Približno prepoznavanje uzorka (pretraživanje teksta)

Seminar iz kolegija Oblikovanje i analiza algoritama

Branimir Jungić

mentor: Saša Singer

24.1.2017.

Uvod

- Kako usporediti dva dokumenta da bismo saznali jesu li identični ili slični?
- Kako u nekom tekstu pronaći riječ za koju nismo sigurni kako se točno piše?
- Trivijalno rješenje: usporediti rečenicu po rečenicu, riječ po riječ.

Udaljenost uređivanjem (*edit distance*)

- Koliko preinaka je potrebno napraviti da bismo od jednog teksta dobili drugi?
- Dozvoljene „operacije”:
  - Zamijeni slovo nekim drugim slovom,
  - Izbriši slovo,
  - Umetni slovo.
- Primjer: kako od „pasta” dobiti „pseto”?

1. P ~~A~~ S T A — izbriši A na poziciji 2
2. P S E T A — umetni E između S i T
3. P S E T A O — zamijeni A sa O

Računanje udaljenosti uređivanjem

		0	1	2	3	4
		P	S	E	T	O
0	P	0	1	2	3	4
1	A	1	1	2	3	4
2	S	2	1	2	3	4
3	T	3	2	2	2	3
4	A	4	3	3	3	3

Računanje udaljenosti uređivanjem

- Neka je *s[0..i]* dio riječi *s* koja počinje na poziciji 0 i završava na poziciji *i*.
- Neka je $d_{i,j} = dist(s[0..i], t[0..j])$
- Kako računati $d_{i,j}$ iz tablice?
- Na koje sve načine je slovo *s[i]* moglo postati *t[j]*?
  - Zamijeni *s[i]* sa *t[j]* i pretvori *s[0..i-1]* u t*[0..j-1]*. Za ovaj korak treba $d_{i-1,j-1}$ + 1 operacija.
  - Obriši *s[i]* i pretvori *s[0..i-1]* u *t[0..j]*. Za ovaj korak treba $d_{i-1,j}$ + 1 operacija.
  - Dodaj *t[j]* i pretvori *s[0..i]* u *t[0..j-1]*. Za ovaj korak treba $d_{i,j-1}$ + 1 operacija.
  - Ako su *s[i]* i *t[j]* ista slova, potrebno je samo pretvoriti *s[0..i-1]* u *t[0..j-1]*. Za ovaj korak treba $d_{i-1,j-1}$ operacija.

Računanje udaljenosti uređivanjem

- Formula po kojoj se računa $d_{i,j}$:
- $$\min \left\{ \begin{array}{l} d_{i-1,j-1} + \left\{ \begin{array}{l} 0, \text{ ako je } p[\,i\,] = t[\,j\,] \\ 1, \text{ ako je } p[\,i\,] \mathrel{!=} t[\,j\,] \end{array} \right. \\ d_{i-1,j} + 1 \\ d_{i,j-1} + 1 \end{array} \right\}$$
- Za računanje $d_{i,j}$ su nam potrebne susjedne ćelije lijevo, gore lijevo i gore.
- Redoslijed kojim punimo tablicu je red po red počevši s lijevim gornjim kutom.

Računanje udaljenosti uređivanjem

- Za lakše računanje dodamo i red -1, u kojem računamo udaljenost između praznog stringa i dijela riječi.

		-1	0	1	2	3	4
			P	S	E	T	O
-1		0	1	2	3	4	5
0	P	1	0	1	2	3	4
1	A	2	1	1	2	3	4
2	S	3	2	1	2	3	4
3	T	4	3	2	2	2	3
4	A	5	4	3	3	3	3

Pseudo-kod algoritma

```
edit_distance(s, t) {
m = s.length
n = t.length
for i = -1 to m-1 dist[i, -1] = i+1 //inicijalizacija -1 stupca
for j = -1 to n-1 dist[-1, j] = j+1 //inicijalizacija -1 retka
for i = 0 to m-1
        for j = 0 to n-1
                if(s[i] == t[j])
                        dist[i,j] = min(dist[i-1,j-1], dist[i-1,j]+1, dist[i,j-1]+1)
                else dist[i,j] = 1 + min(dist[i-1,j-1], dist[i-1,j], dist[i,j-1])
}
```

Složenost algoritma

- *O(mn)* – dvije ugniježđene *for* petlje.
- Predprocesiranje traje *O(m+n)* – inicijalizacija -1 retka i -1 stupca

Najbolje približno podudaranje (*best approximate match*)

- U tekstu *t* tražimo najbolje približno podudaranje s uzorkom *p.*
- Tražimo podriječ *w* = *t[i..j]* takvu da je *dist(p,w)* najmanja.
- Trivijalno rješenje: izračunati udaljenost između *p* i svih podriječi od *t*.
- Ako je $m = |p|$, $n = |t|$, ima n^2 podriječi od *t*, da usporedimo svaki sa *p* nam treba $O(mn)$ vremena - ukupno $O(mn^3)$ operacija!

