Algoritmi de sortare în informatică

Sebastian Mărginean
Departamentul de Informatică
Facultatea de Matematică și Informatică
Universitatea de Vest Timișoara, România
Email: sebastian.marginean02@e-uvt.ro

Rezumat

Lucrarea se concentrează asupra problemei sortării în domeniul informaticii. Sortarea reprezintă procesul de aranjare a unui set de elemente într-o anumită ordine. Aceasta este o problemă fundamentală și esențială în dezvoltarea algoritmilor și are o aplicabilitate largă în diverse domenii, cum ar fi bazele de date, procesarea de imagini și analiza datelor. Scopul lucrării este de a explora diferite algoritmi de sortare și de a identifica cei mai eficienți în funcție de criterii precum complexitatea temporală și spațială.

Soluția propusă constă în analiza și compararea mai multor algoritmi de sortare, inclusiv bubble sort, insertion sort, selection sort, merge sort, quicksort și heapsort. Se vor evalua performanțele acestor algoritmi în funcție de seturile de date de intrare și se vor identifica avantajele și dezavantajele fiecăruia. În final, se vor trage concluzii și se vor propune direcții pentru optimizarea și adaptarea algoritmilor de sortare în funcție de cerințele specifice ale aplicațiilor.

Cuprins

1 Introducere

Motivație

Problema sortării este una dintre cele mai fundamentale probleme în domeniul informaticii și se referă la organizarea unui set de elemente într-o anumită ordine. Scopul este de a reorganiza elementele într-un mod care să respecte criteriul de ordonare specificat.[1] Problema sortării este interesantă datorită importanței sale practice și a impactului său în eficiența algoritmilor și a aplicațiilor care utilizează sortarea. Optimizarea algoritmilor de sortare poate duce la îmbunătățirea performanței sistemelor informatice și la economisirea de timp și resurse.[3]

Descriere informală a soluției

Soluția problemei se bazează pe studiul unor algoritmi fundamentali de sortare, care vor fi analizăți pe baza mai multor fișiere de intrare de dimensiuni diferite. Această lucrare dorește urmărirea comportamentului fiecărui algoritm pe seturile de date generate și analiza timpului de execuție.

Exemple simple ce ilustrează problema și soluția

Problemă: Având o listă de numere [5, 4, 3, 2, 1], sortează numerele în ordine crescătoare.
Soluție: Una dintre soluțiile posibile este de a folosi algoritmul bubble sort. Acest algoritm compară numerele adiacente și le interschimbă dacă sunt în ordinea greșită, iterând prin listă de mai multe ori până când lista este complet sortată[2]. Aplicând bubble sort la lista dată, rezultă [1, 2, 3, 4, 5], care este lista sortată în ordine crescătoare. Dezavantajele acestui algoritm de sortare sunt datorate faptului ca acesta are o complexitate de $O(n^2)$.

Exemplu complex

Problemă: Într-o aplicație de gestionare a bazelor de date, există nevoia de a sorta un volum mare de date în funcție de diferite criterii, cum ar fi numele, data sau valoarea. Această sortare trebuie să fie eficientă și să ofere rezultate rapide, având în vedere cantitatea mare de date de prelucrat.

Soluție: Pentru a evalua eficiența algoritmilor de sortare în contextul specific al aplicației de gestionare a bazelor de date, se pot realiza teste și compara performanțele diferitelor algoritmi. Se pot utiliza algoritmi precum quicksort, mergesort sau heapsort pentru a sorta seturile de date, și se poate măsura timpul de execuție și resursele utilizate de fiecare algoritm.[4] În funcție de rezultatele obținute, se poate determina care algoritm este cel mai eficient și potrivit pentru necesitățile aplicației de gestionare a bazelor de date.

Declarație de originalitate

Recunosc că am consultat diverse surse de informații și am folosit referințe relevante pentru a sprijini argumentele și afirmațiile prezentate în lucrare, iar aceste surse sunt corect menționate și citate în conformitate cu standardele academice.

