je ujedno i prefiks uzorka P (kao na slici 2.4 (b)). Primijetimo da je zapravo ideja pomaka dobrog sufiksa vrlo slična prefiks funkciji π kod Knuth-Morris-Pratt algoritma.



Slika 2.4: Pomak dobrog sufiksa. **(a)** Podniz u se ponovno javlja u uzorku s time da mu prethodi znak `c` različit od `a`. **(b)** Samo se sufiks podniza u pojavljuje u uzorku.

Pomak lošeg znaka poravnava znak teksta $T[i + j]$ s njegovim „najdesnijim" pojavljivanjem u dijelu uzorka $P[1..m - 1]$ (vidi sliku 2.5 (a)). Ako se znak $T[i + j]$ uopće ne pojavljuje u traženom segmentu, tada znamo da niti jedno pojavljivanje uzorka u tekstu



Slika 2.5: Pomak lošeg znaka. **(a)** Znak b se pojavljuje u uzorku P. **(b)** Znak b se ne pojaljuje u uzorku P.

neće obuhvaćati znak $T[i + j]$, pa stoga poravnamo lijevi kraj uzorka sa znakom nakon $T[i + j]$, odnosno sa znakom $T[i + j + 1]$ (kao na slici 2.5 (b)).

Kako pomak lošeg znaka može rezultirati negativnim pomakom, Boyer-Moore algoritam uzima maksimum između oba pomaka. Opisane pomake sada ćemo definirati sasvim formalno.

Pomak dobrog sufiksa ćemo pohraniti u tablici GS duljine $m + 1$. Najprije definirajmo dva uvjeta:

$$\begin{aligned} \text{condSfx}(i, s) &: \text{ za svaki } k,\ i < k \leqslant m,\ \text{vrijedi } s \geqslant k \text{ ili } P[k - s] = P[k] \\ \text{condDiff}(i, s) &: \text{ ako } s < i,\ \text{onda } P[i - s] \neq P[i]. \end{aligned}$$

Sada, za svaki $1 \leqslant i \leqslant m$ imamo

$$\text{GS}[i + 1] = \min\{s > 0 : \text{condSfx}(i, s) \text{ te } \text{condDiff}(i, s) \text{ su ispunjeni}\}.$$

Dodatno definiramo da je GS[1] jednak duljini perioda uzorka.

Pomak lošeg znaka je pohranjen u tablici BC veličine abecede, a definiran je za svaki $c \in \Sigma$ na sljedeći način

$$\text{BC}[c] = \begin{cases} \min\{i : 1 \leqslant i < m,\ P[m - i] = c\} & \text{ako se } c \text{ pojavuljuje u } P \\ m & \text{inače.} \end{cases}$$

Navedene funkcije mogu biti unaprijed izračunate u vremenskoj složenosti $O(m + |\Sigma|)$ (konkretna implementacija je dana u [1]). Sada dajemo pseudokod Boyer-Moore algoritma koristeći pomoćne funkcije IzračunajGS i IzračunajBC koje računaju pomak dobrog prefiksa, odnosno lošeg znaka.

Algoritam 4 BoyerMoore (T, P)

```
1:  n ← T.length
2:  m ← P.length
3:  GS ← IzračunajGS(P)
4:  BC ← IzračunajBC(P)
5:  j = 0
6:  while j ⩽ n − m do
7:      for i = m downto 1 do
8:          if P[i] ≠ T[i + j] then
9:              break
10:     if i == 0 then
11:         print „Uzorak se javlja s pomakom” j
12:         j ← j + GS[0]
13:     else
14:         j ← j + max {GS[i], BC[T[i + j]] − m + i}
```

Primijetimo da je pomak lošeg znaka relativno jednostavan. Spomenuta pojednostavljena verzija Boyer-Mooreovog algoritma se upravo odnosi na korištenje samo pomaka lošeg znaka, a takav algoritam je objavio Nigel Horspool 1980. godine i poznat je pod nazivom Boyer-Moore-Horspool algoritam.

Poglavlje 3

Algoritmi za aproksimativno podudaranje

U uvodnom poglavlju smo već definirali *edit*-udaljenost te problem aproksimativnog podudaranja stringova. Sada ćemo objasniti kako se računa *edit*-udaljenost dvaju stringova pomoću dinamičkog programiranja. Iz tog algoritma će zatim slijediti i rješenje za naš problem pronalaženja podnizova teksta T koji su najbliži uzorku P.

3.1 *Edit*-udaljenost

Kako ćemo u nastavku uvesti formalne oznake potrebne za dokazivanje korektnosti algoritma, sada ćemo ponoviti neke već spomenute definicije.

Operacija uređivanja je par $(a, b) \neq (\varepsilon, \varepsilon)$ stringova duljine manje ili jednake 1. Označavat ćemo je s $a \to b$. String B se dobije primjenom operacije $a \to b$ na string A, u oznaci $A \Rightarrow B$ *pomoću* $a \to b$, ako je $A = uav$ i $B = ubv$, za neke stringove u i v. Operaciju $a \to b$ zovemo *operacijom zamjene* ako su $a \neq \varepsilon$, $b \neq \varepsilon$; *operacijom brisanja* ako je $b = \varepsilon$; *operacijom ubacivanja* ako je $a = \varepsilon$.

Neka je S niz $s_1, \ldots, s_m$ operacija uređivanja. *S-izvođenje iz A u B* je niz stringova $A_0, \ldots, A_m$ takvih da je

$$A = A_0, \quad B = A_m, \quad A_{i-1} \Rightarrow A_i \text{ pomoću } s_i, \text{ za } 1 \leqslant i \leqslant m.$$

Kažemo da S *prevodi* A u B ako postoji neko S-izvođenje iz A u B.

Neka je γ proizvoljna *funkcija troška* koja svakoj operaciji uređivanja $a \to b$ pridružuje nenegativan realan broj $\gamma(a \to b)$. Proširimo funkciju γ na niz operacija uređivanja $S = s_1, \ldots, s_m$ na prirodan način tako da definiramo

$$\gamma(S) = \sum_{i=1}^{m} \gamma(s_i).$$

(Ako je $m = 0$, tada stavimo da je $\gamma(S) = 0$.) Sada definiramo *edit-udaljenost od stringa A do stringa B* kao minimalni trošak svih nizova operacija uređivanja koji A prevode u B, odnosno formalno zapisano imamo da je

$$\delta(A, B) = \min\{\gamma(S) : S \text{ je niz operacija uređivanja koji prevodi } A \text{ u } B\}.$$

U nastavku ćemo pretpostaviti da vrijedi $\gamma(a \to b) = \delta(a, b)$. Ekvivalentno, mogli smo pretpostaviti da vrijedi $\gamma(a \to a) = 0$ i $\gamma(a \to b) + \gamma(b \to c) \geqslant \gamma(a \to c)$. Primijetimo da ako je funkcija δ simetrična i strogo pozitivna na svim operacijama $a \to b$, $a \neq b$, tada δ predstavlja metriku na prostoru svih znakovnih nizova – zbog toga je i nazivamo (*edit*) udaljenost.

3.2 Trag

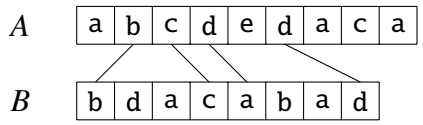
Kako bi pojednostavili problem pronalaženja *edit*-udaljenosti između stringova A i B, definirat ćemo funkciju troška na tzv. *tragovima* i pokazati da tragovi imaju sljedeća dva svojstva:

(T1) za svaki trag T od A do B, postoji niz operacija uređivanja koji prevodi A u B tako da vrijedi $\gamma(S) = \text{cost}(T)$

(T2) za svaki niz operacija uređivanja S koji prevodi A u B, postoji trag T od A do B tako da vrijedi $\text{cost}(T) \leqslant \gamma(S)$.