Najbolje približno podudaranje (*best approximate match*)

- Definiramo: $ad_{i,j} = \min\{dist(p[0..i], t[l..j]) \mid 0 \leq l \leq j+1\}$.
- Napomena: *t[l..j]* može biti i prazan string ako je $l=j+1$.
- Na koje sve načine možemo dobiti slovo *t[j]* od *p[i]*?
  - Možemo zamijeniti *p[i]* sa *t[j]*, ubaciti *t[j]*, izbrisati *p[i]* ili su možda *t[j]* i *p[i]* ista slova.
- Možemo se poslužiti formulom za računanje *dist(w,p)*.
- Dopušteno je *p* reducirati na prazan string.
- Prazan string je podriječ od bilo koje podriječi od t

  => $ad_{-1,j} = 0$, za svaki j

Najbolje približno podudaranje

- Formula po kojoj se računa $ad_{i,j}$:
- $$\min \begin{cases} ad_{i-1,j-1} + \begin{cases} 0, \text{ ako je } p[\,i\,] = t[\,j\,] \\ 1, \text{ ako je } p[\,i\,] \mathrel{!=} t[\,j\,] \end{cases} \\ ad_{i-1,j} + 1 \\ ad_{i,j-1} + 1 \end{cases}$$
- Za računanje $ad_{i,j}$ su nam potrebne susjedne ćelije lijevo, gore lijevo i gore.
- Redoslijed kojim punimo tablicu je red po red počevši s lijevim gornjim kutom – dinamičko programiranje.
- U zadnjem retku tražimo najmanju vrijednost – to je rješenje koje nam daje algoritam.

Pseudo-kod algoritma

```
n=t.length
m=p.length
for i=-1 to m-1 adist[i,-1] = i+1 //inicijalizacija -1 stupca
for j=0 to n-1 adist[-1, j] = 0;  //inicijalizacija -1 retka
for i=0 to m-1
        for j=0 to n-1 {
              if(p[i] == t[j])
                     adist[i,j] =
                            min(adist[i-1,j-1], adist[i-1,j]+1, adist[i,j-1]+1)
              else adist[i,j] = 1 +
                     min(adist[i-1,j-1], adist[i-1,j], adist[i,j-1])
        }
difference = m;
 for j=0 to n-1
        if(adist[m-1, j]<difference) difference= adist[m-1, j]
return difference
```

Složenost algoritma

- *O(mn)* – dvije ugniježđene *for* petlje.
- Predprocesiranje traje *O(m+n)* – inicijalizacija -1 retka i stupca

Primjer

```
Cauchy-Schwarz-Bunyakovsky Shvartz
         C   a   u   c   h   y   -   S   c   h   w   a   r   z   -   B   u   n   y   a   k   o   v   s   k   y
     0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
  S  1   1   1   1   1   1   1   1   0   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
  h  2   2   2   2   2   1   2   2   1   1   1   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
  v  3   3   3   3   3   2   2   3   2   2   2   2   3   3   3   3   3   3   3   3   3   3   3   2   3   3   3
  a  4   4   3   4   4   3   3   3   3   3   3   3   2   3   4   4   4   4   4   4   3   4   4   3   3   4   4
  r  5   5   4   4   5   4   4   4   4   4   4   4   3   2   3   4   5   5   5   5   4   4   5   4   4   4   5
  t  6   6   5   5   5   5   5   5   5   5   5   5   4   3   3   4   5   6   6   6   5   5   5   5   5   5   5
  z  7   7   6   6   6   6   6   6   6   6   6   6   5   4   3   4   5   6   7   7   6   6   6   6   6   6   6
position: 14
najmanja udaljenost izmedu "Shvartz" i neke podrijeci recenice
"Cauchy-Schwarz-Bunyakovsky" je 3 a ta podrijec je: "Schwarz"
```

Pretraživanje sa znakom ‘nebitno’ (?)

