Projektowanie Algorytmów i Metod Sztucznej Inteligencji

Algorytmy Sortowania

Dawid Krekora 254003

29 kwietnia 2022

Spis treści

1 Wprowadzenie

Potrzeba implementacji algorytmów sortowania nastała wraz z rozwojem technologicznym. Przy ogromnej ilości wszechobecnych danych, ręczne dopsowywanie elementów do zadanego klucza byłoby operacją czasochłonną i niepozbawioną błędów wynikających z dokładności pracy ludzkiej. Dlatego też opracowano narzędzia pomocnicze które miały za zadanie temu pomóc - a sam wybór algorytmu zależał już od rodzaju problemu który napotkamy.

2 Opis badanych algorytmów sortowania

2.1 Sortowanie przez scalanie

Sortowanie przez scalanie (ang. merge sort) jest typowym reprezentantem metody "dziel i zwyciężaj". Algorytm jest podzielony na 3 główne części:

- dzielenie głównego problemu na dwie równe części
- wywołanie rekurencyjne sortowania przez scalanie dla każdej z nich
- połączenie posortowanych elementów w całość

Klasa obliczeniowa sortowania przez scalanie jest taka sama dla każdegj złożoności problemu i wynosi $O(nlogn)$ co oznacza, że nasz algorytm jest wydajnym algorytmem którego czas wykonania przyrasta dużo wolniej od np. wzorostu kwadratowego - który jest mało optymalny jeżeli chcemy uzyskać jak najlepszą wydajność obliczeniową.

2.2 Sortowanie szybkie

Sortowanie szybkie (ang. quick sort) tak samo jak poprzednik opiera swoje działanie na metodzie "dziel i zwyciężaj". Schemat działania algorytmu wygląda bardzo podobnie do tego w merge sort:

- w losowy sposób wyznaczamy piwot który będzie naszym odnośnikiem do danych w strukturze danych (np. tablicy liczb)
- wykonujemy operacje porównań które doprowadzają do podziału tablicy na dwie części: z liczbami mniejszymi od piwota po lewej i większymi po prawej.
- wywołujemy procedurę quick sort dla dwóch części tablicy osobno
- połączenie posortowanych elementów w całość

Dużą zaletą quick sorta jest fakt, że w momencie dojścia do momentu połączenia tablic, elementy są już posortowane. Losowe wybieranie piwota niesie jednak ryzyko uzyskania najgorszego możliwego scenariusza - takiego w którym to piwot zawsze będzie wylosowany jako ostatni element tablicy. Powoduje to, że przy średniej złożoności problemu osiągamy klasę złożoności $O(nlogn)$, jednak w najgorszym przypadku musimy liczyć się z kwadratowym przyrostem czasu w stosunku do liczby przetwarzanych elementów: $O(n^2)$.

2.3 Sortowanie przez kopcowanie

Sortowanie przez kopcowanie (ang. heap sort) wykorzystuje do swojego działania drzewo binarne typu maksymalnego. To struktura danych w której wyróżniamy z góry określoną relację rodzic-dziecko pomiędzy elementami struktury (rodzic może mieć dwoje dzieci ale dziecko tylko jednego rodzica). I to właśnie ta struktura jest tutaj kluczowa w założeniu tego algorytmu sortowania. Elementy już w momencie umieszczania w kopcu muszą przejść podstawowe procedury sortowania tak, aby było spełnione założenie drzewa binarnego. To zapewnia niezwykle równą i efektywną klasę złożoności dla każdego z przypadków problemów: $O(nlogn)$.

3 Przebieg eksperymentów

Testowanie algorytów polegało w dużej mierze na sprawdzeniu ich zachowania przy rozwiązywaniu rzeczywistego problemu. Same testy opierały się na mierzeniu czasu trwania sortowania za pomocą konkretnych algorytów:

- w pierwszym etapie przygotowane zostały tablice o różnej wielkości (10000,50000,100000,500000,1000000)
- następnie tablice te zostały wypełnione zgodnie z założeniem testu (wszystkie elementy losowe, posortowane ale malejąco lub posortowane w pewnym %)
- dla każdego rozmiaru tablicy i kryterium podstawowego każdy test został przeprowadzony 100 krotnie, mierząc czas wykonania każdego z testu osobno
- wyciągnięto średnią z wyników - uzyskano średni czas działania algorytmów

