Textialgoritmid

Oliver-Matis Lill

April 1, 2017

Sessiooni Struktuur

- Sessioon on jaotatud kaheks blokiks
- Bloki alguses pean ma loengu, ülejäänud aeg on implementeerimiseks
- Et materjali kinnistada, soovitan vähemalt standardülesanded kodus ära lahendada

Teemad

- Räsimine
- Knuth-Morris-Pratt algoritm
- Trie
- Aho Corasick-i algoritm

Ülesanded 1, 2

- Meil on antud kaks sõne a ja b.
- Leia kui mitu korda sõne b esineb sõnes a.
- $|a|, |b| \leq 10^5$

Idee

- Oletame et me soovime kahe sõne a ja b puhul leida kas $a = b$. Naiivse algoritm teeks $O(|a|)$ operatsiooni
- Seda saab teha $O(1)$ ajaga, juhul kui kummagi sõne kohta on mingeid eelteadmisi
- Oletame et sõned koosnevad väikestest inglise tähtedest, vaatame neid kui 26-nd süsteemi arve, nt: $\text{"}bdac\text{"} = 1 \cdot 26^3 + 3 \cdot 26^2 + 0 \cdot 26 + 2$
- Probleem on et need arvud tulevad liiga suured. Vaatame hoopis nende jääke mingi piisavalt väikese arvu p-ga jagades
- Kui $|a| = |b|$ siis nende jäägid on samad, kui $|a| \neq |b|$, siis tõenäosus et jäägid on samad on väiksem kui $\frac{1}{p}$

Räsimise ülesanne

- Vaatame uuesti ülesannet 1
- Nüüd kus me saame sõnesi räside kaudu $O(1)$ ajaga võrrelda tuleb meil veel need vajalikud räsid küllalt kiiresti leida
- $\text{hash}(b)$ saab leida $O(M)$ ajaga, näiteks $\text{hash}(\text{"}eafg\text{"}) = (4 \cdot 26^3 + 0 \cdot 26^2 + 5 \cdot 26 + 6 \mod p)$
- $\text{hash}_{[1:M]}(a)$ tuleb leida $O(M)$ ajaga, aga iga järgneva $\text{hash}_{[1+i:M+i]}(a)$ saab leida $O(1)$ ajaga kasutades seost: $\text{hash}_{[1+i:M+i]}(a) = \text{hash}_{[1+i-1:M+i-1]}(a) \cdot 26 - a_{1+i-1} \cdot 26^M + a_{M+i}$

Räsimise kood

```
bool contains(string a, string b) {
    const int p = 1000000007, base = 26;
    int aHash = 0, bHash = 0, N = a.size(), M = b.size();
    for(int i=0;i<M;i++) {
        //Multiplying by 1LL is a simple way to convert to a 64-bit integer
        aHash = (1LL*aHash*base + a[i]) % p;
        bHash = (1LL*bHash*base + b[i]) % p;
    }
    for(int i=0;i+M < N; i++) {
        if(aHash == bHash)
            return true;
        aHash = (aHash*base - a[i]*modpow(base, M, p) + a[i+M]) % p;
    }
    return aHash == bHash;
}
```

Knuth-Morris-Pratt algoritm

Idee

- Arvutame iga alamsõne a positsiooni i jaoks mis on suurim väärtus d_i, nii et $a_{[i-d_i+1:i]} = b_{[1:d_i-1]}$. Näiteks kui $a = "abcbade"$ ja $b = "cbad"$, siis $d_5 = 3$ kuna $a_{[3:5]} = "cba" = b_{[1:3]}$ aga $a_{[2:5]} = "bcba" \neq "cbad" = b_{[1:4]}$
- Sisuliselt d_i vastab suurima sellise b prefiksi pikkusele, mis ühilduks sõnes a positsioonis i lõppeva samapikkuse suffiksiga
- Paneme tähele, et $d_{i+1} \leq d_i + 1$, muidu d_i ei oleks maksimum

Knuth-Morris-Pratt algoritm

Idee

- Paneme tähele, et iga i juures $b_{[1:d_{i+1}-1]}$ on $b_{[1:d_i]}$ suffiks. See tähendab et d_{i+1} on suurim selline väärtus, kus $b_{[1:d_{i+1}-1]}$ on $b_{[1:d_i]}$ suffiks ja $a_{i+1} = b_{d_{i+1}}$ (kui $d_{i+1} > 0$)
- Paneme tähele, et kui t ja u on s-i suffiksid ning $t < u$, siis t on u suffiks
- Arvutame sõne b iga positsiooni i jaoks mis on suurim väärtus p_i, nii et $b_{[1:p_i-1]} = b_{[i-p_i+1:i]}$
- Tulemusena me saame kergesti b prefiksiks olevad $b_{[1:d_i]}$ suffiksid suuremast väiksemale läbida kuni leiame sobiva
- p jadas saab suurema indeksiga elemendid väiksemate kaudu täpselt sama moodi prefiksitele hüppamise kaudu arvutada
- Ülimalt $O(N)$ väiksema prefiksi peale hüppamist kokku

