Algoritmi pentru procesarea șirurilor de caractere

Dobre Gigi-Alexandru 322CD

Universitatea Politehnica București - Facultatea de Automatică și Calculatoare - Departamentul de Calculatoare - Domeniul Calculatoare și Tehnologia Informației

Rezumat: Doi algoritmi care rezolvă problema pattern searching-ului in știința calculatoarelor.

1 Introducere

Pattern searching-ul este una dintre cele mai importante și întâlnite probleme în știința calculatoarelor. Una dintre aplicațiile algoritmilor de pattern searching este atunci când dorim să căutăm un șir de caractere într-un fișier text/ browser/ bază de date. Un enunț pentru un astfel de algoritm ar putea fi:

Având șirul de caractere txt[0,...,n-1] și tiparul pat[0,...,m-1], să se scrie o funcție search(char pat[], char txt[]) care afișează toate aparițiile lui pat[] in txt[].

Problema poate fi soluționată folosind următorii algoritmi: **KMP(Knuth Morris Pratt)** și **Boyer Moore**.

1. Ideea de bază al algoritmului KMP este că de fiecare dată când detectăm o nepotrivire(după câteva potriviri), știm deja o parte dintre caracterele din șirul următor. Acest lucru este benefic deoarece nu vom mai potrivi caracterele despre care oricum știm că se vor potrivi.

2. Algoritmul Boyer Moore face preprocesarea din același motiv ca al lui KMP. Procesează tiparul si creează diferite array-uri pentru fiecare dintre cele două euristici: Bad Character Heuristic și Good Suffix Heuristic.

Testele au fost realizate manual, încercând a fi surprinsă eficiența algoritmilor aleși. Acestea vor fi diversificate si vor avea lungimi diferite pentru a putea testa și corectitudinea algoritmilor prezentați.

2 Prezentarea soluțiilor

2.1 KMP - Knuth Morris Pratt

Algoritmul KMP se bazează pe preprocesarea pattern-ului pe care dorim să îl căutăm in textul nostru, creând astfel un array auxiliar, lps, de aceeași lungime cu pattern-ul, folosit pentru a sări peste caracterele despre care știm că se potrivesc deja. LPS vine de la longest preffix suffix, ceea ce inseamnă că funcția care construiește acest array se bazează pe căutarea in sub-pattern-ul [0,...,i], unde i poate lua valori de la 0 până la m-1 și m este lungimea șirului, al celui mai lung prefix care este si sufix, reținând în lps[i], lungimea lui.

Pseudocodul algoritmului:

```
KMPSearch(P, T, output) - m e lungimea lui P; n a lui T; output pentru scriere
        computeLPSarray(P, m, lps)
        i ← 0;
        j ← 0;
        cât timp (i < n)
            dacă (P[j] == T[i])
                j ← j + 1;
                i ← i + 1;
            dacă (j == m)
                ieșire -> i-j;
                j ← lps[j-1];
            altfel dacă (i<n și P[j] != T[i])
                dacă (j != 0)
                    j ← lps[j-1];
                altfel
                    i ← i + 1;

computeLPSarray(P, m, lps)
        len ← 0;
        lps[0] ← 0;
        i ← 1;
        cât timp (i < m)
            dacă (P[i] == P[m])
                len ← len + 1;
                lps[i] ← len;
                i ← i + 1;
            altfel
                dacă (len != 0)
                    len ← lps[len-1];
                altfel
                    lps[i] ← 0;
                    i ← i + 1;
```

Analiza complexității

La fiecare iteratie a loop-ului, ori creștem i-ul, ori mutăm pattern-ul la dreapta. Fiecare poate fi facută de cel mult n ori, prin urmare loop-ul poate sa aibă cel mult 2n execuții, cu costul fiecărei operații O(1). Prin urmare, dacă presupunem că array-ul lps este deja procesat, vom avea O(n). Pentru algoritmul lps, complexitatea este O(m), ceea ce inseamnă că prin combinarea celor două, complexitatea algoritmului KMPSearch este de O(n + m).

Principalele avantaje și dezavantaje

Avantaje:

- timpul de rulare este unul foarte rapid
- algoritmul nu trebuie niciodată să se mute inapoi în textul T datorită arrayului lps, ceea ce este benefic pentru fișierele cu input mare.

Dezavantaj:

- nu funcționează bine când mărimea alfabetului crește, si din această cauză, e predispus la erori.

2.2 Boyer-Moore

Pentru algoritmul Boyer-Moore implementat in această temă, am folosit Bad Character Heuristic deoarece este mai simplu de utilizat, fiind suficient pentru implementarea noastră. Caracterul din text care nu se potrivește cu cel curent din pattern este numit Bad Character. Când nu se potrivesc, pattern-ul este shiftat până când nepotrivirea devine potrivire sau pattern-ul P a trecut de caracterul care nu s-a potrivit. În cazul în care am fi folosit si Good Suffix Heuristic, atunci am fi ales maximul dintre cele două pentru un anume pas.

