Proiect la Analiza Algoritmilor

Năstase Maria, 321CA

Facultatea de Automatică și Calculatoare
Universitatea Politehnica din București
mnastase20@gmail.com

1 Introducere

1.1 Descrierea problemei rezolvate

Procesarea de texte este una dintre cele mai importante funcții ale unui calculator. Computerele sunt folosite pentru a edita, descărca și a trimite documente. În fiecare zi sunt generate date noi, fapt care a rezultat în dezvoltarea unor sisteme care pot arhiva multe informații prezentate în format text. În prezent, internetul a devenit o colecție vastă de informații care includ texte distribuite și stocate în diverse formate (HTML, PDF, Email,doc, etc.), dar și principalul mod de partajare a fișierelor. În consecință, deoarece textele stau la baza comunicării în modul online și deoarece au utilități în absolut orice domeniu, s-a dorit evoluția procesării de texte pentru a răspunde nevoilor utilizatorilor.

În clasica problemă de căutare a unui "pattern", se dă un text de lungimea n și un șir de caractere având o lungime m mai mică sau egală cu n. Se caută patternul pentru a se determina dacă acesta este un subșir al textului. În cazul în care pattern-ul a fost identificat, se poate cere indicele de început, sau numărul de apariții ale sale.

În mod similar, potrivirea aproximativă a stringurilor stă la baza multor funcții de editare a textului. În cadrul acestei probleme, se dă un șir "pattern" și un șir text și se cere identificarea unui subșir, care dintre toate subșirurile textului, are cel mai mic edit distance față de pattern. Numărul de operații necesare pentru a converti șirul în pattern determină gradul de potrivire al acestora. Operațiile diferă în funcție de caz, dar cele mai uzuale sunt inserția, ștergerea si substituția.

1.2 Aplicație practică a problemei

Atât problema de căutare a unui pattern, cât și potrivirea aproximativă a șirurilor sunt aplicabile în tehnica de autocompletare, dar și în software-uri de verificare a scrisului ortografic. Un bun exemplu este Microsoft Word, care oferă mai multe opțiuni pentru a corecta greșeli de editare sau a corecta anumite erori de exprimare conform unei gramatici definite. De asemenea, aceste probleme stau la baza metodelor de detectare ale plagiatului, folosind algoritmi pentru a compara două texte și a gasi asemănări pentru a determina dacă lucrarea a

fost copiată. Căutarea unui pattern are o utilitate în motoarele de căutare și are un rol important în respingerea spamului (în Email prin căutarea unor cuvinte speciale). O altă aplicație reprezintă potrivirea secvențelor de nucleotide pentru datele ADN-ului.

1.3 Specificarea soluțiilor alese

Pentru a rezolva problema potrivirii aproximative a șirurilor, am ales algoritmul Edit distance, mai specific distanța Levenshtein și distanța Longest Common Subsequence (LCS). Distanța Levenshtein constă în măsurarea diferenței dintre două șiruri, fiind exprimată în numărul minim de editări ale caracterelor pentru a ajunge la același șir. Operațiile specifice acesteia sunt inserția, ștergerea si substituția. Acest algoritm și-a obținut numele de la matematicianul Vladimir Levenshtein, care a luat în considerare acest tip de distanță în 1965. Pe de altă parte, LCS are funcția de a găsi cea mai lungă subsecvență comună tuturor secvențelor (subsecvențele nu ocupă în mod particular poziții consecutive). LCS acceptă doar 2 operații, anume inserția si ștergera. Având în vedere că LCS are un număr mai mic de operații față de Levenshtein, rezultatele celor doi algoritmi vor fi diferite, iar distanța Levenshtein va fi întotdeauna mai mică. Pentru ambele tipuri de distanțe o să se utilizeze matrici $n \times m$ (n și m sunt lungimile celor doua siruri) care au rolul de a calcula costul convertirii unui șir în celălalt, iar valoarea finală a distanței se va afla pe ultima linie, în ultima coloană. În cazul în care costul, sau mai bine zis distanța dintre cele două stringuri este mică, atunci acestea se potrivesc parțial.