Näide

a:	a	a	b	a	a	a	b	b
d:	1	2	3	4	5	2	3	0

b:	a	a	b	a	a	b	c
p:	0	1	0	1	2	3	0

Knuth-Morris-Pratt kood

```
vector<int> calculateP(string b) {
    int M = b.size();
    vector<int> p(M);
    p[0] = 0;

    for(int i=1;i<M;i++) {
        p[i] = p[i−1];
        while(p[i] > 0 && b[i] != b[p[i]])
            p[i] = p[p[i]−1];
        if(b[p[i]] == b[i])
            p[i]++;
    }
    return p;
}
```

Knuth-Morris-Pratt kood

```
vector<int> calculateD(string a, string b) {
    int N = a.size(), M = b.size();
    vector<int> d(N), p = calculateP(b);
    d[0] = 0;
    if(a[0] == b[0])
        d[0] = 1;

    for(int i=1;i<M;i++) {
        d[i] = d[i-1];
        while(d[i] > 0 && (d[i] == M || a[i] != b[d[i]]) )
            d[i] = p[d[i]-1];
        if(a[i] == b[d[i]])
            d[i]++;
    }
    return d;
}
```

Ülesanne 3

- Meil on "Sõneraamat" mis koosneb kuni 10^5 sõnest
- Meil on vaja leida kuni 10^5 teise sõne juures, kas nad esinevad antud "sõneraamatus"
- Sõneraamatu sõnede pikkuste summa on ülimalt 10^5 ja otsitavate sõnede pikkuste summa on ülimalt $2 \cdot 10^5$

Trie struktuur

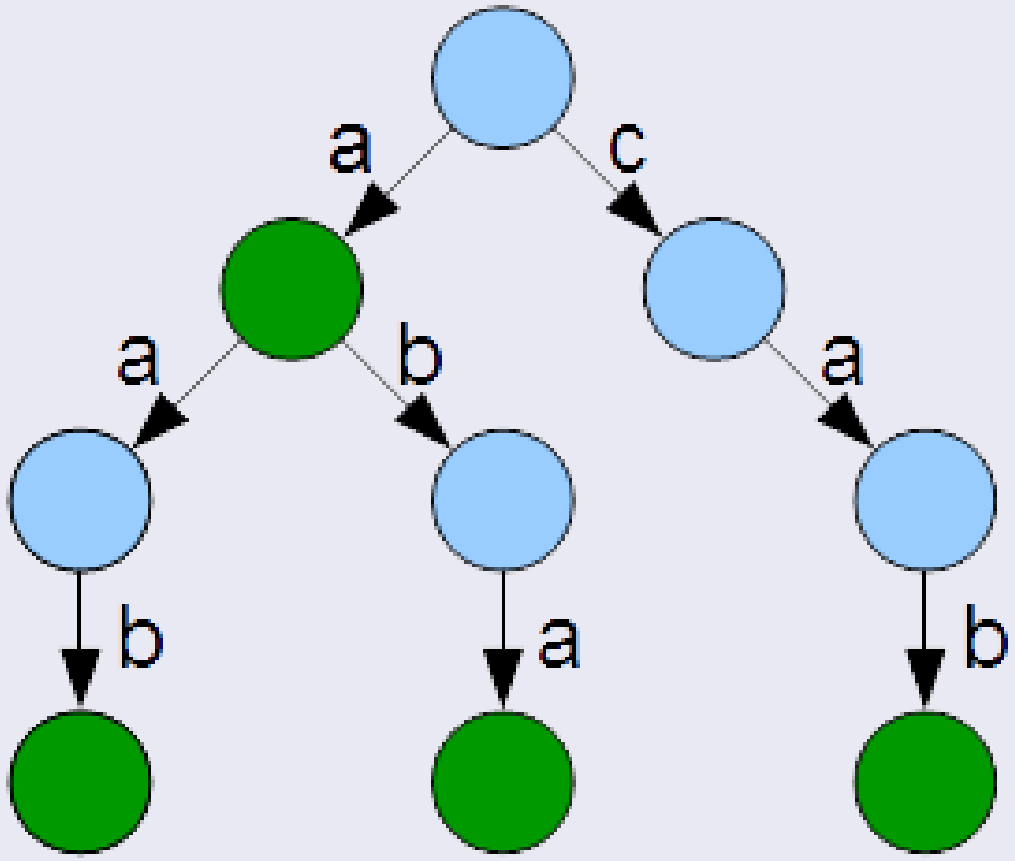
Idee

- Loome sõneraamatust puu kus tipud tähistavad mingit alamsõnet ja servad tähistavad vastava tähemärgi lisamist
- Sõneraamatu sõnede lõppudele vastavad tipud märgime lõpptippudeks (ei pruugi lehed olla)
- Sõne esinemise kontrollimiseks läbime sõneraamatu puu ja vaatame kas jõuame mingisse lõpptippu