Poniżej przedstawiono tabele z rezultatami testów dla trzech badanych algorytmów sortowania:

wszystkie wartości losowe			
N	MergeSort	QuickSort	HeapSort
10000	0.0208673	0.00802521	0.00769146
50000	0.0991466	0.0440124	0.0440154
100000	0.197439	0.0910949	0.0871245
500000	1.03079	0.508856	0.466176
1000000	2.1163	1.07343	1.01783
posortowane malejaco			
N	MergeSort	QuickSort	HeapSort
10000	0.0162585	0.00705142	0.00631057
50000	0.0726288	0.0373537	0.0323545
100000	0.143784	0.0787337	0.064115
500000	0.744671	0.444072	0.361252
1000000	1.51617	0.932056	0.762113
posortowane w 25%			
N	MergeSort	QuickSort	HeapSort
10000	0.018396	0.00797607	0.00785131
50000	0.0880546	0.0417776	0.0424538
100000	0.179291	0.0874307	0.0873265
500000	0.95129	0.486351	0.525914
1000000	1.95532	1.02354	1.21533
posortowane w 50%			
N	MergeSort	QuickSort	HeapSort
10000	0.0167267	0.00728159	0.00832416
50000	0.0816507	0.0401705	0.0466089
100000	0.165424	0.0829945	0.104418
500000	0.872609	0.467372	0.568238
1000000	1.78876	0.982349	1.17921

(a)

posortowane w 75%			
N	MergeSort	QuickSort	HeapSort
10000	0.0157754	0.00728159	0.00832416
50000	0.0755553	0.0401705	0.0466089
100000	0.15279	0.0829945	0.104418
500000	0.799957	0.467372	0.568238
1000000	1.63427	0.982349	1.17921
posortowane w 95%			
N	MergeSort	QuickSort	HeapSort
10000	0.0150228	0.00677202	0.00837795
50000	0.071549	0.0372122	0.0454213
100000	0.143823	0.0767756	0.0935324
500000	0.746917	0.430167	0.523837
1000000	1.52257	0.904994	1.1104
posortowane w 99%			
N	MergeSort	QuickSort	HeapSort
10000	0.0151636	0.00682129	0.00845528
50000	0.0705665	0.0373782	0.0455699
100000	0.142584	0.0763638	0.0938571
500000	0.73795	0.428194	0.524787
1000000	1.50214	0.900824	1.10476
posortowane w 99.7%			
N	MergeSort	QuickSort	HeapSort
10000	0.0151695	0.00686774	0.00860079
50000	0.0707803	0.0369086	0.0457381
100000	0.14254	0.0767682	0.0939921
500000	0.735997	0.426747	0.524132
1000000	1.49887	0.896166	1.10439

(b)

Rysunek 1: Rezultaty pracy algorytów - wyniki czasowe.

4 Podsumowanie i wnioski

Tabele pokazane na rysunku (Rysunek 1.) prowadzą do kilku istotnych wniosków:

- Każdy z zaprezentowanych algorytmów, pomimo posiadania złożoności obliczeniowej $O(nlogn)$ posortował tablice w różnym czasie
- najsłabiej wypadł algorytm sortowania MergeSort, był on zdecydowanie najwolniejszy od dwóch pozostałych. Wynika to z faktu, że stopień posortowania danych zapewnia mu jedynie brak potrzeby zamiany elementów miejscami. Etap porównań musi w dalszym ciągu przeprowadzić dla całego zakresu tablicy. Najbardziej kosztowna jest jednak ilość porównań które trzeba wykonać ponownie po dojściu do najniższego poziomu wywołania rekurencji
- algorytmy quicksort i heapsort rywalizowały ze sobą o miano najszybszego co jest dużym zaskoczeniem ponieważ w uniwersyteckich modelach algorytm heapsort okazywał się najwolniejszym z nich trzech. W naszym przypadku, przy mniejszym rozmiarze problemu, dorównywał quicksortowi, a nawet był od niego szybszy. O ile ten aspekt można wytłumaczyć, np. niekorzystnym rozmieszczaniem piwotu to w żaden sposób nie tłumaczy tak wyraźnej przewagi nad mergesortem.
- wniosek końcowy: algorytmy quicksort i heapsort wydają się być zaimplementowane w sposób poprawny, zgodnie z definicją książkową. Algorytm mergesort posiada pewne bliżej nieokreślone różnice w składni, przez co efekt jego działania nie jest idealny

5 Literatura

- "Data Structures and Algorithms in C- Michael T.Goodrich
- Ąlgorytmy, struktury danych i techniki programowania- Piotr Wróblewski
- www.wikipedia.com