Stoga, $\delta(A, B)$ je minimalni trošak traga od A do B te ćemo se u nastavku baviti pronalaženju istoga.

U nastavku ćemo dati preciznu definiciju traga te dokazati njegova navedena svojstva. Intuitivno, trag možemo shvatiti kao opis niza operacija uređivanja S koji A transformira u B, zanemarujući pritom redoslijed obavljanja operacija i redundantnosti u S. Promotrimo sliku 3.1.



Slika 3.1: Grafički prikaz traga.

U njoj linija koja spaja dva znaka $A[i]$ i $B[j]$ znači da je $B[j]$ izveden iz $A[i]$, bilo direktno ako je $A[i] = B[j]$, ili indirektno ako S primjenjuje jednu ili više operacija uređivanja na $A[i]$. Pozicije u stringu A koje ne dodiruje nijedna linija predstavljaju znakove od A koji su obrisani. S druge strane, pozicije u stringu B koje ne dodiruje nijedna linija predstavljaju znakove od B koji su umetnuti u A.

Definicija 3.2.1. *Trag od A do B* je uređena trojka (T, A, B) gdje je T skup uređenih parova (i, j) koji zadovoljava sljedeće:

(1) $1 \leqslant i \leqslant |A|$ te $1 \leqslant j \leqslant |B|$

(2) za svaka dva različita para (i_1, j_1) te (i_2, j_2) iz T vrijedi

(2a) $i_1 \neq i_2$ te $j_1 \neq j_2$

(2b) $i_1 < j_1$ ako i samo ako $j_1 < j_2$.

Napomena 3.2.2. 1. Par (i, j) opisuje liniju koja spaja pozicije i u stringu A te j u stringu B. Kažemo da (i, j) *dodiruje* te pozicije.

2. Uvjet (1) osigurava da linije zapravo dodiruju znakove iz odgovarajućih stringova.

3. Uvjet (2a) osigurava da se pozicija svakog stringa dodiruje najviše jednom, dok uvjet (2b) osigurava da se linije ne križaju.

4. Kada bude iz konteksta bilo jasno o kojim stringovima se radi, tada ćemo trag (T, A, B) kraće označavati s T.

Sada ćemo definirati funkciju troška na tragovima iz čije definicije će direktno slijedi svojstvo (T1).

Definicija 3.2.3. Neka je T trag od A do B. Neka su I i J skupovi pozicija u A i B, respektivno, koje ne dodiruje nijedna linija iz T. *Trošak traga*, u oznaci cost, definiramo s

$$\text{cost}(T) = \sum_{(i,j)\in T} \gamma(A[i] \to B[j]) + \sum_{i\in I} \gamma(A[i] \to \varepsilon) + \sum_{j\in J} \gamma(\varepsilon \to B[j]).$$

Trošak traga T je zapravo trošak niza operacija uređivanja koji prevodi A u B, a sastoji se od operacija zamjene $A[i] \to B[j]$, za svaki $(i, j) \in T$, operacija brisanja $A[i] \to \varepsilon$, za svaki $i \in I$, te operacija ubacivanja $\varepsilon \to B[j]$, za svaki $j \in J$.

Tragovi se mogu prirodno komponirati. Ako su T_1 trag od A do B i T_2 trag od B do C, tada je $T_1 \circ T_2$ trag od A do C. Pri tome $\circ$ označava uobičajenu kompoziciju relacija, odnosno

$$T_1 \circ T_2 = \{(i, j) : (i, k) \in T_1 \text{ i } (k, j) \in T_2 \text{ za neki } k\}.$$

Sljedeća lema jednostavno slijedi iz činjenice da je $\gamma(a \to b) = \delta(a, b)$.

Lema 3.2.4. *Vrijedi*

$$cost(T_1 \circ T_2) \leqslant cost(T_1) + cost(T_2),$$

gdje su T_1 trag od A do B i T_2 trag od B do C.

Sada ćemo matematičkom indukcijom po duljini niza operacija uređivanja m pokazati da vrijedi svojstvo (P2). Preciznije, za niz operacija uređivanja $S = s_1, \ldots, s_m$ i S-izvođenje $(A_0, \ldots, A_m)$ iz A u B, trebamo pronaći trag T od A_0 do A_m za koji vrijedi $\text{cost}(T) \leqslant \gamma(S)$.

Ako je $m = 0$, traženi trag T definiramo na sljedeći način:

$$T = \{(i, i) : 1 \leqslant i \leqslant |A_0|\}.$$

Tada po definiciji imamo da je $\text{cost}(T) = 0 = \gamma(S)$ pa smo time pokazali bazu indukcije.

Ako je $m > 0$, tada, po pretpostavci indukcije, postoji trag T_1 od A_0 do A_{m-1} takav da je $\text{cost}(T_1) \leqslant \gamma(s_1, \ldots, s_{m-1})$. Kako vrijedi $A_{m-1} \Rightarrow A_m$ pomoću $s_m = a \to b$, to postoje stringovi u i v takvi da $A_{m-1} = uav$ i $A_m = ubv$. Neka je T_2 trag od A_{m-1} do A_m definiran s

$$T_2 = \{(i, i) : 1 \leqslant i \leqslant |u|\} \cup \{(i, i + d) : |ua| + 1 \leqslant i \leqslant |A_{m-1}|\} \cup L,$$

pri čemu je $d = |b| - |a| \in \{-1, 0, 1\}$ te

$$L = \begin{cases} \{(|u| + 1, |u| + 1)\} & \text{ako je } s_m \text{ operacija zamjene} \\ \emptyset & \text{inače.} \end{cases}$$

Očito je T_2 trag i vrijedi $\text{cost}(T_2) = \gamma(a \to b) = \gamma(s_m)$. Sada definiramo $T = T_1 \circ T_2$. T je trag od A_0 do A_m, te po lemi 3.2.4 imamo

$$\text{cost}(T) \leqslant \text{cost}(T_1) + \text{cost}(T_2) \leqslant \gamma(s_1, \ldots, s_{m-1}) + \gamma(s_m) = \gamma(S),$$

te smo time pokazali korak indukcije. Dakle, po principu matematičke indukcije, zaključujemo da svojstvo (P2) vrijedi za sve nizove operacija uređivanja.

Istaknimo još jednom da zbog svojstava (P1) i (P2) vrijedi sljedeći teorem.

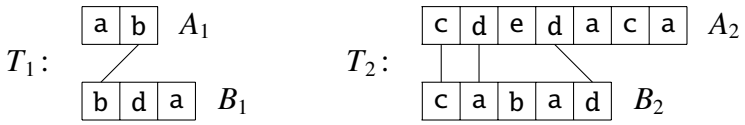
Teorem 3.2.5. *Vrijedi* $\delta(A, B) = \min\{\text{cost}(T) : T \text{ je trag od } A \text{ do } B\}$.

3.3 Računanje *edit*-udaljenosti

Vratimo se sada na grafičku reprezentaciju traga T od A do B. Neka je $A = A_1A_2$, $B = B_1B_2$, i pretpostavimo da nijedna linija od T ne spaja znakove stringova A_i te B_j, $i \neq j$, $i, j \in \{1, 2\}$. Tada se trag (T, A, B) može razdvojiti na tragove (T_1, A_1, B_1) i (T_2, A_2, B_2) kako je pokazano na slici 3.2.

Nadalje, vrijedi $\text{cost}(T) = \text{cost}(T_1) + \text{cost}(T_2)$, pa ako je T trag s minimalnim troškom od A do B, tada je i T_i trag s minimalnim troškom od A_i do B_i, $i \in \{1, 2\}$.