Instrucțiuni de citire

Lucrarea este organizată în așa fel încât cititorul să înțeleagă în primul rând care este problema și care este soluția, după care lucrarea urmărește implementarea soluției urmată de un studiu de caz și câteva comparații.

2 Descrierea formală a problemei și soluției

Descrierea problemei

Dată o listă de elemente, problema constă în a sorta elementele într-un anumit ordine, cum ar fi crescătoare sau descrescătoare. Scopul este de a proiecta un algoritm de sortare eficient care să poată aranja elementele în ordinea dorită.

Context

Problema sortării este un subiect fundamental în informatică și matematică, cu diverse aplicații în prelucrarea datelor, recuperarea informațiilor și proiectarea algoritmilor. Este esențial să înțelegem caracteristicile și proprietățile diferitelor algoritmi de sortare pentru a alege abordarea cea mai potrivită pentru cerințele specifice.

Descrierea soluției

Soluția pentru problema de sortare constă în dezvoltarea unui algoritm care rearanjează elementele listei de intrare în ordinea dorită, pe baza anumitor criterii impuse.

Proprietăți ale algoritmului[7]

1. Corectitudine: Algoritmul de sortare ar trebui să producă ordonarea corectă a elementelor conform criteriilor specificate.
2. Analiză a complexității: Complexitatea temporală și de spațiu a algoritmului trebuie analizată pentru a evalua eficiența și

cerințele sale de resurs, pentru a identifica costul minim.

3. Stabilitate: O metodă de sortare este considerată stabilă dacă ordinea relativă a elementelor care au aceeași valoare a cheii nu se modifică în procesul de sortare.

4. Adaptabilitate: Algoritmul ar trebui să poată manipula liste de intrare de diferite dimensiuni și să își ajusteze performanța în consecință.

Demonstrația proprietăților soluției

Proprietățile algoritmului de sortare, cum ar fi corectitudinea, complexitatea, stabilitatea și adaptabilitatea, pot fi demonstrate formal folosind raționamente matematice, argumente logice și tehnici de demonstrație. De asemenea, algoritmul poate fi ilustrat folosind un exemplu complex, prezentându-se execuția sa pas cu pas și modul în care obține ordinea de sortare dorită.

3 Modelarea și implementarea problemei și soluției

Bubblesort(Algoritm de complexitate O(n^2))[8]

1. Considerăm un vector X cu n elemente

2. Parcurgem vectorul și pentru oricare două elemente învecinate care nu sunt în ordinea dorită, interschimbăm valorile

3. Repeta parcurgerea până când nu se mai face nicio interschimbare, moment în care vectorul este sortat

Heapsort(Algoritm de complexitate O($nlogn$))[12]

1. Începem prin construirea unui heap din tabloul de elemente de sortat

2. După construirea heap-ului, cel mai mare element este extras și plasat într-o poziție corespunzătoare în tabloul sortat, dupa care heap-ul este reconstruit pentru a reflecta noua configurație a elementelor rămase

3. Se repetă pașii precedenți până când toate elementele sunt extrase și plasate în ordine în tabloul sortat

Insertionsort(Algoritm de complexitate O(n^2))[9]

1. Considerăm un vector X cu n elemente

2. Dacă secvența cu indici 0, 1, ..., i-1 este ordonată, atunci putem insera elementul X[i] în această secvență astfel încât să fie ordonată secvența cu indici 0, 1, ..., i-1, i

3. Luăm pe rând fiecare element X[i] și îl inserăm în secvența din stânga sa

Mergesort(Algoritm de complexitate O($nlogn$))[10]

1. Mergesort este o metodă eficientă de sortare a elementelor unui tablou, bazată pe ideea: dacă prima jumătate a tabloului are elementele sortate și a doua jumătate are de asemenea elementele sortate, prin interclasare se va obține tabloul sortat.

