Temă

- Analiza Algoritmilor -

Cosmin-Viorel Lovin
Grupa - 325CD
- Facultatea de Automatică și Calculatoare -
- Universitatea Politehnică București -
adresa de email : cosmiinreus@gmail.com

17 decembrie 2020

Rezumat. Mi-am propus ca în această temă să prezint și să analizez doi algoritmi pentru procesarea șirurilor de caractere. Algoritmii pe care eu i-am ales sunt: Boyer-Moore și Aho-Corasick, iar de-a lungul acestui proiect voi prezenta atât modalitatea de implementare a acestor algoritmi, cât și rezultatul unor teste care să evidențieze comportamentul acestora(și aici fac referire la comportamentul fiecărui algoritm atunci când întampinăm un caz favorabil, respectiv mai puțin favorabil). De asemenea, voi prezenta și o comparație între Knuth-Morris-Pratt și Aho-Corasick, și voi decide care algoritm este mai bun și în ce cazuri.

Keywords. Boyer-Moore - Aho-Corasick - Knuth-Morris-Pratt

Cuprins

1 Introducere

1.1 Descrierea problemei rezolvate

Algoritmii pentru procesarea șirurilor de caractere se referă la acei algoritmi prin care încercăm într-un mod cât mai eficient posibil să găsim un anumit cuvânt, o propoziție, o serie de caractere, un număr, sau chiar și o secventă de text, într-un alt text, mult mai mare.

1.2 Aplicatii pentru problema aleasa

Acești algoritmi au numeroase aplicații practice printre care se numără: editoarele de text, bibliotecile digitale, motoarele de căutare, sisteme de detectare a intrușilor, bioinformatica, detectarea plagiarismului, detectarea unor semnături digital sau text mining.

1.3 Specificarea solutiilor alese

Pentru rezolvarea acestor probleme, vom prezența mai mulți algoritmi prin intermediul cărora putem obține poziția în text unde se întâlnește cuvântul ori cuvintele căutate de noi. Algoritmul Boyer-Moore este un algoritm foarte important, reprezentând un reper principal în căutarea subsirurilor de caractere, în timp de algoritmul Aho-Corasick, este un algoritm cu ajutorul căruia putem caută în timp liniar, mai multe cuvinte într-un text, el fiind folosit și pentru implementarea comenzii din Unix, "fgrep".

1.4 Criterii de evaluare ale algoritmilor

Pentru a testa acești algoritmi, se va realiza un set de teste, al căror dimensiune să fie din ce în ce mai mare, progresiv, pentru a putea observă răspunsul în timp al algoritmilor și eficiența acestora. Se va testa atât cazul cel mai favorabil cât și cazul cel mai nefavorabil, alături de căutări multiple de subșiruri într-un text dat.

2 Prezentarea solutiilor

2.1 Boyer-Moore

Algoritmul Boyer-Moore este un algoritm de procesare a șirurilor de caractere, fiind considerat unul dintre cele mai eficiente algoritme de procesare si un reper de baza pentru ceea ce inseamna string searching. El a fost creat de către Robert S. Boyer și J Strother Moore în 1977. O versiune simplificată a acestui algoritm este folosită si implementată de

editorele de text pentru comenzile de căutare sau înlocuire a unor cuvinte în text. Algoritmul potrivește șirul căutat(pattern) cu o secvență din text începând de la coada șirului spre începutul acestuia. Există două modalități de implementare a acestui algoritm, "Bad Character heuristic" și "Good Suffix heuristic".

2.1.1 Bad Character heuristic

"Bad Character heuristic" presupune crearea unui vector (numit "bad match table") care va avea dimensiunea lungimii pattern-ului, și va conține pozițiile în pattern la care trebuie să sărim în cazul în care pattern-ul nu se mai potrivește cu secvența de text pe care o analizăm. Algoritmul începe să compare de la coadă la început caracterele, iar în cazul în care două caractere nu coincid, se shifteaza la dreapta cu valoarea corespunzătoare fiecărui caracter din bad match table. Dacă caracterul căutat nu se află în bad match table, atunci se shifteaza peste caracterul care nu coincide.

2.1.2 Good Suffix heuristic

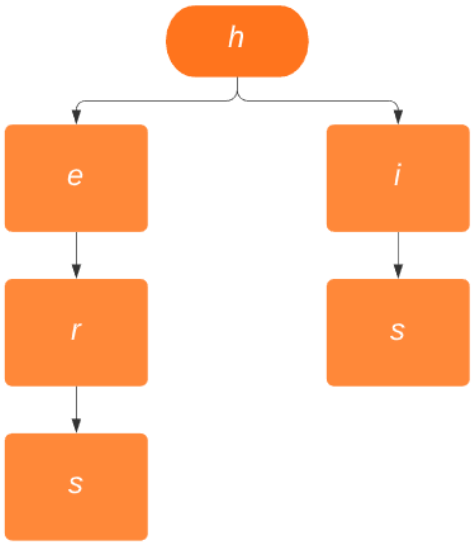
"Good Suffix heuristic" presupune crearea a doi vectori, unul de shift, care reține distanța cu care trebuie să shiftăm la dreapta pattern-ul pentru a corespunde cu o secvență sau o parte din secvența de text inițial, și un vector ce reține pozițiile marginilor (considerăm o margine acele secvențe de caractere care pot fi atat prefixe cat și sufixe). Sunt trei cazuri pe care le putem întalni:
Cazul 1:
Pattern-ul nu corespunde, însa o secvența din pattern corespunde cu o secvența din textul inițial, și atunci putem shifta pattern-ul astfel încat cele doua secvențe sa coincida.
Cazul 2:
Doar prefixul pattern-ului corespunde, și atunci îl shiftăm la dreapta.
Cazul 3:
Nu corespunde nimic, și atunci sărim pur și simplu peste acea secvență din text.

2.2 Knuth-Morris-Pratt

Algoritmul Knuth-Morris-Pratt este un algoritm de procesare a șirurilor de caractere, și a fost creat de către James H. Morris, Donald Knuth și Vaughan Pratt în jurul anului 1970. Ideea care sta la baza acestui algoritm este că se foloseste de comparatiile deja facute pentru a sari peste eventuale comparatii inutile. Folosind această informație, se crează un vector care are lungimea pattern-ului, unde se stochează poziția în pattern a celui mai bun prefix.

2.3 Aho-Corasick

Algoritmul Aho-Corasick este, de asemenea, un algoritm de procesare a șirurilor de caractere. El a fost inventat de către Alfred V. Aho și Margaret J. Corasick. Acest algoritm a avut o contribuție fundamentala în crearea comenzii fgrep din Unix. Poate fi asemănat cu un dicționar care caută nu doar unul, ci mai multe cuvinte într-un text și returnează pozițiile acestora. Ceea ce sta la baza algoritmului, este crearea unui trie, care retine pe fiecare nod cate un caracter din vectorul de cuvinte astfel încât, coborând pe noduri, până ajungem la frunze, să putem obține toate combinațiile de cuvinte. Un astfel de arbore ar putea arata așa, spre exemplu, pentru un vector de cuvinte de genul: strings[] = "his", "hers", "her":



2.4 Analiza complexitatii

Complexitatea algoritmului Boyer-Moore de căutare a substrigurilor, folosind "Bad Character heuristic", în worst-case este de $(O(n * m))$, unde n este lungimea textului, iar m este lungimea pattern-ului. Worst-case-ul apare atunci când textul și pattern-ul sunt formate din același caracter. Această complexitate reiese din adunarea complexitatilor celor două funcții, cea de creare a bad match table $(O(m))$, și cea de căutare a subsirului în text, $(O(n*m))$. În best-case, lucrurile se schimbă. Best-case apare atunci când niciun caracter din pattern nu se întâlnește în șirul dat, astfel că complexitatea obținută în acest caz va fi $(O(n/m))$. Complexitatea folosind "Good Suffix heuristic", în worst-case, ca și mai sus este de $(O(n * m))$, iar în best-case, este de $(O(n/m))$.

Complexitatea algoritmului Aho-Corasick de căutare a substringurilor este, la fel că și la Knuth-Morris-Pratt una liniară. Pentru Knuth-Morris-Pratt, complexitatea adunând complexitatea pentru crearea vectorului care este de $(O(m))$ unde m este lungimea pattern-ului, cu complexitatea funcției de căutare, $(O(n))$, este de $(O(n+m))$. Pe de alta parte, complexitatea pentru algoritmul Aho-Corasick, o vom obține din adunarea complexitatilor funcțiilor de generare a trie-ului $O(p)$ și de căutare propriu-zisă$(O(n+z))$ rezultând o complexitate de $(O(n+p+z))$, unde z reprezintă numărul total de apariții ale tuturor cuvintelor în text.

2.5 Avantaje si dezavantaje

Algoritmul Boyer-Moore este un algoritm eficient ce folosește preprocesarea pentru a putea calcula cu exactitate numărul de poziții cu care poate sări în text atunci când o pereche de caractere nu coincide. Drept urmare, algoritmul obține rezultate mai bune atunci când pattern-ul este mai lung. Algoritmul Knuth-Morris-Prat utilizează de asemenea preprocesarea, însă pentru a refolosi caracterele din pattern care au fost deja potrivite în text. La fel că și Knuth-Morris-Pratt, algoritmul Aho-Corasick permite să se sară peste verificări repetate astfel încât este asigurat faptul că, caracterele ce au fost deja verificate, prin mutarea la dreapta a pattern-ului, nu mai trebuie reverificate. Avantajul acestui algoritm apare atunci când dorim să căutăm mai multe pattern-uri în același text, deoarece implementarea acestuia parcurge textul o singură dată, iar preprocesarea se realizează pentru toate pattern-urile în același timp. Dacă ar fi să se caute un vector de cuvinte într-un text dat, complexitatea acestui program folosind algoritmul Knuth-Morris-Prat ar fi $(O(n*k+p))$ unde k este numărul de cuvinte din vectorul de cuvinte, iar p reprezintă suma lungimilor tuturor cuvintelor din acest vector, pentru că textul ar trebui parcurs pentru fiecare pattern iar preprocesarea ar avea loc, de asemenea, pentru fiecare pattern în parte. Cu toate acestea, algoritmul Aho-Corasick necesită destul de mult spațiu pentru partea de preprocesare, deci pentru căutarea unui singur cuvânt în text, sunt mai avantajoși ceilalți doi algoritmi.

3 Evaluare

3.1 Modalitatea de construire a testelor

Setul de teste a fost construit astfel: pentru algoritmii Boyer-Moore(good suffix heuristic) și Knuth-Morris-Pratt, pe prima linie se află numărul de caractere din textul dat, pe a două linie, numărul de pattern-uri, urmând că în continuare fișierul să conțină de la 1 cuvânt până la 10000 de cuvinte. Pentru algoritmii Aho-Corasick și Boyer-Moore(bad

character heuristic), testele sunt construite similar, cu excepția faptului că pe prima linie nu se mai află numărul de caractere din textul dat.
Toate testele au fost construite de mâna astfel încât să cuprindă toate cazurile posibile atât best-case cât și worst-case. Sunt douăzeci și două de teste (+1 destul de mare) comune. Acestea sunt alcătuite progresiv, de la texte mici, la texte mari. Textele au fost construite pornind la un text de baza care a fost repetat de un număr de ori pentru a obține unul mai mare și pentru a putea verifică corectitudinea output-ului mai ușor. Pentru testele 10, 13, 14, 16, pattern-urile au fost alese de mâna, în timp ce pentru testele 11, 12, 15, 10000words, pattern-urile reprezintă "Top 100/200/1000/10000 most common English words". Pentru generarea testelor 18, 19, 20 s-a folosit un repetor de text, pentru a repetă doar caracterul "a".

3.2 Specificațiile sistemului de calcul

Sistemul de calcul este unul pe 64 de biți, iar procesorul este Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz. În plus, sistemul de calcul are un SSD de 128 de GB dintre care 20 GB liberi și o memorie RAM de 8GB, dintre care 5.7 GB liberi.

3.3 Performanța algoritmilor

Timpii au fost exprimați în microsecunde.
Pentru setul de teste 0 - 7, 13, 16, 17, 21. Testele sunt normale, cu un singur pattern de căutat în textul inițial.