Pseudocodul algoritmului:

```
BoyerMooreSearch(P, T, output) - m e lungimea lui P; n a lui T; output-scriere
        badCharHeuristic(P, m, badchar);
        s ← 0;
        cât timp (s < (n - m))
            j ← m - 1;
            cât timp (j >= 0 și P[j] == T[s + j])
                j ← j - 1;
            dacă (j == -1)
                ieșire -> s;
                dacă (s + m < n)
                    s ← s + m - badchar[T[s + m]];
                altfel
                    s ← s + 1;
```

```
            altfel
                s ← s + max(1, j - badchar [T[s + j]];

badCharHeuristic(P, m, badchar)
        pentru i = 0 până la NOofChars
            bachar[i] ← -1;
        pentru i = 0 până la m
            badchar[pat[i]] ← i;
```

Analiza complexității

Din punct de vedere al preprocesării, complexitatea funcției badCharHeuristic este de O(m + NOofChars). Complexitatea algoritmului, considerând că preprocesarea este deja făcută, este de O(n). Judecând după cazurile în care se poate afla, pe cazul general complexitatea algoritmului este O(mn) sau O(n/m) in cazul cel mai bun caz.

Principalele avantaje și dezavantaje

Avantaje:
- algoritmul preprocesează pattern-ul și nu textul
- algoritmul devine din ce în ce mai rapid atunci când crește lungimea patternului.

Dezavantaje:
- nu este bun atunci când alfabetul este unul mic

3 Evaluarea

Pentru fișierele de input am folosit următorul stil de a le redacta: toate rândurile care au maxim 80 de caractere vor constitui txt, până când este întâlnit '*', care este un delimitator între txt și pat. Tot ce urmează după constituie pattern-ul. Totodată, în cazul în care avem texte și un rând se întrerupe, iar continuarea propoziției este pe următorul rând, este indicat ca la final de rând să se pună ' ' pentru a evita concatenarea fără spațiu dintre cele 2 rânduri. Acest lucru se aplică și pentru pattern, deoarece și el poate fi constituit din mai multe rânduri. Acești algoritmi pot fi aplicați și pe alte tipuri de caractere în afară de litere. Este de dorit a se evita folosirea '*' în cadrul textului deoarece este tratat ca delimitator.

Testele au fost rulate pe o mașină virtuală pe 64-bit, cu 4 core-uri ale unui procesor Intel(R) Core(TM) i7-9750H si 6GB RAM.
Testele test0.in si test1.in nu afișează nimic deoarece ele au fost concepute astfel încât fie lipsește textul, fie pattern-ul.

Timpii de execuție pe teste(în microsecunde) sunt cei prezentați in tabelul următor:

Timp execuție	test2.in	Test3.in	Test4.in	Test5.in	Test6.in	Test7.in
Alg. KMP	3	493	11	6	47	6
Alg. BoyerMoore	2	387	11	6	65	6

Pentru a măsura timpii de execuție am folosit biblioteca chrono, pe care am introdus-o în algo.h. Timpii de execuție sunt orientativi, deoarece la fiecare rulare a testelor aceștia pot fi diferiți de rularea anterioară.
Nicio valoare neașteptată nu a apărut pe parcursul testării, iar pentru a se observa corectitudinea celor 2 algoritmi, i-am aplicat pe ambii pe aceleași date de intrare, verificându-se unul pe celălalt.

4 Concluzii

Concluziile ce pot fi trase în urma analizării celor doi algoritmi sunt următoarele:

- timpul de execuție ai celor doi algoritmi, pe același set de date, nu par a fi foarte diferiți, prin urmare oricare dintre aceștia sunt o alegere bună atunci când vine vorba de pattern searching.

- acești algoritmi au numeroase aplicații în practică, precum motoarele web de căutare, care se bazează pe ceea ce este introdus de om, reușind într-un timp foarte scurt să întoarcă o mulțime de completări relevante lucrului pe care dorim să îl găsim.

- o altă aplicație poate fi cea care prezintă situația unui laborator ce se ocupă cu ADN-ul uman, deoarece în astfel de situații este nevoie de o căutare cât mai rapidă prin secvențe largi de ADN, moment în care intervin acești algoritmi de căutare.

- totodată, unele programe de detectare ale plagiarismului folosesc KMP și Boyer Moore, împărțind ambele documente in token-uri, aplicându-se algoritmii pentru a determina dacă o lucrare este copiată sau originală.

- ultimul exemplu de aplicație este cel al unui sistem bazat pe un cod; de fiecare dată când introducem un cod într-o astfel de verificare, cel introdus este comparat cu cel stocat, iar dacă acesta este greșit, o alarmă este pornită sau un mesaj de eroare este afișat. Acest exemplu se bazează mai mult pe o potrivire de 1:1.

References

1. https://www.geeksforgeeks.org/boyer-moore-algorithm-for-pattern-searching/
2. https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
3. https://www.educba.com/kmp-algorithm/
4. https://www.cs.ubc.ca/ hoos/cpsc445/Handouts/kmp.pdf
5. https://www.inf.hs-flensburg.de/lang/algorithmen/pattern/bmen.htm
6. Introduction to Algorithms, 3rd Edition by Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest