Algoritmi eficienți pentru sortarea și căutarea stringurilor

Dima Oana-Teodora, IS 406

Decembrie 2022

Jon L. Bentley, Robert Sedgewick. Fast Algorithms for Sorting and Searching Strings.
SODA, Ianuarie 1997

1 Context

Autorii acestui paper își doresc să aducă în atenția cititorilor alternative eficiente ale QuickSort-ului clasic pentru sortarea unor dicționare de șiruri de caractere. Pornind de la implementarea obișnuită folosind BST, vor dezvolta pe parcursul lucrării un set de teoreme și ipoteze privind eficacitatea abstractizării algoritmului către utilizarea TST. Acestea vor fi demonstrate și susținute printr-o serie de experimente efectuate pe inputuri diverse și în medii cu nivele de performanță diferite.

2 QuickSort clasic: analiză, algoritmi și teoreme

QuickSort este un algoritm recursiv care folosește metoda Divide et Impera pentru sortarea unui vector, pe baza alegerii unui pivot și permutarea elementelor în funcție de acesta. Modul de funcționare poate fi redus la structura unui BST.

Complexitatea de timp a algoritmului poate fi influențată de modul în care este ales pivotul: drept primul/ultimul element, random sau median-of-three.

De-a lungul timpului, mai mulți cercetători au exprimat soluții diverse pentru tratarea cazului în care există mai multe elemente egale cu pivotul. **Hoare** a propus o metodă de partiționare binară: plasarea valorilor mai mici la stânga și pe cele mai mari la dreapta, dar elementele egale pot apărea în orice parte.

Dijkstra a încercat găsirea unei soluții bazate pe un algoritm ternar în timp liniar, popularizând astfel "Problema steagului național olandez". Soluția se bazează pe coloritul stindardului olandez: elementele mai mici sunt colorate în roșu, cele egale sunt albe, iar cele mai mari sunt albastre. Teoretic, eficiența de timp este datorată faptului că algoritmul evaluează o singură dată fiecare element, însă în practică, codul pentru a-l implementa are o constantă semnificativ mai mare ca algoritmul binar al lui Hoare.

Ulterior, vor mai încerca și alți cercetători o **abordare ternară** asupra problemei: **Bentley și McIlroy** vor prezenta o partiționare ternară bazată pe un loop invariant contraintuitiv, iar **Wegner** va descrie și el scheme asemănătoare, dar cu un cod mult mai complex ca soluția propusă de Bentley și McIlroy.

De asemenea, vor fi emise și demonstrate **4 teoreme** privind modurile de partiționare utilizate de QuickSort și costurile de căutare așteptate. Concluziile care reies pe baza acestora sunt evidente. Un **arbore perfect echilibrat**, cât și alegerea unei **mediane** vor reduce semnificativ costurile de căutare ale algoritmului la $\lg n$ comparații, respectiv la $cn \lg n + O(n)$ (timp ideal, nu worst case; cn - numărul de comparații efectuate pentru calcularea medianei).

3 Multikey QuickSort: analiză, algoritmi și teoreme

Multikey Quicksort sortează un set de n vectori cu k componente fiecare. Ca și în cazul unui Quicksort clasic, acesta partiționează seturile de date în seturi mai mici și mai mari decât un pivot, însă asemănător Radix Sort, trece la următorul element odată ce elementul curent este egal cu elementul ales.

Hoare nu a studiat și propus algoritmi eficienți doar pentru QuickSort-ul clasic, cât va enunța și modele de algoritmi pentru Multikey Quicksort. O implementare elegantă a acestora se bazează pe proprietățile TST. Acesta sortează o secvență s de lungime n, care are o anumită secvență de componente identice. Pivotul poate fi ales în moduri diferite, asemenea QuickSort-ului, de la calcularea unei mediane până la alegerea sa aleatorie. Fiecare nod din arbore va conține: un pivot, pointeri către descendenții mici și mari și un pointer către un descendent egal care reprezintă setul de vectori cu valori egale cu pivotul.

Mulți cercetători au studiat modalitățile de implementare și reducere a costurilor de sortare pentru n, k vectori folosind TST. În urma studiilor efectuate, au fost emise alte **4 teoreme**. Demonstrațiile acestora includ și ipotezele teoremelor anterioare, având și concluziile asemănătoare cu acestea. Utilizarea unui **TST perfect balansat** în Multikey QuickSort, cât și calcularea unei **mediane**, crește nivelul de performanță al algoritmului prin reducerea costului de căutare, necesitând cel mult $[\lg n] + k$ (optim), respectiv cel mult $cn(\lg n + k)$ comparații(cn - numărul de comparații efectuate pentru calcularea medianei).