Svaki trag se može razdvojiti na tragove T_1 i T_2, kako je gore opisano, tako da su duljine stringova A_2 i B_2 najviše 1, ali nisu obje istovremeno 0. Upravo to je ideja za sljedeći



Slika 3.2: Razdvajanje tragova.

teorem na kojem je zasnovan algoritam za računanje *edit*-udaljenosti koji u nastavku iznosimo.

Neka su A i B stringovi. Označimo s

$$D[i, j] = \delta(A[1..i], B[1..j]),$$

gdje su $0 \leqslant i \leqslant |A|$, $0 \leqslant j \leqslant |B|$. Primijetimo da je po teoremu 3.2.5 $D[i, j]$ isto tako i najmanji trošak traga od $A[1..i]$ do $B[1..j]$.

Teorem 3.3.1. *Vrijedi*

$$\begin{aligned} D[i, j] = \min\{&D[i-1, j-1] + \gamma(A[i] \to B[j]),\\ &D[i-1, j] + \gamma(A[i] \to \varepsilon),\\ &D[i, j-1] + \gamma(\varepsilon \to B[j])\} \end{aligned}$$

za sve i, j, $1 \leqslant i \leqslant |A|$, $1 \leqslant j \leqslant |B|$.

Dokaz. Neka je T trag s minimalnim troškom od $A[1..i]$ do $B[1..j]$. Ako su znakovi $A[i]$ i $B[j]$ dodirnuti nekom linijom, tada oba moraju biti dodirnuti istom linijom jer bi se u protivnom te linije sjekle. Zbog toga jedan od iduća tri slučaja mora vrijediti.

Slučaj 1. $A[i]$ i $B[j]$ su spojeni linijom iz T, tj. $(i, j) \in T$. Tada je trošak od T jednak $m_1 = D[i-1, j-1] + \gamma(A[i] \to B[j])$ što odgovara trošku transformiranja stringa $A[1..i-1]$ u $B[1..j-1]$ te dodatno trošku mijenjanja $A[i]$ u $B[j]$.

Slučaj 2. $A[i]$ nije dodirnut nijednom linijom iz T. Tada je trošak od T jednak $m_2 = D[i-1, j] + \gamma(A[i] \to \varepsilon)$ što odgovara trošku transformiranja $A[1..i-1]$ u $B[1..j]$ i brisanja znaka $A[i]$.

Slučaj 3. $B[j]$ nije dodirnut nijednom linijom iz T. Tada je trošak od T jednak $m_3 = D[i, j-1] + \gamma(\varepsilon \to B[j])$ što odgovara trošku transformiranja $A[1..i]$ u $B[1..j-1]$ i umetanja znaka $B[j]$.

Kako jedan od navedena tri slučaja mora vrijediti i $D[i, j]$ mora biti minimalni trošak, tada vrijedi $D[i, j] = \min\{m_1, m_2, m_3\}$. □

Teorem 3.3.2. *Za* $1 \leqslant i \leqslant |A|$, $1 \leqslant j \leqslant |B|$ *vrijedi*

$$D[0,0] = 0$$
$$D[i,0] = \sum_{r=1}^{i} \gamma(A[r] \to \varepsilon)$$
$$D[0,j] = \sum_{r=1}^{j} \gamma(\varepsilon \to B[r]).$$

Dokaz. Jedini trag, a shodno tome i s najmanjim troškom, od $A[1..i]$ do $B[1..j]$ kada su i ili j jednaki nuli je $\emptyset$. Tvrdnje teorema sada direktno slijede iz definicije troška traga. □

Teoremi 3.2.5 i 3.3.2 zapravo dokazuju da algoritam 5 ispravno računa $D[i, j]$ za sve za $0 \leqslant i \leqslant |A|$, $0 \leqslant j \leqslant |B|$.

Algoritam 5 WagnerFischer *(A, B)*

```
1:  n ← A.length
2:  m ← B.length
3:  D[0,0] ← 0
4:  for i = 1 to n do
5:      D[i,0] ← D[i−1,0] + γ(A[i] → ε)
6:  for j = 1 to m do
7:      D[0,j] ← D[0,j−1] + γ(ε → B[j])
8:  for i = 1 to n do
9:      for j = 1 to m do
10:         m1 ← D[i−1,j−1] + γ(A[i] → B[j])
11:         m2 ← D[i−1,j] + γ(A[i] → ε)
12:         m3 ← D[i,j−1] + γ(ε → B[j])
13:         D[i,j] ← min{m1, m2, m3}
```

Primijetimo da ovim algoritmom nismo zapravo riješili problem aproksimativnog podudaranja stringova koji je naveden u definiciji 1.2.3 jer algoritam 5 računa *edit*-udaljenosti samo između svih prefiksa teksta i svih prefiksa uzorka. Za rješavanje problema aproksimativnog podudaranja znakovnih nizova potrebno je odrediti najmanju *edit*-udaljenost između uzorka i *nekog* podniza teksta (koji ne mora nužno biti prefiks teksta). Kako je broj svih podnizova teksta reda veličine $O(n^2)$, naivno rješenje spomenutog problema bi rezultiralo vremenskom složenošću $O(mn^3)$ – izračunamo *edit*-udaljenosti između uzorka i svakog podniza teksta te izaberemo najmanju.

Bolje rješenje je predložio Peter H. Sellers. U nastavku ćemo iznijeti samo glavne ideje njegova pristupa, dok se detalji mogu pronaći u [9].

Sellersovo rješenje se zasniva na sitnoj modifikaciji problema: za sve $1 \leqslant i \leqslant n$, $1 \leqslant j \leqslant m$ potrebno je „proći" svim podnizovima teksta koji završavaju na poziciji i te odrediti koji od njih ima najmanju *edit*-udaljenost do prefiksa uzorka $P[1..j]$. Označimo takvu minimalnu *edit*-udaljenost s $E(i, j)$. Određivanje matrice E je slično određivanju *edit*-udaljenosti između dvaju stringova (algoritam 5). Jedina razlika je u inicijalizaciji matrice D (koja odgovara matrici E u ovom kontekstu) kojoj je potrebno prvi stupac postaviti na nulu, tj. $D[i, 0] = 0$, za sve $0 \leqslant i \leqslant n$.

Rješenje originalnog problema aproksimativnog podudaranja stringova sada dobijemo tako da odredimo sve podnizove teksta za koje je $E(i, m)$ minimalno. Da bi to mogli napraviti, potrebno je zapamtiti „put kojim smo došli" do $E(i, m)$, odnosno koju od vrijednosti $E(i - 1, j - 1)$, $E(i - 1, j)$ ili $E(i, j - 1)$ smo koristili u određivanju vrijednosti $E(i, j)$ (vidi linije 10-13 algoritma 5). Pretpostavimo da je minimalna vrijednost u zadnjem stupcu $E(x_2, m)$ i da prateći put izračunavanja do prvog stupca stignemo u $E(x_1, 0)$. Tada podniz $T[x_1 + 1..x_2]$ ima minimalnu *edit*-udaljenost do uzorka P.

Poglavlje 4

Algoritmi za podudaranje sa zamjenskim znakovima

U mnogim situacijama prirodno se javlja potreba za pronalaženjem svih znakovnih nizova s nekim svojstvom koje se može opisati znakovima $\star$ ili **?**. Primjerice, u datotečnom sustavu htjeli bismo pronaći sve tekstualne datoteke, tj. datoteke sa nastavkom `.txt`. Problem se zapravo svodi na traženje datoteka kojima se ime podudara sa znakovnim nizom $\star$`.txt`. Drugi primjer bi mogao biti pronalaženje riječi određene duljine sa zadanim korijenom u tekstu napisanom na hrvatskom jeziku. Zbog specifičnosti samog jezika (promjene po padežima, glasovne promjene, ...) nužno je koristiti jedan ili više znakova **?** nadovezanih na korijen riječi. U ovom poglavlju opisat ćemo načine na koji se ovakvi problemi mogu riješiti.