- U uzorku koji tražimo nalazi se znak ‘?’ koji ne pripada alfabetu koji koristimo.
- ‘?’ označava da nije bitno koji znak se nalazi na toj poziciji.
- Koristit ćemo Aho-Corasick algoritam, složenosti $O(m+n)$, gdje je m duljina uzorka bez *‘?’* znakova, n duljina teksta koji pretražujemo, a k broj pojava znaka ‘?’.

Primjer

- Uzorak p =*‘r?ss?l’* možemo razdijeliti na manje probleme: u zadanom tekstu tražimo pojave p_1 = *‘r’*, p_2 =*‘ss’* i p_3 = *‘l’*.
- Pozicije l_i od p_i u p su redom: $l_1 = 0$, $l_2 = 2$, $l_3 = 6$.
- Ako se *‘r?ss?l’* u tekstu t nalazi na poziciji 7 to znači da se *‘r’* nalazi na poziciji $7 + l_1 =7$, *‘ss’* na poziciji $7 + l_2 = 9$ te *‘l’* na poziciji $7 + l_3 = 13$.
- Ideja algoritma: postavimo brojače $c[i]$ koji uvećavamo za 1 ako se na poziciji $i + l_j$ pojavljuje p_j.
- $c[i] = k$ ako i samo ako se p_1 nalazi na $i+l_1$, p_2 na $i+l_2$, … p_k na $i+l_k$; tj. ako se $p_1?p_2?...?p_k$ nalazi na poziciji $c[i]$.

Pseudo-kod algoritma

```
m = p.length
k = 0
for i = 0 to m c[i] = 0
//pronađi sve pod-uzorke od p i spremi ih u sub
    sub[k].pattern = p[start..i-1]
    sub[k].start = start
    ++k
P = {sub[0].pattern, …, sub[k-1].pattern}
aho_corasick(P, t)      //pronađi sve pozicije od p_i iz P u t
for each match of sub[j].pattern in t starting at position i {
    c[i-sub[j].start] ++
    if c[i-sub[j].start] == k return i-sub[j].start
}
```

Složenost algoritma

```
m = p.length
k = 0
for i = 0 to m c[i] = 0
//pronađi sve pod-uzorke od p i spremi ih u sub
    sub[k].pattern = p[start..i-1]
    sub[k].start = start
    ++k
P = {sub[0].pattern, …, sub[k-1].pattern}
aho_corasick(P, t)      //pronađi sve pozicije od p_i iz P u t
for each match of sub[j].pattern in t starting at position i {
    c[i-sub[j].start] ++
    if c[i-sub[j].start] == k return i-sub[j].start
}
```