4 Aplicații

Autorii acestei lucrări au ales să-și conducă experimentele și aplicațiile practice folosind limbajul C, datorită rapidității și eficienței care-l caracterizează.

Utilizarea teoremelor, cât și implementarea algoritmilor folosind structuri arborescente pot fi foarte utile atunci când avem de sortat stringuri. În următoarele secțiuni se va face referire la 2 tipuri de seturi de date:

- **S1** - un dicționar englez cu 72 275 cuvinte și 696 436 caractere.
- **S2** - un set de chei unice (aproximativ 86 000 extrase din fiecare fișier), reprezentând numere de card și având o lungime medie de 22,5 de caractere (DIMACS library call numbers).

Dar și un set de mașini pe care se vor rula testele:

- **M1** - MIPS R4400(150MHz), MIPS R4000(100MHz), Pentium(90MHz) și 486DX(33MHz).

4.1 Sortarea unui vector de stringuri fără TST

Multikey QuickSort este utilizat pentru ordonarea lexicografică a unui vector x de n pointeri la șiruri de caractere, prin **descompunerea ternară recursivă**, aplicată caracter cu caracter și fără utilizarea TST. Astfel, modalitatea de funcționare a acestui program va fi bazată pe QuickSort-ul clasic, cu alegerea unui pivot aleatoriu.

Sortarea va fi implementată sub forma unui **simple qsort** folosind 2 funcții auxiliare: una care mută secvențe de elemente egale din pozițiile lor temporare de la sfârșitul vectorului înapoi la locul lor corespunzător din mijloc, iar cealaltă va transforma un "întreg în caracter" de pe poziția curentă. După fiecare partiționare sortăm recursiv secvențele de elemente mai mici, mai mari și pe cele egale până la sfârșitul șirurilor.

Au fost desfășurate 2 experimente distincte prin care se evaluează **numărul de secunde** necesare sortării pe M1 și utilizând algoritmii: system qsort, simple qsort, tuned sort și highly tuned radix sort a lui Bostic și McIlroy(cel mai rapid cunoscut).

Primul experiment presupune aplicarea tuturor algoritmilor menționați pentru sortarea S1. În urma analizei rezultatelor, **simple qsort** este mai rapid ca system qsort, dar nu mai rapid ca tuned sort și radix sort, însă tuned sort e doar cu câteva procente mai lent ca radix sort.

Al doilea experiment presupune aplicarea tuturor algoritmilor menționați pentru sortarea S2. În urma analizei rezultatelor, **tuned sort** s-ar putea dovedi mai rapid decât radix sort și în alte contexte, acesta fiind cu 20% mai rapid pe mașinile MIPS.

4.2 Sortarea unui vector de stringuri folosind TST

Multikey QuickSort este folosit pentru ordonarea lexicografică a unui vector x de n pointeri la șiruri de caractere **prin utilizarea TST**. Această alegere va duce la implementarea unui model de tipul **string symbol table**, fiind o alternativa viabilă la hash tables.

Al treilea experiment prezentat presupune testarea performanței pe S1 prin raportarea la **numărul mediu de ramuri** parcurse în toate căutările de succes. Concluziile trase expun diferențele dintre inputuri și dau naștere unei noi teoreme: numărul de noduri dintr-un TST este constant și independent de ordinea în care nodurile sunt inserate.

Într-un **TST perfect balansat** vor fi cele mai puține ramuri formate, iar timpul este la jumătate față de worst-case-ul așteptat. În timp ce folosirea unui tournament tree utilizează cu 80% mai multe comparații ca în arborii echilibrați, iar în random tree cu 50% mai multe comparații.

Al patrulea experiment are ca scop diferențierea dintre TST și symbol tables structures pe S1 și S2. Pentru măsurarea performanței ne vom raporta la **numărul de secunde** pentru căutări cu succes și eșuate. Rezultatele arată că **TST sunt mai rapizi pentru căutările eșuate**, deoarece pot descoperi nepotriviri după câteva examinări ale caracterelor, în timp ce hashing-ul are nevoie de procesarea întregului șir. Pentru chei lungi avantajul este și mai semnificativ(pe S2 a luat mai puțin cu 1/5 din timpul de hash).

Deși din punct de vedere al eficienței de timp, TST și-a demonstrat eficacitatea în fața symbol tables, însă din punctul de vedere al **memoriei utilizate** pierde considerabil în fața acestora. Pentru S1, TST folosește 4.573 MB, iar hashing-ul 1.564 MB. Există și o reprezentarea alternativă care salvează spațiu pentru TST: când un subarbore conține un singur string, stocăm un singur pointer la șirul respectiv (la fiecare nod se stochează 3 biți care spun dacă descendenții săi indică noduri sau șiruri), rezultând astfel un cod mai lent și mai complex, dar reducând spațiul utilizat la o valoare apropiată de cel folosit de hashing.