4.1 Regularni izrazi

Intuitivno, regularni izrazi su nizovi znakova koji definiraju traženi uzorak u tekstu. Pojavili su se u 50-tim godinama 20. stoljeća kada je Stephen Kleene formalizirao opis regularnih jezika. Danas imaju podršku u svim glavnim programskim jezicima te se stoga nameću kao prirodno rješenje problema podudaranja stringova sa zamjenskim znakovima. Međutim, njihov nedostatak je što su oni mnogo ekspresivniji od problema kojeg mi rješavamo te stoga imaju relativno veliku vremensku složenost.

Sada dajemo formalnu definiciju regularnih izraza.

Definicija 4.1.1. Neka je Σ abeceda. Regularni izrazi definiraju se rekurzivno nad abecedom Σ. Uz svako rekurzivno pravilo naveden je i jezik određen tim pravilom. Rekurzivna pravila za regularne izraze su:

1) $\emptyset$ jest regularni izraz i označava jezik $L(\emptyset) = \{\}$.

2) ε jest regularni izraz i označava jezik $L(\varepsilon) = \{\varepsilon\}$.

3) Za svaki $a \in \Sigma$, a jest regularni izraz i označava jezik $L(a) = \{a\}$.

4) Ako su r i s regularni izrazi koji označavaju jezike $L(r)$ i $L(s)$, onda:

   a) $(r) + (s)$ jest regularni izraz koji označava jezik $L((r) + (s)) = L(r) \cup L(s)$ koji nastaje unijom jezika $L(r)$ i $L(s)$. Često se koristi i oznaka $(r)|(s)$.

   b) $(r)(s)$ jest regularni izraz koji označava jezik $L((r)(s)) = L(r)L(s)$ koji nastaje nadovezivanjem jezika $L(r)$ i $L(s)$.

   c) $(r)^*$ jest regularni izraz koji označava jezik $L((r)^*) = L(r)^*$ koji nastaje primjenom Kleeneovog operatora nad jezikom $L(r)$.

Primjer 4.1.2. Regularni izraz $r_1 = (\mathtt{0} + \mathtt{1})(\mathtt{0} + \mathtt{1})$ definira jezik $L(r_1) = \{\mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{11}\}$. Regularni izraz $r_2 = (\mathtt{0} + \mathtt{1})^*$ definira jezik u kojem su prazni niz ε te svi nizovi znakova 0 i 1.

Regularni izraz možemo iskoristiti za rješenje našeg problema na dva bitno različita načina od kojih svaki ima svojih prednosti i mana.

Za svaki regularni izraz r se može izgraditi ekvivalentan nedeterministički konačni automat N koji se zatim prevodi u ekvivalentan deterministički konačni automat D takav da je $L(r) = L(N) = L(D)$. Takva konstrukcija je vremenske složenosti $O(2^{|r|})$ i možemo je smatrati pretprocesiranjem, dok se „pravi" algoritam za podudaranje izvršava u vremenskoj složenosti $O(n)$ gdje je n duljina teksta.

Drugi pristup je da se nedeterministički konačni automat N gradi za svaki pojedini upit. Na taj način izbjegavamo eksplicitnu konstrukciju determinističog konačnog automata D, no tada je vremenska složenost algoritma $O(mn)$.

4.2 Optimizacija rekurzivnog algoritma

Za postizanje efikasnog rješenja problema podudaranja stringova sa zamjenskim znakovima najveću prepreku predstavlja znak $\star$. Da bi se u to uvjerili promotrimo algoritam 6 koji rješava navedeni problem u slučaju kada je jedini dozvoljeni zamjenski znak **?**.

Očito se tekst i uzorak neće podudarati ako su različitih duljina (linije 3-4). U protivnom, dovoljno je provjeriti da se svi odgovarajući znakovi ili podudaraju ili je znak uzorka jednak **?** (linije 5-8). Tekst i uzorak se ne podudaraju ako i samo ako naiđemo na par znakova za koji ne vrijedi navedeno svojstvo. Navedeni algoritam je linearne vremenske složenosti u broju znakova teksta.

Algoritam 6 JEDNOSTAVNI *(T, P)*

```
1: n ← T.length
2: m ← P.length
3: if n ≠ m then
4:     return FALSE
5: for i = 1 to n do
6:     if T[i] ≠ P[j] and P[j] ≠ ? then
7:         return FALSE
8: return TRUE
```

Komplikacije koje nastaju uvođenjem zamjenskog znaka $\star$ proizlaze iz činjenice da ne znamo unaprijed hoće li taj znak zamijeniti 0, 1 ili više znakova teksta. Ispitivanje svih mogućnosti navodi na rekurzivno rješenje dano algoritmom 7.

Algoritam 7 REKURZIVNI *(T, P, i, j)*

```
1: n ← T.length
2: m ← P.length
3: if j > m then
4:     return i > n
5: if i > n then
6:     return P[j] == ⋆ and j == m
7: if P[j] == ? or T[i] == P[j] then
8:     return REKURZIVNI(T, P, i + 1, j + 1)
9: if P[j] == ⋆ then
10:    return REKURZIVNI(T, P, i, j + 1) or REKURZIVNI(T, P, i + 1, j)
11: return FALSE
```

Algoritam utvrđuje podudaraju li se stringovi $T[i..n]$ i $P[j..m]$, a poziva se s ulaznim vrijednostima $i = j = 1$. U nastavku ćemo dokazati konačnost te korektnost algoritma.

Lema 4.2.1. *Algoritam 7 staje u konačno mnogo koraka.*

Dokaz. Iz početnih uvjeta (linije 3-6), vidimo da algoritam staje kada je $i > n$ ili $j > m$. Ukoliko promotrimo izraz $k := n - i + m - j \in \mathbb{N}$, možemo zaključiti da će algoritam stati u slučaju $k < 0$. U svakom rekurzivnom pozivu (linije 8, 10) inkrementira se i ili j ili obje vrijednosti istovremeno. Dakle, svaki rekurzivni poziv će smanjiti k barem za 1, te će u konačno mnogo koraka k postati negativan. Kako pojedini rekurzivni poziv može generirati najviše dva nova rekurzivna poziva, zaključujemo da vrijedi tvrdnja. □

Za potrebe sljedećeg dokaza pretpostavit ćemo da ne postoje dva uzastopna znaka ⋆ u uzorku. Ta pretpostavka nije ograničavajuća jer možemo u linearnom vremenu sva uzastopna pojavljivanja znaka ⋆ zamijeniti jednim znakom ⋆, a pritom će se dobiveni uzorak podudarati s originalnim.

Propozicija 4.2.2. *Algoritam 7 je korektan, odnosno strigovi $T[i..n]$ i $P[j..m]$ se podudaraju ako i samo ako algoritam 7 vrati* TRUE .

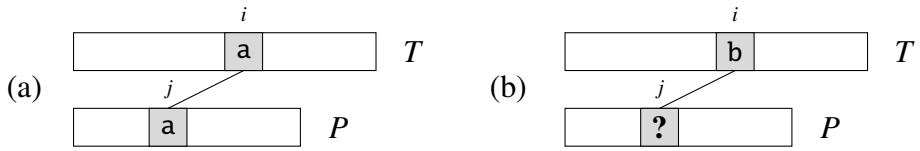
Dokaz. Najprije dokažimo korektnost početnih uvjeta. (Napomenimo da pri referenciranju na tekst podrazumijevamo promatrani tekst, odnosno $T[i..n]$. Isto vrijedi i za uzorak.)