2. Dacă $st <= dr$, problema este elementară, secvența fiind deja sortată

3. Dacă $st < dr$ se împarte problema în subprobleme, identificând mijlocul secvenței, m = (st + dr) / 2; Se rezolvă subproblemele, sortând secvența delimitată de st și m, respectiv secvența delimitată de m+1 și dr – apeluri recursive; Se combină soluțiile, interclasând cele două secvențe, moment în care, secvența delimitată de st și dr este sortată.

Quicksort(Algoritm de complexitate O([$nlogn$, n^2])[11])

1. Se alege un element din vector, numit pivot

2. Se ordonează elementele listei, astfel încât toate elementele din stânga pivotului să fie mai mici sau egale cu acesta, și toate elementele din dreapta pivotului să fie mai mari sau egale cu acesta

3. Se continuă recursiv cu secvența din stânga pivotului și cu cea din dreapta lui, ajungând într-un final că vectorul să fie sortat

Radixsort(Algoritm de complexitate O($d * (n + b)$))

1. Pornind de la poziţia celui mai nesemnificativ caracter, algoritmul numără de câte ori apare fiecare caracter pe poziţia respectivă, apoi împarte un vector auxiliar în secţiuni. Numărul de secţiuni este numărul de caractere diferite ce pot exista în vector, adică fiecărei secţiuni îi este asociat un caracter. Dimensiunea unei secţiuni depinde de numărul de apariţii ale caracterului asociat

2. Pune fiecare element(în vectorul auxiliar) în secţiunea corespunzătoare, apoi copiază vectorul auxiliar înapoi în vectorul ce trebuie sortat. Se obţine un vector sortat până la poziţia curentă

3. Algoritmul rece la poziţia următoare şi repetă paşii, ultima poziţie fiind cea a celui mai semnificativ caracter

Selectionsort(Algoritm de complexitate O(n^2))[13]

1. Fie un vector X cu n elemente
2. Parcurgem vectorul cu indicele i
3. Parcurgem cu indicele j elementele din dreapta lui X[i]
4. Dacă elementele X[i] și X[j] nu sunt în ordinea dorită, le interschimbăm

4 Studiu de caz / Experiment

Specificațiile sistemului de calcul pe care s-au făcut testele

- Procesor - Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz
- RAM - 8.00 GB
- Tipul sistemului - 64-bit
- Sistem de operare - Windows 10 Home

Seturi de date

Nr. el.	Bubble	Heap	Insertion	Merge	Quick	Radix	Selection
10^3	0.05 s	0.002 s	0.02 s	0.001 s	0.001 s	0.001 s	0.02 s
10^4	6.88 s	0.06 s	3.51 s	0.034 s	0.01 s	0.01 s	2.87 s
10^5	718 s	0.55 s	300 s	0.55 s	0.25 s	0.26 s	283 s
10^6	85241 s	7.60 s	33147 s	6.28 s	3.42 s	3.95 s	30149 s

Nr. el.	Bubble	Heap	Insertion	Merge	Quick	Radix	Selection
10^3	0.04 s	0.003 s	0.02 s	0.001 s	0.001 s	0.001 s	0.03 s
10^4	7.26 s	0.08 s	3.75 s	0.053 s	0.01 s	0.01 s	2.97 s
10^5	845 s	0.65 s	332 s	0.77 s	0.35 s	0.31 s	317 s
10^6	87243 s	7.75 s	36137 s	7.24 s	3.91 s	4.45 s	32145 s

Nr. el.	Bubble	Heap	Insertion	Merge	Quick	Radix	Selection
10^3	0.03 s	0.002 s	0.01 s	0.001 s	0.001 s	0.001 s	0.02 s
10^4	7.16 s	0.05 s	3.45 s	0.033 s	0.01 s	0.01 s	2.65 s
10^5	798 s	0.55 s	311 s	0.58 s	0.29 s	0.27 s	303 s
10^6	84258 s	6.92 s	32485 s	6.97 s	3.63 s	4.17 s	29754 s

De ce se structurează datele experimentale?