În scopul rezolvării problemei de potrivire a patternului, am ales să folosesc structura de date de tip arbore numita Trie, a cărui funcționalitate constă în reținerea stringurilor și în căutarea rapidă a unui pattern. Plănuiesc să împart problema în două subprobleme: căutarea unui întreg cuvânt sau a unui prefix, care va fi rezolvată folosind un Trie standard, și problema căutării unui sufix, soluționată prin intermediul unui Suffix Trie. Operațiile specifice ale acestei structuri de date sunt găsirea, inserarea și ștergerea. Numele Trie provine de la cuvântul “retrieval”, care face referire la utilitatea sa in recuperarea datelor. Suffix Trie constă în reținerea fiecărui sufix al unui șir de caractere. Fiecare nod exceptând rădăcina (e asociată cu un șir nul) al unui Trie conține un singur caracter și toți fiii unui nod au un prefix comun asociat cu părintele. Într-un standard Trie, parcurgerea de la orice fiu al rădăcinii către orice frunză rezultă într-un string care a fost reținut de către arbore, pe când un Suffix Trie, așa cum indică numele, reține toate sufixele unui șir de caractere.

1.4 Criteriile de evaluare pentru soluția propusă

Pentru a evalua algoritmii aleși, mi-am propus să generez 35 de teste. În cazul arborilor de tip Trie, voi începe testarea cu căutarea unui pattern într-un cuvânt. Apoi, voi adăuga la input din ce în ce mai multe cuvinte ce vor forma un text pentru a putea analiza comportamentul și eficiența în timp a acestei structuri

de date în procesarea șirurilor de caractere. Deoarece atât Suffix Trie, cât și Standard Trie acceptă căutarea unui prefix, sau a întregului cuvânt, voi testa rapiditatea acestora pe același set de teste. Apoi, mă voi concentra pe testarea căutării unui sufix într-un text prin intermediul unui Suffix Tree și voi alege pattern-uri de diferite lungimi.

Pentru algoritmii de Edit distance, am decis să generez teste separate față de Trie și voi analiza outputul celor două tipuri de distanțe alese și complexitatea acestora. Deoarece LCS nu permite substituția, distanța rezultată se va putea exprima întotdeauna în funcție de distanța Levenshtein întrucât costul operației de substituție este egal cu suma costurilor unei inserții și a unei ștergeri. În analiza celor două distanțe, voi pune în evidență timpul obținut în parcurgerea algoritmilor prin compararea unor șiruri, fie de aceeași mărime, fie de mărimi diferite și voi testa cum se comportă algoritmii la diferențe mai mari între textele date în input.

2 Prezentarea soluțiilor

2.1 Descrierea modului de funcționare al algoritmilor aleși

Edit Distance Pentru a afla valoarea distanței Edit dintre două șiruri se vor folosi distanța LCS și distanța Levenshtein, care vor genera rezultate diferite, deoarece, spre deosebire de distanța Levenshtein, LCS nu admite operația de substituție.

Se dorește compararea a două șiruri de lungime n și m prin metoda Levenshtein. Se construiește o matrice $(n+1) \times (m+1)$, unde prima coloană și primul rând vor reprezenta costul de construire a fiecărui șir. Astfel, pe primul rând vor fi puse valorile 0, 1, ... m și prima coloană va conține costurile 0, 1, ... n. Apoi, se trece la completarea matricii cu restul costurilor. Când algoritmul ajunge la poziția $[i][j]$, primele "i" caractere din primul șir, vor fi comparate cu primele caractere "j" din cel de-al doilea șir și se va introduce valoarea calculată a distanței Levenshtein dintre cele două subșiruri în matrice. Pentru a calcula în mod eficient distanța dintre cele două subșiruri, se va calcula costul fiecărei operații admisă (substituție, ștergere și inserție), iar apoi acestea vor fi comparate și se va alege costul minim drept rezultat. Odată ce matricea a fost parcursă și completată în întregime, atunci distanța Levenshtein se va găsi în colțul din dreapta jos. Deoarece o denumire alternativă a distanței Edit este distanța Levenshtein, se consideră că valoarea acestora este egală.