$O(m)$

$O(m + n)$

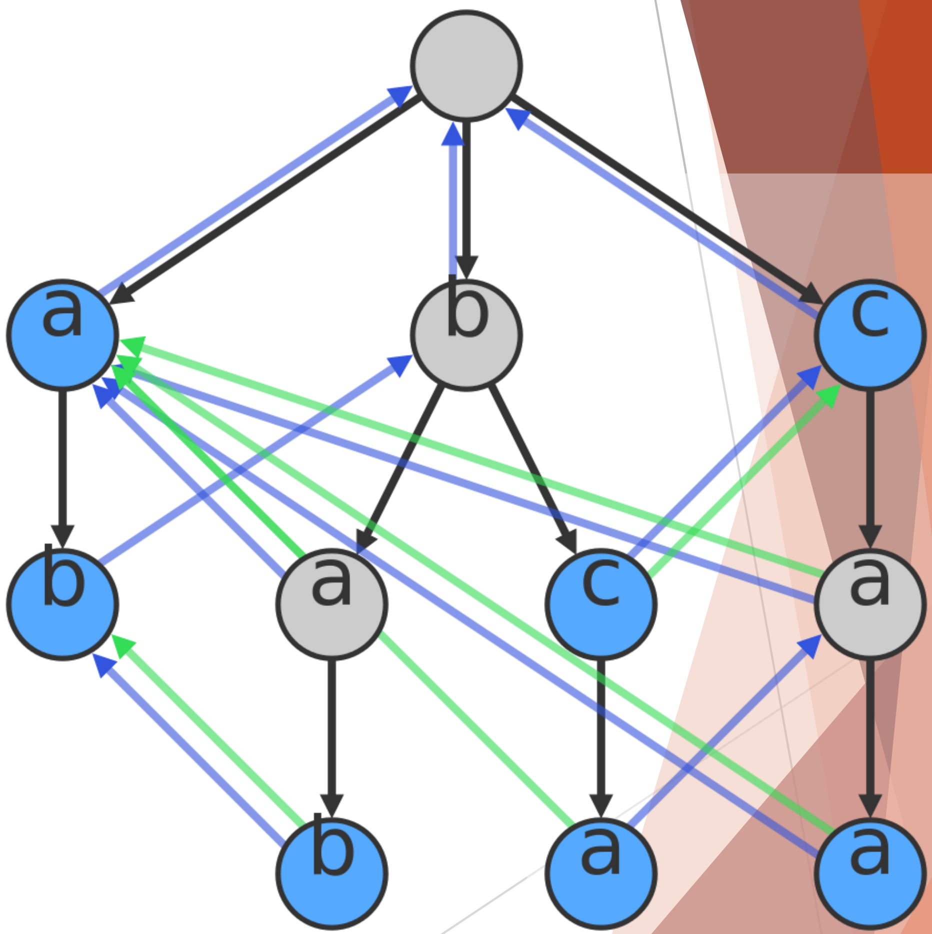
$O(kn)$

Složenost algoritma

- $O(m + kn)$.
- Ukoliko je k ograničen, to je $O(m + n)$, ako nije može biti i $O(mn)$.
- Zašto koristiti Aho-Corasick algoritam?
- Zato što je ‘usko grlo’ ovog algoritma povećavanje brojača $c[i]$, ako to napravimo paralelno složenost algoritma je $O(m + n)$.

Ukratko o Aho-Corasick algoritmu

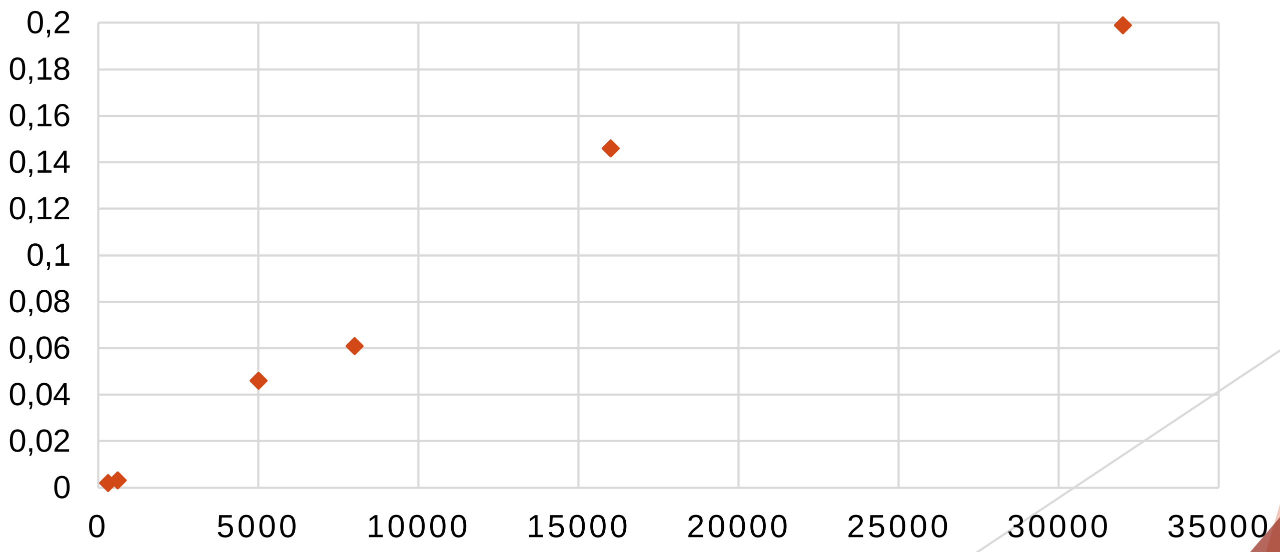
- Prvo gradi posebnu strukturu stabla sa vezama prema čvorovima djeci, sufiksima i sufiksima u rječniku.
- Potom kao automat prolazi po tekstu i stablu koje je izgradio i pronalazi nalazi li se podriječ rečenice u zadanom rječniku.
- Linearne složenosti.



rječnik: {a, ab, bab, bc, bca, c, caa}

Testiranje, k = 10

n (dužina teksta)	Aho-Corasick
300	0.002
600	0.003
5 000	0.046
8 000	0.061
16 000	0.146
32 000	0.199



Literatura

- *Algorithms*; Richard Johnsonbaugh, Marcus Schaefer; Pearson Education, 2003.
- *Aho–Corasick algorithm*, Wikipedia, https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm (pristupljeno 20.1.2017.)
- *Lecture 4: Set Matching and Aho-Corasick Algorithm*, Pekka Kilpelainen, University of Kuopio, Department of Computer Science