Datele experimentale se structurează deoarece algoritmii pot da timpi diferiți în funcție de datele pe care utilizatorul le introduce. Așadar experimentul făcut demonstrează faptul că nu există cel mai bun algoritm de sortare și că timpii de execuție al acestora depinde de numărul de elemente și ordinea în care acestea sunt amplasate în vector.

Cum se rulează experimentul?

Experimentul rulează pe 1.000, 10.000, 100.000, 1.000.000 de elemente, care sunt generate random, urmând că mai apoi să se execute algoritmii de sortare pe fiecare dintre aceste dimensiuni și pe 3 seturi de date.

Cum se interpretează rezultatele experimentale?

Rezultatele experimentale oferă o idee generală despre ce timp de execuție scoate fiecare algoritm din cei analizați, aplicați pe mai multe seturi de date și pe dimensiuni diferite. Datele furnizate sunt orientative, deoarece timpii de execuție pot varia pe același număr de elemente și pe același set de date, fapt datorat sistemului de operare, gestiunii memoriei calculatorului precum și de limbajul de programare folosit.

5 Comparatie cu literatura

Pe baza experimentului facut putem observa ca fiecare algoritm de sortare are avantajele si dezavantajele sale, acestia comportandu-se diferit in functie de seturile de date pe care utilizatorul le introduce asa cum afirma si Ashok Kumar Karunanithi in A Survey, Discussion and Comparison of Sorting Algorithms[14].

6 Concluzii și direcții viitoare

În concluzie, această lucrare a furnizat o analiză cuprinzătoare a diferențelor dintre diferiți algoritmi de sortare. Am examinat performanța, eficiența și adecvarea lor pentru diferite tipuri de seturi de date. Astfel din această lucrare se pot extrage informații cu referire la modul în care se comportă algoritmii. Referitor la direcțiile viitoare voi studia comportamentul algoritmilor de sortare pe sisteme de calcul paralel.

7 Bibliografie

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *Introducere în algoritmi* (ediția a 3-a). Editura Tehnică.

[2] Sedgewick, R., Wayne, K. (2011). *Algoritmi* (ediția a 4-a). Editura Teora.

[3] Knuth, D. E. (1998). *Arta programării calculatoarelor, volumul 3: Sortare și căutare* (ediția a 2-a). Editura Teora.

[4] Mehta, D., Sahni, S. (2008). *Ghid de structuri de date și aplicații*. Editura Polirom.

[5] Goodrich, M. T., Tamassia, R., Goldwasser, M. H. (2014). *Structuri de date și algoritmi în Python*. Editura Teora.

[6] Manber, U. (1989). *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Publishing Company.

[7] Zaharie, D. (n.d.). Retrieved from `https://staff.fmi.uvt.ro/~daniela.zaharie/alg/algoritmica_cap4.pdf`

[8] Sortarea metoda bulelor. (n.d.). Retrieved from `https://www.pbinfo.ro/articole/5589/metoda-bulelor`

[9] Sortarea prin inserție. (n.d.). Retrieved from `https://www.pbinfo.ro/articole/5609/sortarea-prin-insertie`

[10] Sortarea prin interclasare. (n.d.). Retrieved from `https://www.pbinfo.ro/articole/7667/sortarea-prin-interclasare`

[11] Quicksort. (n.d.). Retrieved from `https://www.pbinfo.ro/articole/7666/quicksort`

[12] Laborator 09. (n.d.). Retrieved from `http://elf.cs.pub.ro/sda-ab/wiki/laboratoare/laborator-09`

[13] Sortarea prin selecție. (n.d.). Retrieved from `https://www.pbinfo.ro/articole/5605/sortarea-prin-selectie`

[14] Fulltext01.pdf. (n.d.). Retrieved from `https://www.diva-portal.org/smash/get/diva2:739880/FULLTEXT01.pdf`