U slučaju $j > m$ (linija 3), uzorak je zapravo prazan string te se stoga podudara s tekstom ako i samo ako je tekst također prazan (linija 4).

Inače, postoji barem jedan znak u uzorku. Ako vrijedi $i > n$ (linija 5), tada se tekst (koji je prazan string) i uzorak podudaraju ako i samo ako je jedini znak uzorka jednak ⋆ (linija 6). Zaista, u protivnom postoji znak uzorka $a \in \Sigma \cup \{?\}$ (koristimo činjenicu da ne postoje dva uzastopna znaka ⋆) te se stoga uzorak očito ne podudara s tekstom.

Ostaje još dokazati korektnost rekurzivnih poziva, tj. da „veći" problem ispravno svodimo na „manji" problem. Razlikujemo dva slučaja.

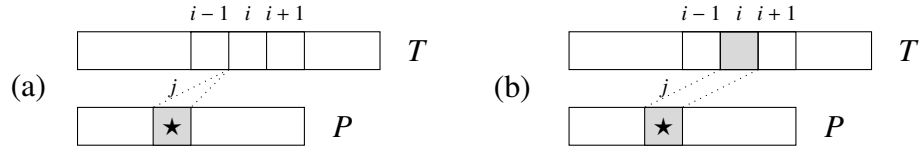
U prvom slučaju, prvi znakovi teksta i uzorka su isti ili je znak uzorka jednak **?** (vidi sliku 4.1). Tada se tekst $T[i..n]$ i uzorak $P[j..m]$ podudaraju ako i samo ako se podudaraju i $T[i+1..n]$ te $P[j+1..m]$ (linije 7-8).



Slika 4.1: Odgovarajući znakovi $T[i]$ i $P[j]$ se podudaraju **(a)** doslovno ili **(b)** u „širem" smislu zbog $P[j] =$ **?** .

Drugi slučaj nastupa ako je prvi znak uzorka jednak ⋆ (vidi sliku 4.2). Tada postupamo u skladu sa značenjem znaka ⋆, odnosno ispitujemo mogućnosti da on zamjenjuje nula ili više znakova teksta. Zamjena nula znakova teksta se odražava rekurzivnim pozivom Rekurzivni$(T, P, i, j+1)$ gdje zapravo zanemarujemo („preskačemo") znak ⋆. Rekurzivni poziv Rekurzivni$(T, P, i+1, j)$ ostvaruje zamjenu više znakova teksta znakom ⋆ – preciznije izravno se znak $T[i]$ zamjenjuje znakom ⋆, ali se kroz (potencijalne) iduće rekurzivne pozive ostvaruje zamjena s dva ili više znakova teksta (linije 9-10).

Dakle, rekurzivni pozivi su ispravni. Još samo preostaje primijetiti slučaj $P[j] \in \Sigma$, $T[i] \neq P[j]$, u kojem se prvi znakovi teksta i uzorka razlikuju te se stoga radi o nepodudaranju (linija 11). □



Slika 4.2: **(a)** Znak $P[j] = \star$ ne zamjenjuje niti jedan znak teksta. **(b)** Znak $P[j] = \star$ zamjenjuje znak teksta $T[i]$.

Kao i sa mnogim rekurzivnim algoritmima, problem ovakvog rješenja je njegova velika vremenska složenost. Sasvim konkretno, vrijedi sljedeća propozicija.

Propozicija 4.2.3. *Vremenska složenost algoritma 7 u najgorem slučaju je jednaka* $\Theta(2^n)$.

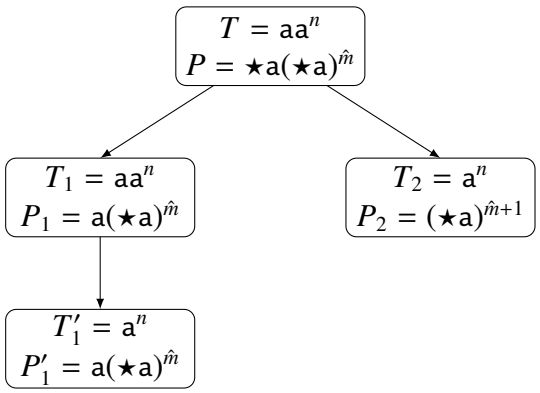
Dokaz. Označimo s $t_{(n,m)}$ vremensku složenost algoritma za ulaz $T = \mathtt{a}^n$, $P = (\star\mathtt{a})^m$, $n, m \in \mathbb{N}_0$. Principom matematičke indukcije ćemo pokazati da za sve $n, m \in \mathbb{N}_0$, $m \geqslant n$ vrijedi

$$t_{(n,m)} = c2^n, \tag{4.1}$$

za neki $c > 0$, iz čega će slijediti tvrdnja propozicije kao posebni slučaj $m = n$.

Baza indukcije $n = 0$, $m \in \mathbb{N}_0$ je trivijalno ispunjena jer se radi o početnom uvjetu algoritma.

Pretpostavimo da (4.1) vrijedi za neki $n \in \mathbb{N}_0$ te za sve $m \geqslant n$. Promotrimo ponašanje algoritma za $T = \mathtt{aa}^n$, $P = \star\mathtt{a}(\star\mathtt{a})^{\hat{m}}$, gdje je $\hat{m} \geqslant n$ (vidi sliku 4.3). Iz samog koda vidimo da će poziv Rekurzivni$(T, P, 1, 1)$ rezultirati dvama pozivima Rekurzivni$(T, P, 1, 2)$ te Rekurzivni$(T, P, 2, 1)$. U prvom pozivu pripadni stringovi su $T_1 = \mathtt{aa}^n$, $P_1 = \mathtt{a}(\star\mathtt{a})^{\hat{m}}$ što



Slika 4.3: Prikaz rekurzivnih poziva za ulaz $T = \mathtt{a}^{n+1}$ i $P = (\star\mathtt{a})^{\hat{m}+1}$.

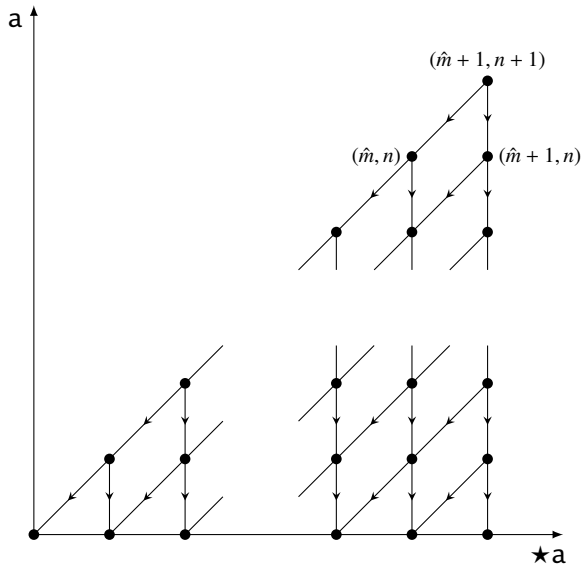
se u idućem pozivu svede na $T'_1 = \mathtt{a}^n$, $P'_1 = (\star\mathtt{a})^{\hat{m}}$. Za drugi poziv pak vrijedi $T_2 = \mathtt{a}^n$,