Distanța LCS este diferită de Levenshtein prin faptul că focusul acesteia este de a căuta cea mai lungă subsecvență comună a două șiruri. Algoritmul folosește o matrice $(n+1) \times (m+1)$, dar primul rând și prima coloană vor conține doar valoarea 0, deoarece se consideră că nu s-a găsit încă o subsecvență comună. În cadrul algoritmului, se compară fiecare caracter al primului șir cu caracterele celui de-al doilea șir. În cazul în care două caractere din cele două șiruri

sunt egale, mărimea celei mai lungi subsecvențe comune va crește cu 1 față de pasul anterior. Astfel, poziția $[i][j]$ din matrice va conține lungimea maximă a subsecvenței comune dintre primele "i" caractere din primul șir și primele "j" caractere din cel de-al doilea șir. Odată ce matricea a fost completată, valoarea lungimii celei mai lungi subsecvențe se va regăsi în colțul din dreapta jos. Deoarece costul unei substituții este dublul unui cost de ștergere sau inserție, valoarea distanței Edit va fi calculată folosind formula $n + m - 2distanta_{LCS}$. Pentru a afla cea mai lungă subsecvență, matricea va fi parcursă din colțul din dreapta jos către colțul din stânga sus. Dacă caracterul situat la indicele "i" în primul șir corespunde cu cacterul de la indicele "j" din cel de-al doilea șir, atunci caracterul va fi adăugat la subsecvență. Deoarece parcurgerea a avut loc de la dreapta la stânga, subsecvența rezultată va fi inversată. În consecință, prin inversarea subsecvenței, se va ajunge la răspunsul dorit.

Trie În implementarea aleasă de mine, căutarea unui pattern se face în mod case sensitive, atât pentru Suffix Trie, cât și pentru Standard Trie. Fiecare nod al unui trie conține un vector de noduri fii, cu o lungime egală cu numărul de caractere din tabelul ASCII, adică 256.

Un Standard Trie inserează fiecare caracter dintr-un cuvânt în următorul fel: un caracter devine nod fiu pentru predecesorul său. Dacă se dorește introducerea unui text, acesta este convertit într-o listă de cuvinte pentru a putea permite căutarea cuvintelor de la mijlocul textului. Altfel, arborele ar avea o înălțime egală cu lungimea textului.

Dacă un cuvânt este inserat, pentru fiecare caracter care nu are un nod fiu existent corespondent în arbore, unul nou va fi creat și adăugat la poziția caracterului din tabelul ASCII în vectorul de noduri fii ai nodului menit predecesorului său. Apoi, când se va ajunge la finalul cuvântului, ultimul caracter va fi marcat drept terminator. Funcția de căutare începe de la rădăcină și verifică dacă nodul curent are un nod fiu la poziția caracterului căutat în tabelul ASCII. În cazul în care nodul fiu a fost găsit, se coboară în ierarhie, iar nodul curent devine nodul fiu și procesul se repetă până este identificat întregul pattern. În caz contrar, se presupune că pattern-ul ori nu există în text, ori nu e un prefix sau cuvânt în text.

Ștergerea unui cuvânt se face prin parcurgerea recursivă a arborelui, căutând nodul care corespunde finalului cuvântului. În consecință, funcția se comportă similar cu cea de căutare până când se găsește întregul cuvânt. Dacă pattern-ul căutat este doar un cuvânt în textul dat, fiecare caracter care nu face parte din alt subșir va fi șters odată cu întoarcerea recursivă. Însă, dacă acesta face parte din alte cuvinte, în sensul că nodul corespondent ultimului caracter are fii, atunci ștergerea cuvântului va consta în eliminarea atribuirii de final de cuvânt.

Un Suffix Trie preia fiecare subșir al unui text și îl inserează în arbore. Deși ocupă mai mult spațiu decât un Standard Trie, acest tip de arbore permite atât

căutarea unui prefix, cât și căutarea unui sufix. De asemenea, fiecare nod va avea o listă de indici care memorează orice apariție în text.

Inserarea unui subșir se face recursiv; pentru fiecare caracter, un nod fiu este creat și adăugat în lista de fii a nodului curent la indicele caracterului în tabelul ASCII. Indicele caracterului din text va fi adăugat la lista de indici ai nodului.

Căutarea unui pattern se face recursiv, căutând pe rând fiecare caracter. Dacă nu se găsește un nod în lista de fii la indicele caracterului în tabelul ASCII, atunci pattern-ul nu există în text. În caz contrar, dacă se găsește nodul fiu corespondent ultimului caracter din subșir, atunci se întoarce lista de indici al acestuia. Pentru a obține indicii tuturor aparițiilor pattern-ului în text, se va scădea lungimea pattern-ului de la fiecare index din lista returnată.

