Near Sorting Algorithms - When to use?

Asked 16 years, 2 months ago Modified 8 years, 2 months ago

Viewed 2k times

8

From time to time I browse the web and look for interesting algorithms and datastructures to put into my bag of tricks. A year ago I came across the Soft Heap data-structure and learned about near sorting.

The idea behind this is that it's possible to break the O(n log n) barrier of compare based sorts if you can live with the fact that the sort algorithm cheats a bit. You get a almost sorted list but you have to live with some errors as well.

I played around with the algorithms in a test environement but never found a use for them.

So the question: Has anyone ever used near sorting in practice? If so in which kind of applications? Can you think up a use-case where near sorting is the right thing to do?

algorithm language-agnostic sorting

Share

edited Sep 29, 2008 at 1:45

Improve this question

Follow

cynicalman
5,881 ● 3 ● 31 ● 30

asked Sep 28, 2008 at 15:02

Nils Pipenbrinck
86.2k ● 33 ● 155 ● 223

6 Answers

Sorted by: Highest score (default)

11

This is a total flying guess, but given the inherent subjectivity of "relevance" measures when sorting search results, I'd venture that it doesn't really matter whether or not they're perfectly sorted. The same could be said for recommendations. If you can somehow arrange that every other part of your algorithm for those things is O(n) then you might look to avoid a sort.

Be aware also that in the worst case your "nearly sorted" data *does not* meet one possible intuitive idea of "nearly sorted", which is that it has only a small number of inversions. The reason for this is just that if your data has only O(n) inversions, then you can finish sorting it in O(n) time using insertion sort or cocktail sort (i.e. two-way bubble sort). It follows that you cannot possibly have reached this point from completely unsorted, in O(n) time (using comparisons). So you're looking for applications where a majority subset of the data is sorted and the remainder is scattered, *not* for applications requiring that every element is close to its correct position.

Share Improve this answer edited Oct 18, 2016 at 1:09

Follow

answered Sep 28, 2008 at 17:18

Steve Jessop

279k ● 40 ● 469 ● 709

5

There are a lot of "greedy" heuristics where you periodically select the minimum of a set. The greedy heuristic is not perfect, so even if you pick the minimum you aren't guaranteed to get to the best final answer. In fact, the GRASP meta-heuristic, you intentionally introduce random error so that you get multiple final solutions and select the best one. In that case, introducing some error in your sort routine in exchange for speed would be a good trade off.

Share Improve this answer

Follow

answered Sep 29, 2008 at 5:00

David Nehme

21.6k ● 8 ● 81 ● 121

4

Just speculating here, but one thing I imagine is database query optimization.

A database query in a declarative language such as SQL has to be translated into a step-by-step program called an "execution plan". One SQL query can typically be translated to a number of such execution plans, which all give the same result but can have very varying performance. The query optimizer has to find the fastest one, or at least one that is reasonably fast.

Cost-based query optimizers have a "cost function", which they use to estimate the execution time of a given plan. Exhaustive optimizers go through all possible plans (for some value of "all possible") and select the fastest one. For complicated queries the number of possible plans may be prohibitively large, leading to overly long optimization times (before you even begin the search in the database!) so there are also non-exhaustive optimizers. They only look at some of the plans, perhaps with a random element in choosing which ones. This works, since there is usually a large number of "good" plans, and it might not be that important to find the absolutely best one -- it is probably better to choose a 5-second plan instead of the optimal 2-second plan, if it requires several minutes of optimization to find the 2-second plan.

Some optimization algorithms use a sorted queue of "promising" (partial) plans. If it doesn't really matter if you find the absolutely best plan, maybe you could use an almost-sorted queue?

Another idea (and I'm still just speculating) is a scheduler for processes or threads in a time-sharing system, where it might not be important if a certain process or thread gets its timeslot a few milliseconds later than if strictly sorted by priority.

answered Sep 29, 2008 at 3:11

Thomas Padron-McCarthy

27.6k ● 8 ● 54 ● 77

+1, I like the DB planning optimisation example. With process scheduling I would guess it is more complicated, since without guarantees on exactly "how and how much" the result fails to be perfectly sorted you could wind up with process starvation. – j_random_hacker Oct 22, 2009 at 8:26

2

A common application for near-sorting is when a human is doing the pairwise comparison and you don't want to have to ask them as many questions.

Say you have a lot of items you'd like a human to sort via pairwise comparison. You can greatly reduce the number of comparisons you need them to do if you're willing to accept that ordering won't be exact. You might, for example, not care if adjacent items have been swapped a long as the preferred items are at the top.

Share Improve this answer

Follow

answered May 27, 2009 at 4:46

James Tauber

3,456 ● 6 ● 29 ● 37

1

Anywhere

1. you are supposed to react fast,
2. you are not promising exact behavior to the client,
3. but internally you have some rules

you can use it. How about "not so strict" rule-based priority queue? Where would that be useful? Maybe thread/process/resource scheduling. In thread/process scheduling you are really not promising any one thread is going to go first, second, or last, but generally you want to give everyone some chance. You might want to enforce loose rule so it's preemptive, prioritized, blabla..

A resource schedule example would be responding to pizza delivery or shipping boxes of books to people etc. You can't use it where deterministic result is expected, but there are lots of example in real life where things are not so deterministic/predictable.

Share Improve this answer

Follow

answered Sep 29, 2008 at 3:27

Eugene Yokota

95.5k ● 45 ● 217 ● 320

-1

O(n log n) is already pretty fast. I don't think anyone would ever *start out* using a near-sort algorithm. You would start out with code that just does a complete sort (since your programming language of choice likely provides a `sort` function and not a `nearsort` function), and when you found empirically that the sort was taking too long, you would start to question whether your data *really* needs to be fully-sorted, and consider using a near-sort.

Basically, you would never even consider using a near sort unless you first discovered sorting to be a severe bottleneck in your program.

Share Improve this answer

Follow

answered Apr 28, 2011 at 7:05

Ryan C. Thompson

41.9k ● 30 ● 101 ● 162