$P_2 = (\star\mathtt{a})^{\hat{m}+1}$. Na parove (T_1', P_1') i (T_2, P_2) možemo primijeniti pretpostavku matematičke indukcije te tako dobijemo

$$t_{(n+1,\hat{m}+1)} = t_{(n,\hat{m})} + t_{(n,\hat{m}+1)} = c2^{n+1}$$

čime je tvrdnja dokazana. □

Rekurzivne pozive možemo grafički prikazati kao na slici 4.4. Iz same slike je uočljiv



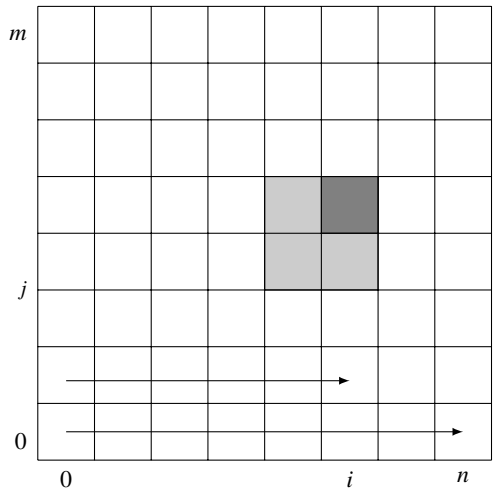
Slika 4.4: Grafički prikaz rekurzivnih poziva za ulaze oblika $T = \mathtt{a}^N$ i $P = (\star\mathtt{a})^M$. x-os označava broj pojavljivanja podniza $\star\mathtt{a}$ u uzorku, dok y-os označava broj znakova $\mathtt{a}$ u tekstu. Strelicama su označeni direktni rekurzivni pozivi.

nedostatak ovakve implementacije rekurzivnog algoritma – iste potprobleme rješava mnogo puta. To možemo popraviti standardnom tehnikom *memoizacije*. Ona uključuje spremanje već izračunatih potproblema tako da se prilikom nailaska na isti problem rješenje samo „pročita" iz memorije. Potprobleme rješava od vrha prema dnu (eng. *top-down*) na rekurzivan način. Iako su rekurzivni algoritmi lako čitljivi i koncizni, oni su isto tako najčešće sporiji u odnosu na ekvivalentne iterativne algoritme zbog same prirode rekurzivnih poziva. Da bi uklonili i taj nedostatak manje potprobleme je potrebno rješavati iterativno od dna prema vrhu (eng. *bottom-up*) te time dobivamo *dinamičko programiranje*. Algoritam 8 ilustrira upravo takvo rješenje problema podudaranja znakovnih nizova sa zamjenskim znakovima.

Algoritam 8 DINAMIČKOPROGRAMIRANJE *(T, P)*

```
1:  n ← T.length
2:  m ← P.length
3:  for i = 0 to n do
4:      DP[i][0] ← i == 0
5:  for j = 1 to m do
6:      DP[0][j] ← DP[0][j − 1] and P[j] == ⋆
7:      for i = 1 to n do
8:          if P[j] ≠ ⋆ then
9:              DP[i][j] ← DP[i − 1][j − 1] and (P[j] == ? or T[i] == P[j])
10:         else
11:             DP[i][j] ← DP[i − 1][j] or DP[i][j − 1]
12: return DP[n][m]
```

U algoritmu *DP* označava dvodimenzionalno polje s vrijednostima TRUE ili FALSE kojim pamtimo rješenja potproblema, tj. $DP[i][j]$ označava podudaraju li se stringovi $T[1..i]$ i $P[1..j]$. Najprije izvršimo očitu inicijalizaciju polja *DP* (linije 3-4). Nakon toga slijedi već opisani način rada algoritma (vidi propoziciju 4.2.2) (linije 5-11). Valja samo primijetiti da su nam za izračun vrijednosti $DP[i][j]$ potrebni najviše $DP[i-1][j-1]$, $DP[i-1][j]$ i $DP[i][j-1]$ koji su zbog načina popunjavanja tablice uvijek postavljeni na ispravne vrijednosti. (vidi sliku 4.5). Konačno, kao rješenje vratimo $DP[n][m]$ (linija 12).



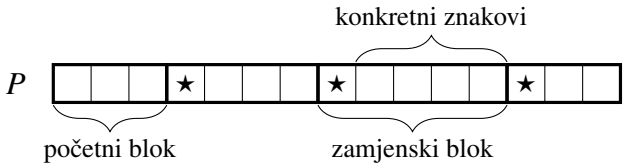
Slika 4.5: Popunjavanje tablice *DP* se obavlja slijeva nadesno i od dolje prema gore. Stoga u trenutku izračunavanja vrijednosti $DP[i][j]$ (označeno tamno sjenčano) imamo ispravno postavljene vrijednosti sve vrijednosti koji su nam potrebne (označeno svjetlo sjenčano).

Vremenska i prostorna složenost ovakvog algoritma je $O(mn)$. Pritom je prostornu složenost moguće smanjiti na $O(\min\{m, n\})$ jer trebamo pamtiti samo dva posljednja stupca ili retka matrice DP.

4.3 Iterativni algoritam s *backtrackingom*

Sada ćemo pokazati još jedno rješenje problema podudaranja znakovnih nizova sa zamjenskim znakovima. Opet ćemo bez smanjenja općenitosti pretpostaviti da se u uzorku ne pojavljuju dva uzastopna znaka $\star$. Radi jednostavnosti znakove iz $\Sigma \cup \{?\}$ ćemo zvati *konkretnim* znakovima aludirajući na to da oni zahtijevaju postojanje točno jednog znaka u tekstu.

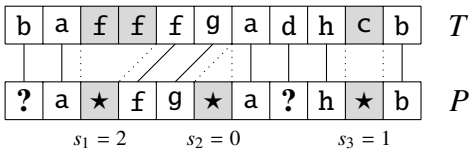
Promotrimo malo detaljnije „strukturu" uzorka (vidi sliku 4.6). On može početi s nula ili više konkretnih znakova koje ćemo zajedno nazivati *početnim blokom*. Nakon početnog bloka u uzorku se pojavljuje nula ili više *zamjenskih blokova*. Pritom zamjenski blok je niz znakova $x_1 x_2 \ldots x_k$ gdje je $x_1 = \star$, a x_i je konkretni znak, za $i > 1$. Da bi se tekst i



Slika 4.6: Uzorak se sastoji od početnog bloka (koji može biti i prazan niz), nakon kojeg slijedi nula ili više zamjenskih blokova. Svaki zamjenski blok počinje znakom $\star$ nakon kojeg slijedi nula ili više konkretnih znakova.

uzorak podudarali, prije svega moramo provjeriti pojavljuje li se početni blok kao prefiks teksta. To je lako ostvarivo pomoću algoritma 6. Zatim moramo redom za svaki zamjenski blok pronaći njegov odgovarajući podniz u tekstu, odnosno podniz s kojim se on podudara. Pri tome se opet suočavamo s poteškoćama koje proizlaze iz definicije zamjenskog znaka $\star$ koji se javlja na početku zamjenskog bloka. Naime, ne znamo koliko će on znakova teksta zamijeniti. Stoga krećemo od pretpostavke da on zamjenjuje nula znakova teksta te provjeravamo podudara li se ostatak zamjenskog bloka s odgovarajućim podnizom u tekstu. Provjera je opet jednostavna jer se radi o konkretnim znakovima. Ako je provjera uspjela, nastavljamo s idućom blokom. U protivnom, pretpostavljamo da zamjenski znak $\star$ zamjenjuje jedan znak. Opisanu proceduru nastavljamo sve dok ne uspijemo pronaći podudaranje ili ne „iscrpimo" sve znakove teksta. U potonjem slučaju zaključujemo da se tekst i uzorak ne podudaraju.