2.2 Analiza complexității soluțiilor

Edit Distance Atât în implementarea algoritmului distanței Levenshtein, cât și în implementarea algoritmului distanței LCS, se folosește programarea dinamică, deoarece complexitatea temporală este O($n \times m$), unde n este lungimea primului șir, iar m reprezintă dimensiunea celui de-al doilea șir. Complexitatea spațială constă în O($n \times m$), deoarece, pentru a calcula valoarea distanței Edit, se folosește o matrice cu dimensiunea $(n + 1) \times (m + 1)$.

Trie Complexitatea temporală pentru a construi un Standard Trie este O($n \times m$), unde m este egal cu numărul de cuvinte care vor fi inserate în arbore, iar n este egal cu lungimea medie a unui cuvânt. De asemenea, operațiile de inserare, căutare și ștergere au o complexitate egală cu O(n), unde n este egal cu lungimea șirului de caractere. În cazul unui Suffix Trie, complexitatea inserării unui șir sau a construcției arborelui este O(n^2). Funcția de căutare are o complexitate egală cu O(l + n), unde l reprezintă lungimea pattern-ului, iar n corespunde numărului de apariții ale pattern-ului.

2.3 Prezentarea principalelor avantaje și dezavantaje pentru soluțiile luate în considerare

Edit Distance Avantajele folosirii algoritmilor Edit Distance constau în simplitatea algoritmului și eficiența în cazul șirurilor scurte. Însă, nu se ia în considerare înțelesul semantic, iar implementarea este costisitoare la nivel computațional, deoarece se folosește o matrice. De asemenea, Edit Distance este un algoritm ineficient pentru șirurile foarte lungi.

Trie Un avantaj al utilizării unui Trie este printarea alfabetică a tuturor cuvintelor dintr-un text. Căutarea unui prefix și autocompletarea sunt realizate în mod eficient. De asemenea, inserarea și ștergerea șirurilor de lungime N se face

în O(N), generând astfel un timp mai bun decât un ABC. Dezavantajul principal al unui Trie este faptul că necesită multă memorie pentru a stoca șiruri de caractere. Fiecare nod conține un vector de noduri fii cu o dimensiune egală cu numărul de caractere în tabelul ASCII.

3 Evaluare

3.1 Descrierea modalității de construire a setului de teste folosite pentru validare

Edit Distance Testele au fost construite pentru a trata diferite cazuri, precum compararea a două cuvinte total diferite (fara caractere comune), compararea a acelorași cuvinte. De asemenea, testele conțin atât șiruri scurte, cât și șiruri lungi. În scopul testării timpului de execuție al programului, am pus accent pe șirurile de dimensiuni foarte mari. Cuvintele foarte lungi au fost alese pe baza unor articole care discută și ordonează cele mai lungi cuvinte. Verificarea corectitudinii testelor a fost realizată prin intermediul unor calculatoare online pentru Edit distance.

În total, au fost create 17 teste. Cel mai scurt cuvânt are o lungime de 3 caractere, iar cel mai lung conține 189819 de caractere.

Trie Testele au fost construite pentru a compara comportamentul programului la șiruri de lungime mică și texte cu dimensiuni mari. Unul dintre teste conține un text format din cuvinte generate prin intermediul unui generator online de șiruri, dar nu s-a dovedit mult prea folositor pentru că nu existau multe pattern-uri care să se repete. Textele de dimensiuni mari sunt preluate din articole de Wikipedia. Verificarea corectitudinii testelor a fost realizată prin intermediul următoarelor comenzi Linux:

head -2 test_number.in | tail -1 | grep ”pattern” pentru operația de căutare a unui Suffix Trie, deoarece pune în evidență pattern-ul în text dacă apare cel puțin o dată

head -2 test_number.in | tail -1 | grep - bo ”pattern” pentru operația de căutare a unui Standard Trie, deoarece returnează numărul de apariții ale unui pattern, case sensitive

În total, au fost create 18 teste. Cel mai scurt text are o lungime de 39 caractere, iar cel mai lung conține 1248 de caractere.