Input	BM : bad character	BM : good suffix	Knuth-Morris-Pratt	Aho-Corasick
0	500	500	500	133693
1	500	500	500	133693
2	500	500	500	134642
3	500	500	500	134672
4	500	500	500	139672
5	500	500	500	136805
6	500	500	500	136645
7	500	500	500	134654
13	2991	3990	3989	139723
16	10970	7976	5985	154278
17	357050	312177	311180	589761
21	997	996	1995	423176

Pentru setul de teste 10, 11, 12, 14, 15, 10000 words. Testele conțin un set de pattern-uri care vor fi căutate in text.

Input	BM : bad character	BM : good suffix	Knuth-Morris-Pratt	Aho-Corasick
10	1000	1000	1000	135603
11	5978	4984	6021	140623
12	736032	856719	929504	228059
14	28922	29919	30928	158612
15	301225	279253	290180	237398
10000words	-	-	-	26889723

Pentru setul de teste 8, 9, 18, 19, 20. Acestea testează best-case și worst-case.

Input	BM : bad character	BM : good suffix	Knuth-Morris-Pratt	Aho-Corasick
8	1000	1032	997	139628
9	1000	997	1028	137613
18	2992	994	2992	146607
19	2585679	2470845	2518251	2605020
20	12244227	10534313	9487182	9271742

3.4 Analiza performanțelor algoritmilor

Din primul tabel al performanțelor se poate observa slăbiciunea algoritmului Aho-Corasick la input-uri ce conțin un singur pattern, cu text relativ scurt. Cu toate acestea, în cel de-al doilea tabel putem observă că algoritmul Aho-Corasick obține rezultate remarcabil mai bune decât ceilalți algoritmi pentru input-urile 12 și 15. Motivul pentru care algoritmul nu are rezultatele așteptate pentru input-urile 10, 11 și 14, este numărul de operații care se realizează în preprocesare. Începând cu 200 de pattern-uri, algoritmul Aho-Corasick începe să fie mai eficient decât celelalte. Totuși, adevărata putere a acestui algoritm se vede pentru input-ul 10000words. Acesta conține 100013 pattern-uri, iar algoritmul Aho-Corasick a îndeplinit acest task în aproximativ 27-28 secunde, în timp ce pentru ceilalți algoritmi, procesul a durat aproximativ 40 de minute. De asemenea, analizând rezultatele testelor 13 și 21 pentru algoritmii Boyer-Moore, observăm într-adevăr o diferența considerabilă de timp, datorită lungimii pattern-ului. Ambele teste conțin același text, însă au pattern-uri diferite. Testul 21, care conține un pattern mai lung decât testul 13 a fost rezolvat mai rapid, deci algoritmul funcționează mai bine pentru pattern-uri mai lungi. În ultimul tabel se pot observă rezultatele algoritmilor în worst-case și best-case.

4 Concluzii

În urmă analizării algoritmilor putem trage următoarele concluzii. Algoritmul Boyer-Moore și-a dovedit eficientă cu adevărat atunci când a venit vorba despre input-uri ce conțin un singur pattern de căutat în text. În plus, eficientă acestuia a crescut odată cu lungimea pattern-ului, iar algoritmul a continuat să obțină rezultate bune și pentru mai multe pattern-uri, fiind chiar mai eficient decât algoritmul Aho-Corasick atunci când numărul de cuvinte căutate era relativ mic(sub 200). Totuși, pentru input-uri mari, Aho-Corasick și-a dovedit cu adevărat puterea. Așadar, pentru input-uri cu un număr de pattern-uri cuprins între 1 și 200, aș folosi algoritmul Boyer-Moore, in timp ce pentru un număr mare de input-uri, aș apela la algoritmul Aho-Corasick.

5 Referinte

https://www.geeksforgeeks.org/boyer-moore-algorithm-good-suffix-heuristic/
https://stackoverflow.com/questions/27428605/constructing-a-good-suffix-table-understanding-an-example
https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/
https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/
https://en.wikipedia.org/wiki/Boyer-Moore
https://en.wikipedia.org/wiki/Knuth-Morris-Pratt
https://en.wikipedia.org/wiki/Aho-Corasick