Ovakav algoritam podsjeća na metodu *backtrackinga*. Ako uzorak ima K zamjenskih blokova, tada rješenje problema možemo zapravo predstaviti uređenom K-torkom



Slika 4.7: Određivanje broja zamjena za znakove ★. Prvi znak ★ zamjenjuje podniz `ff` stoga je $s_1 = 2$. Slično vrijedi i za ostale znakove ★.

$(s_1, \ldots, s_K)$ gdje s_k označava koliko znakova teksta zamjenjuje znak ★ k-tog zamjenskog bloka, za $k \in \{1, \ldots, K\}$. Očito, za svaki $k \in \{1, \ldots, K\}$, vrijedi

$$0 \leqslant s_k \leqslant n. \tag{4.2}$$

Također ukupna suma broja znakova koje zamjenjuju znakovi ★ mora biti jednaka razlici broja znakova u tekstu i broja konkretnih znakova u uzorku, tj. mora vrijediti

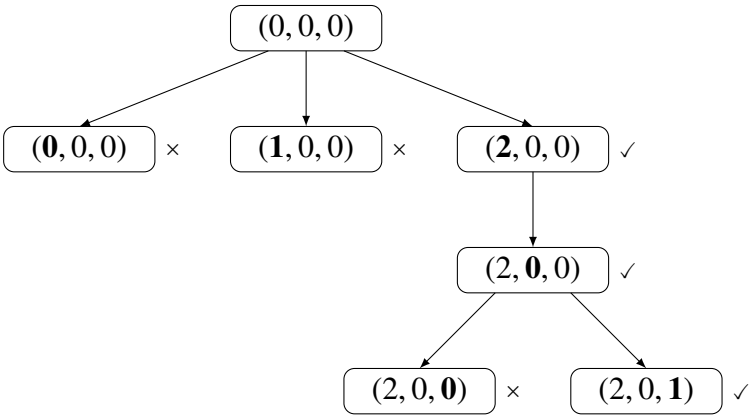
$$\sum_{k=1}^{K} s_k = n - (m - K). \tag{4.3}$$

Osim ovih „tehničkih" uvjeta, mora biti zadovoljen i kriterij podudaranja konkretnih znakova iz zamjenskog bloka, odnosno sasvim konkretno ako je k-ti zamjenskih blok jednak $P[r_k..r_k + k']$, tada mora vrijediti

$$P[r_k + 1..r_k + k'] = T[r'_k + 1..r'_k + k'] \tag{4.4}$$

gdje je $r'_k = \sum_{i=1}^{k} s_k + (r_k - k)$. Tada maloprije opisani algoritam zapravo odgovara generiranju stabla rješenja. Na početku kao prvo dijete korijena stabla stavimo K-torku $(0, \ldots, 0)$ te zatim tražimo vrijednosti koordinata redom od 1 do K tako da su zadovoljeni gore navedeni uvjeti.

Valja primijetiti određene razliku u odnosu na metodu *backtrackinga* opisanu u uvodnom poglavlju. U našem primjeru svaki čvor stabla rješenja predstavlja potencijalno rješenje (a ne samo listovi stabla). Spomenuli smo već da dobar *backtracking* algoritam ne generira cijelo stablo rješenja nego samo one grane koje bi uistinu mogle voditi do konačnog rješenja. Za opisani algoritam vrijedi još i više – jednom kada se „spusti" na k-tu razinu stabla rješenja, više ne generira podstabla visine manje od k. U nastavku dajemo dokaz navedene tvrdnje.



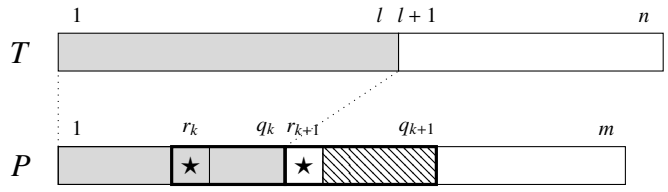
Slika 4.8: Stablo rješenja za primjer sa slike 4.7. Masnim slovima su označene koordinate i vrijednosti koje u svakom čvoru provjeravamo. Neuspješne provjere su označene znakom ×, dok su uspješne označene znakom ✓. Nakon uspješne provjere, „spuštamo " se jedan nivo u stablu i određujemo sljedeću koordinatu.

Propozicija 4.3.1. *Pretpostavimo da je gore opisani algoritam A odredio prvih k koordinata (*$1 \leqslant k < K$*) rješenja problema* $(s_1, \ldots, s_k)$*, ali nije uspio pronaći prvih* $k+1$*. Tada se tekst i uzorak ne podudaraju.*

Dokaz. Označimo i-ti zamjenski blok sa $P[r_i..q_i]$. Algoritam A je odredio prvih k koordinata rješenja problema, pa stoga postoji najmanji $0 \leqslant l \leqslant n$ takav da se podniz teksta $T[1..l]$ podudara s $P[1..q_k]$. Kako algoritam A pokušava odrediti vrijednost pojedine koordinate u uzlaznom poretku (0, 1, . . .), to slijedi da je

$$\sum_{j=1}^{k} s_j = l - (q_k - k).$$

Činjenica da A nije uspio odrediti prvih $k+1$ koordinata rješenja povlači da se konkretni dio $(k+1)$. zamjenskog bloka ne pojavljuje u podnizu teksta $T[l+1..n]$. Kako je l najmanji



Slika 4.9: $T[1..l]$ se podudara s $P[1..q_k]$, no konkretni dio $(k+1)$ zamjenskog bloka se ne pojavljuje u $T[l+1..n]$.

mogući indeks s gore navedenim svojstvom, zaključujemo da vrijedi tvrdnja. □

U nastavku dajemo jednu moguću implementaciju opisa prethodnog algoritma. Zbog propozicije 4.3.1 dovoljno će biti koristiti samo jedan kursor koji će služiti za „vraćanje unatrag" prilikom određivanja k-te koordinate rješenja.

Algoritam 9 ITERATIVNI *(T, P)*

```
1:  n ← T.length
2:  m ← P.length
3:  i ← 1
4:  j ← 1
5:  i_0 ← n + 1
6:  j_0 ← m + 1
7:  while i ⩽ n and j ⩽ m and P[j] ≠ ⋆ do
8:      if T[i] ≠ P[j] and P[j] ≠ ? then
9:          return FALSE
10:     i ← i + 1
11:     j ← j + 1
12: while i ⩽ n do
13:     if P[j] == ⋆ then
14:         j ← j + 1
15:         if j > m then
16:             return TRUE
17:         i_0 ← i + 1
18:         j_0 ← j
19:     else if T[i] == P[j] or P[j] == ? then
20:         i ← i + 1
21:         j ← j + 1
22:     else
23:         i ← i_0
24:         j ← j_0
25:         i_0 ← i_0 + 1
26: while j ⩽ m and P[j] == ⋆ do
27:     j ← j + 1
28: return j > m
```

U algoritmu 9 varijablu i, odnosno j, koristimo za iteriranje po tekstu, odnosno uzorku. Najprije moramo provjeriti podudara li se početni blok uzorka s odgovarajućim podnizom teksta (linije 7-11). Nakon toga slijedi glavni dio algoritma koji implicitno određuje koordinate rješenja redom. Razlikujemo tri slučaja. Ako naiđemo na znak $\star$ u uzorku (linije 13-18), tada postavimo kursore za povratak i_0, j_0 koji služe za određivanje broja znakova

koje će zamijeniti trenutni znak $\star$. Dodatno, ako je znak $\star$ zadnji znak u uzorku, tada se očito uzorak i tekst podudaraju jer će taj $\star$ zamijeniti sve znakove teksta do kraja (linija 16). Drugi slučaj se javlja ako se znakovi uzorka i teksta podudaraju. Tada jednostavno idemo na idući znak u tekstu i uzorku (linije 19-21). Konačno, treći slučaj nastupa prilikom nepodudaranja znakova $T[i]$ i $P[j]$. To zapravo znači da smo napravili pogrešnu pretpostavku o broju znakova koje zamjenjuje trenutni znak $\star$ te stoga tu brojku povećavamo za jedan (linija 25). Osim toga, moramo iznova provjeriti podudara li se konkretni dio zamjenskog bloka s pripadnim podnizom teksta, pa stoga ažuriramo kursore (linije 23-24). Na kraju samo ostaje provjeriti da nakon što smo iscrpili sve znakove teksta više ne postoji konkretnih znakova u uzorku (linije 26-28).