4.3 Partial-match searching folosind TST

Problema partial-match searching presupune căutarea într-un dicționar(aici vector n,k) a unui șir s, după un anumit pattern (ex: ".o.o.o"), unde punctul este denumit ca fiind un caracter "don't care".

Cercetătorul **Rivest** a propus un model pentru partial-matching, care va fi implementat și testat folosind TST. Dacă o literă este specificată, se evaluează pe rând câte o singură ramură dată, iar pentru un caracter "don't care" se caută recursiv pe toate ramurile.

La fel ca la probleme anterioare, și aici au fost desfășurate mai multe experimente pentru evaluarea performanței. Au fost folosite datele din S1, dar și un set de date aleatoriu. Rezultatele obținute sunt surprinzătoare: nespecificarea pozițiilor la începutul cuvântului este mult mai costisitoare ca nespecificarea pozițiilor la sfârșitul acestuia(dacă începe sau se termină cu un caracter "don't care"). Explicația ar fi că, la începutul cuvintelor numărul de ramuri este mai mare decât la sfârșitul lor.

4.4 Nearest neighbor searching folosind TST

Problema "nearest neighbor searching" presupune găsirea cuvintelor dintr-un dicționar(aici vector n,k) care sunt apropiate într-o **distanță Hamming** de cuvântul dat (exemplu bazat pe S1: pentru d=2 și cuvântul "soda" ar fi "code", "coma" și alte 117 cuvinte). Pentru studiul performanței și această problemă va fi implementată folosind TST, unde vor fi date ca input un arbore, un string și o distanță.

Concluziile rezultate în urma efectuării unei serii de experimente pe această implementare sunt influențate de valorile distanțelor (de la 0 la 4). Aceste experimente au arătat că, căutarea **vecinilor apropiați** este relativ eficientă, iar căutarea pentru vecinii îndepărtați devine mai scumpă.

5 Concluzii

Ideile care stau la baza Multikey QuickSort implementat folosind TST au ca rezultate **algoritmi eficienți**. Acest tip de arbori sunt preferați deoarece combină 2 lumi: low overhead al BST (în ceea ce privește spațiul și timpul de rulare) și eficiența character-based a căutărilor.

De asemenea, este de menționat și faptul că aplicabilitatea acestora în viața reală este subestimată de cele mai multe ori. Ei sunt utilizați în reprezentarea dicționarelor engleze în sistemul OCR(Optical Character Recognition), deoarece **sunt mai rapizi decât hashing-ul**, permițând gestionarea unor seturi de zeci de mii de caractere Unicode Standard și pot răspunde eficient la multe tipuri de interogări care ar necesita timp liniar într-un tabel hash. Astfel, TST pot concura cu cel mai bun symbol table cunoscut.

6 Terminologie și abrevieri utilizate

- BST - binary search tree - arbore în care fiecare nod are maxim 2 descendenți. Toate nodurile din subarborele stâng sunt mai mici ca rădăcina și toate nodurile din subarborele drept sunt mai mari ca rădăcina, iar fiecare subarbore este la rândul său BST.
- TST - ternary search tree - arbore în care fiecare nod are maxim 3 descendenți fiecare fiind ordonat sub forma unui BST. Un nod este reprezentat ca un subset de vectori având o valoare de partiționare și trei indicatori: unul către elemente mai mici, unul către cele mai mari și unul către elemente egale.
- median-of-three (a unui vector) - elementul din mijloc din sortarea în ordine crescătoare a primului, ultimului și elementului din mijloc a unui vector.
- pivot (al unui vector) - elementul de referință atunci când se fac niște permutări cu scopul obținerii unui array/subarray sortat.
- arbore perfect echilibrat (balanced) - arbori în care valoarea fiecărui nod este mediana setului de elemente dintr-o dimensiunea dată.
- Radix sort - algoritm de sortare a numerelor pe baza grupării individuale a cifrelor aflate pe aceleași poziții.
- performance tuning - tehnici utilizate pentru îmbunătățirea eficienței cum ar fi: sortarea subvectorilor mici cu insertion sort, calcularea medianelor, specifice limbajului utilizat (C- înlocuirea indicilor vectorilor cu pointeri), salvarea diferenței dintre elementele comparate, reordonarea testelor, utilizarea regiștrilor. Pot crește cu până la 10% eficiența unui program.
- tournament tree - TST în care inputul este sortat începând prin inserarea de pe nodul din mijloc, apoi pe cel stâng și drept.

- string symbol table - hash table de mărimea n folosit pentru reprezentarea a n stringuri.
- distanța Hamming - numărul de poziții ale căror simboluri corespunzătoare sunt diferite între două șiruri de lungime egală