3.2 Specificații ale sistemului de calcul pe care au fost rulate testele

Toate testele au fost rulate mai întâi pe stația personală cu următoarele specificații:

- Procesor: Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz
- Memorie RAM: 16 GB
- Memorie libera: 7.9 GB
- Memorie folosita de sistem: 7.8 GB
- Tip de sistem: x64
- Sistem de operare: Windows 10 Pro

Am întâmpinat niște probleme la testarea implementării Trie-urilor, deoarece o parte din testele mele conțineau șiruri foarte lungi (de 8000 - 10000 caractere), iar Suffix Trie-ul devenea mult prea mare. Din această cauză, IntelliJ genera eroarea "Out of Memory: Java Heap Space", ceea ce înseamnă că programul rămânea fără memorie. Pentru a rezolva această problemă, am redus șirurile la 5500 de caractere și programul a rulat fără erori.

Apoi, pentru a testa Makefile-ul, am folosit o mașină virtuală cu următoarele specificații:

- Procesor: Intel® Core™ i7-10700 CPU @ 2.90GHz × 2
- Memorie RAM: 4 GB
- Memorie libera: 2.7 GB
- Memorie folosita de sistem: 0.8 GB
- Tip de sistem: x64
- Sistem de operare: Ubuntu 20.04.1 LTS

Când am reluat testarea pe mașina virtuală, problema a reapărut, așa că am ales să reduc lungimea șirurilor la 1200 de caractere, iar programul a funcționat cum trebuie.

3.3 Ilustrarea rezultatelor evaluării soluțiilor pe setul de teste

Pentru a obține timpii de rulare ai testelor am creat doua variabile: startTime și endTime, care măsoară când începe și când se termină execuția unui algoritm. Pentru a obține un timestamp, se folosește comanda Java System.nanoTime(), care măsoară timpul în nanosecunde. Pentru a calcula timpul exact de execuție, se face diferența dintre timpul obținut la începutul rulării algoritmului și timpul de la final. Însă, din cauza benchmarking-ului, timpul de execuție variază odată cu fiecare rulare. Astfel, pentru a calcula o valoare cât mai exactă a timpului de execuție, am rulat fiecare test de 10 ori și am aflat media per test.

Edit Distance În tabelul următor voi ilustra timpii de rulare ai testelor de Edit Distance în microsecunde.

Nr. Test	**Nr. caractere șir1**	**Nr. caractere șir2**	**Levenshtein**	**LCS**
1	6	7	475	683
2	4	4	491	673
3	3	3	476	576
4	7	11	494	807
5	10	13	563	831
6	7	8	497	692
7	10	8	566	790
8	34	10	569	836
9	11	13	495	834
10	36	12	519	838
11	30	28	524	854
12	25	17	579	818
13	58	29	623	825
14	45	44	788	986
15	28	14	510	833
16	30	10	505	827
17	189832	183	944286	188821

Trie În tabelul următor voi ilustra timpii de rulare ai testelor de Trie în milisecunde.

Nr. Test	Nr. caractere text	Nr. pattern-uri	Standard Trie	Suffix Trie
18	38	6	8229	4537
19	159	15	8701	20529
20	318	18	9310	82540
21	1083	21	11262	581039
22	1082	24	10192	596791
23	1184	17	15137	69763
24	1183	25	10822	699629
25	1190	27	10526	718162
26	1164	24	9081	688264
27	1146	25	11016	665413
28	1190	24	10433	707700
29	1207	29	11088	744859
30	1174	25	11957	757949
31	1245	29	11382	783648
32	1239	30	11959	772167
33	1233	35	9428	800259
34	1247	33	11310	807840
35	1246	21	16544	800155

3.4 Prezentare a valorilor obținute pe teste

Edit Distance După cum se poate observa în tabelul pentru Edit Distance, timpii de rulare sunt relativ mici (cel mai mare timp de rulare este de 944 ms). În majoritatea testelor, algoritmul pentru distanța Levenshtein generează un timp de rulare mai mic în comparație cu algoritmul pentru distanța LCS. Acest fapt se datorează căutării suplimentare a celei mai lungi subsecvențe comune din cadrul algoritmului LCS. Dacă nu se realizează această căutare, timpii rezultați ar avea valori mult mai apropiate.