Iako se algoritam 9 ima prosječnu vremensku složenost bolju od dosad navedenih algoritama za problem podudaranja znakovnih nizova sa zamjenskim znakovima, u najgorem slučaju složenost mu je kvadratna. Njegova prednost je također da, osim ulaznih podataka, koristi konstantno memorije.

Propozicija 4.3.2. *Vremenska složenost algoritma 9 u najgorem slučaju iznosi* $O(n^2)$.

Dokaz. Pogledajmo rad algoritma za dane ulazne podatke $T = \mathtt{a}^{n-1}\mathtt{b}$ i $P = \star\mathtt{a}^{n-1}$. Tekst i uzorak se očito ne podudaraju, no da bi algoritam 9 to ustvrdio, on će redom isprobavati da znak $\star$ zamjenjuje $0, 1, \ldots, n$ znakova teksta. Da bi se uvjerio da se konkretni dio uzorka ne podudara s dijelom teksta kojeg nije zamijenio znak $\star$, treba mu redom $n, n-1, \ldots, 0$ usporedbi, odnosno reda veličine $O(n^2)$. □

Bibliografija

[1] C. Charras, T. Lecroq, *Handbook of Exact String Matching Algorithms*, King's College Publications, London, 2004.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *Introduction to Algorithms, 3rd Edition*, Cambridge, The MIT Press, 2009.

[3] J. Handy, *Wildcard string compare (globbing)*, dostupno na http://www.codeproject.com/Articles/1088/Wildcard-string-compare-globbing (rujan 2015.).

[4] Z. Horvat, *Efficient String Matching Algorithm with Use of Wildcard Characters*, dostupno na http://www.c-sharpcorner.com/uploadfile/b81385/efficient-string-matching-algorithm-with-use-of-wildcard-characters/ (rujan 2015.).

[5] K. K. Krauss, *Matching Wildcards: An Algorithm*, dostupno na http://www.drdobbs.com/architecture-and-design/matching-wildcards-an-algorithm/210200888 (rujan 2015.).

[6] K. K. Krauss, *Matching Wildcards: An Empirical Way to Tame an Algorithm*, dostupno na http://www.drdobbs.com/architecture-and-design/matching-wildcards-an-empirical-way-to-t/240169123 (rujan 2015.).

[7] R. Manger, *Strukture podataka i algoritmi*, Element, Zagreb, 2014.

[8] R. Sedgewick, K. Wayne, *Algorithms, 4th Edition*, Addison-Wesley, Boston, 2011.

[9] P. H. Sellers, *The Theory and Computation of Evolutionary Distances: Pattern Recognition*, Journal of Algorithms 1, br. 4, New York, 1980, 359–373.

[10] S. Singer, *Uvod u složenost algoritama*, dostupno na http://web.math.pmf.unizg.hr/~singer/oaa/scans/pog_1.pdf (rujan 2015.).

[11] R. A. Wagner, M. J. Fischer, *The String-to-String Correction Problem*, J. ACM 21, br. 1, New York, 1974, 168–173.

[12] *Approximate string matching* dostupno na https://en.wikipedia.org/wiki/Approximate_string_matching (rujan 2015.).

Sažetak

U ovom radu obradili smo algoritme koji rješavaju problem podudaranja znakovnih nizova. Promatrali smo tri različita problema: problem egzaktnog podudaranja, problem aproksimativnog podudaranja te problem podudaranja sa zamjenskim znakovima. Pritom smo se više usredotočili na posljednje dvije klase problema. Za svaku od klasa naveli smo jedan ili više najznačajnijih algoritama koji rješava dotični problem.

Tako smo za problem egzaktnog podudaranja znakovnih nizova, uz naivni algoritma, analizirali i klasične algoritme kao što su Rabin-Karpov, Knuth-Morris-Prattov te Boyer-Mooreov. Za svaki od njih objasnili smo osnovnu ideju te pokazali vremensku složenost.

Za problem aproksimativnog podudaranja znakovnih nizova odabrali smo Wagner-Fischerov algoritam temeljen na ideji dinamičkog programiranja koji računa *edit*-udaljenost između dvaju stringova. Detaljno smo dokazali njegovu korektnost te vremensku složenost. Iako on sam ne rješava problem aproksimativnog podudaranja znakovnih nizova kako smo ga mi u radu definirali, vidjeli smo da se njegovom sitnom modifikacijom koju je uveo Sellers može rješiti i naš problem.

Na kraju, za problem podudaranja stringova sa zamjenskim znakovima predstavili smo tri rješenja. Prvo rješenje se zasniva na regularnim izrazima koji su zapravo puno ekspresivniji nego je nama potrebno pa imaju relativno veliku vremensku složenost. Drugo rješenje je dobiveno rekurzivnim algoritmom koji smo optimizirali koristeći dinamičko programiranje. Posljednje rješenje se zasniva na ideji *backtrackinga*, a u usporedbi s preostala dva ima najmanju prosječnu vremensku složenost.

Sve spomenute algoritme smo implementirali u programskom jeziku Java uz zaključak da se dobiveni eksperimentalni rezultati podudaraju s teoretskim.

Summary

In this thesis we studied string matching algorithms. We analyzed three different problems: exact, approximate and wildcard string matching. We focused on last two classes of algorithms. We mentioned one or more most significant representatives of each class.

For exact string matching, beside naive algorithm, we also analyzed classic algorithms such as Rabin-Karp, Knuth-Morris-Pratt and Boyer-Moore. For each of them we described main idea and determined complexity.

For approximate string matching, we chose Wagner-Fischer algorithm which is based on dynamic programming. It computes edit distance between two strings. While it does not solve problem of approximate string matching as we defined it, we saw that it can be modified in order to solve our problem, too. This modification was introduced by Sellers.

Finally, for wildcard string matching we presented three solutions. First one was based on regular expressions which are in fact more expressive than we need so they have relative high complexity. Second solution was recursive algorithm which is further optimized by dynamic programming. Last solution is based on idea of backtracking, and in comparison with another two solutions had lowest average time complexity.

We implemented all mentioned algorithms in programming language Java. Conclusions were that all experimental results coincide with theoretical results.

Životopis

Rođen sam 26. lipnja 1991. godine u Rijeci. Osnovnu školu „Drago Gervais" Brešca pohađam od 1998. do 2006. godine. Iste godine upisujem matematički smjer Gimnazije „Andrije Mohorovičić" Rijeka. Državnu maturu polažem 2010. godine nakon čega iste godine upisujem preddiplomski studij matematike na Prirodoslovno-matematičkom fakultetu u Zagrebu. Preddiplomski studij završavam 2013. godine kada sam i nagrađen Priznanjem za izniman uspjeh tijekom preddiplomskog studija. Iste godine upisujem diplomski studij računarstva i matematike na istom fakultetu. U 2015. godini dobio sam Priznanje za izniman uspjeh tijekom diplomskog studija. Tijekom studija sam bio demonstrator iz kolegija Programiranje 1 i 2.