Trie struktuur

Näide

"aab"
"aba"
"cab"
"a"



Ülesanne 4

- Siin jälle mingi sõnede kogum, ehk "sõneraamat"
- Seekord on vaja leida kui mitu korda iga sõneraamatu sõne esineb mingi suurema sõne alamsõnena

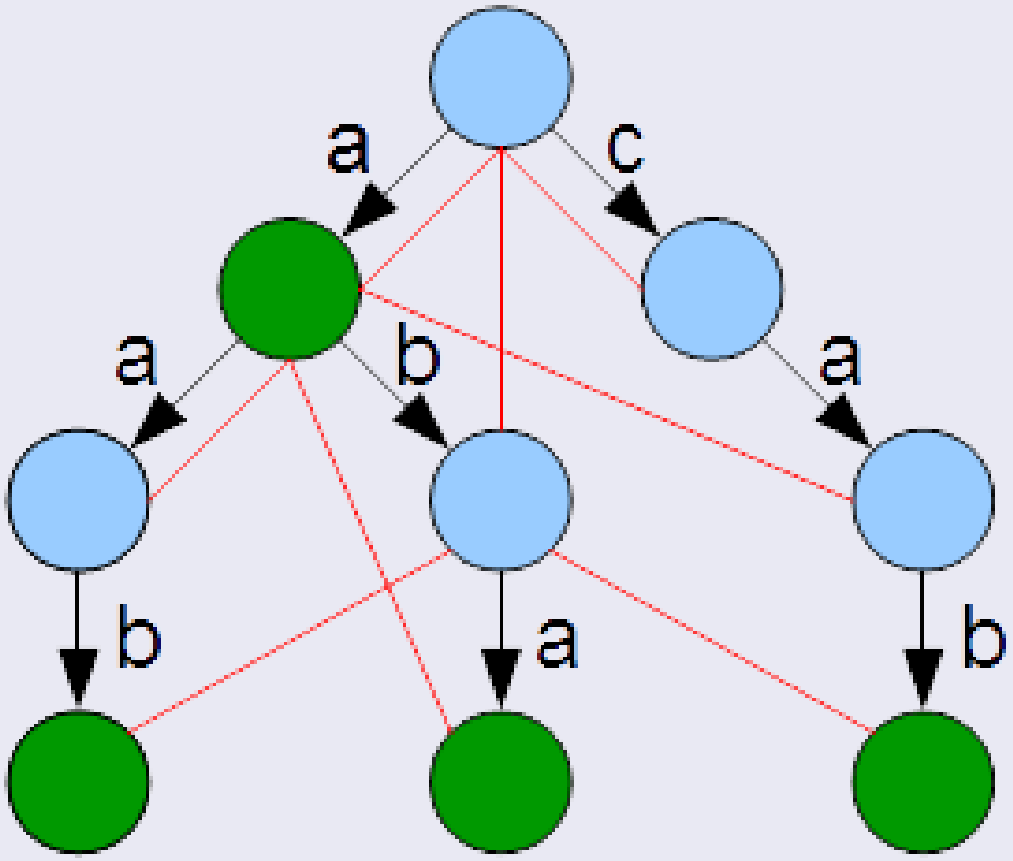
Aho-Corasick-i algoritm

Idee

- Kui sõneraamat sisaldaks ainult üht sõne, siis saaks Knuth-Morris-Pratt algoritmi kasutada
- On võimalik KMP lähenemise Trie peale laiendada
- Iga Trie tipu jaoks leiame mis on selle tipu sõne suurim suffiks, mis vastab mingile teisele Trie tipule
- Suurt sõne läbides toimub Trie peal samasugune väiksematele prefiksitele tagurdamine nagu KMP algoritmi juuresgi

Näide

"aab"
"aba"
"cab"
"a"



Aho-corasick-i kood

```
void AhoTrie::construct() {
    vector<Node*> front(1, root);
    for(int i=0;i<front.size();i++) {
        Node* cur = front[i];
        front.insert(front.end(), cur->children.begin(), cur->children.end());
        if(cur == root || cur->parent == root) {
            cur->link = root;
            continue;
        }
        cur->link = cur->parent->link;
        while(cur->link != root && !cur->link->hasEdge(cur->lastChar) )
            cur->link = cur->link->link;
        if(cur->link->hasEdge(cur->lastChar))
            cur->link = cur->link->next(cur->lastChar);
    }
}
```