Singura excepție este testul 17, unde timpul de execuție pentru algoritmul Levenshtein este de 9 ori mai mare decât cel pentru algoritmul LCS. Levenshtein calculează costurile a 3 operații și apoi le compară pe toate 3 pentru a obține costul minim. De asemenea, dacă două caractere diferă, costul va crește. Astfel, în cazul unei comparații de șiruri foarte mari cu multe caractere diferite, se va ajunge la numere foarte mari. Spre deosebire de Levenshtein, LCS admite doar două operații și face o comparație între două costuri pentru a identifica maximul. În consecință, LCS nu va lucra cu numere foarte mari în cazul în care se dorește comparația a două șiruri de lungimi mari cu puține caractere în comun. La testul 17, valoarea distanței Levenshtein este egală cu 189649, iar distanța LCS are o valoare egală cu 170.

De aici se poate trage concluzia că algoritmul LCS este mai eficient în compararea șirurilor de lungimi mari.

Trie Din tabelul pentru Trie se poate observa că timpii de rulare sunt mult mai mari decât cei de la Edit distance, fiind de ordinul miilor de milisecunde. În general, Suffix Trie-ul are un timp de rulare mult mai mare, deoarece construirea acestuia necesită mai mult timp și spațiu, fiind inserate toate sufixele unui șir. În cazul cel mai defavorabil, numărul de noduri poate fi egal cu n^2, unde n e dimensiunea șirului. De asemenea, căutarea unui sufix în Standard Trie se va termina relativ repede pentru că acesta nu va fi găsit, pe când căutarea într-un Suffix Trie va parcurge un număr de noduri egal cu lungimea pattern-ului pentru a-l declara drept găsit.

Deși un Standard Trie nu poate căuta și identifica un sufix, va fi mai eficient decât un Suffix Trie atât din punct de vedere al memoriei, cât și al timpului de execuție.

4 Concluzii

În concluzie, după o analiză detaliată a algoritmilor de Trie și de Edit Distance, se poate spune că sunt algoritmi buni, fiecare având limitări specifice și timpi de execuție mari în cazul în care sunt procesate texte de lungimi foarte mari. Aceștia sunt folosiți în informatică pentru căutarea pattern-urilor într-un șir sau pentru potrivirea aproximativă a string-urilor. De aceea, la fiecare trebuie analizată aplicabilitatea și eficiența algoritmului în funcție de problema care trebuie rezolvată și trebuie ales algoritmul potrivit

Pentru a cuantifica disimilitudinea a două șiruri, algoritmul Levenshtein se dovedește a fi mai benefic, deoarece distanța Edit reprezintă numărul minim de editări pentru a transforma un șir în celălalt. Aceasta diferă de LCS, care calculează dimensiunea celui mai lung subșir comun și determină valoarea distanței Edit prin intermediul unei formule, având în vedere că nu admite substituția. Însă, valorile pentru Edit Distance calculate folosind ambii algoritmi studiați diferă, iar în majoritatea cazurilor, Levenshtein va genera un cost al editărilor mai mic decât LCS. Astfel, deoarece se cere valoarea minimă, reiese că algoritmul Levenshtein este mai exact, deși poate necesita un timp de rulare mai mare pentru șirurile foarte lungi.

În mod normal, aș opta pentru algoritmul Levenshtein, dar în cazul în care se cere valoarea distanței Edit dintre două texte foarte lungi, aș utiliza algoritmul LCS. Acesta va obține un timp mai bun și un rezultat care nu va diferi prea mult față de valoarea calculată folosind Levenshtein.

În mod similar, Suffix Trie este mai ineficient din punct de vedere al spațiului și timpului față de Standard Trie. Totuși, Suffix Trie poate îndeplini mai multe funcții decât un Standard Trie, deoarece permite și căutarea sufixelor, pe lângă căutarea cuvintelor și a prefixelor într-un text.

În consecință, aș opta pentru un Suffix Trie dacă încerc să caut pattern-uri fără a ști dacă sunt sufixe sau nu. Pe de altă parte, voi apela la un Standard Trie în cazul pentru a căuta doar prefixe și cuvinte întregi.

Bibliografie

1. Michael T. Goodrich, Roberto Tamasia, Michael H. Goldwasser: Data Structures & Algorithms in Java
2. Marios Hadjieleftheriou, Divesh Srivastava: Approximate String Processing
3. William J. Masek, Michael S. Paterson: A faster algorithm computing string edit distances, Journal of Computer and System Sciences 20, 18–31, 1980.