Randomized, Data-Oblivious Sorting Algorithms

CS223 Final Project

Erik Godard

Spring 2015

Background

According to Goodrich, a *data-oblivious* sorting algorithm is one in which the sequence of compare-exchange operations the algorithm makes is independent of the original input [1]. This has applications is special purpose hardware as well as in communication protocols between multiple parties in contexts where privacy is important [1]. In this project, I tested two randomized data-oblivious sorting algorithms as well as one deterministic one as a baseline. I also tested a standard sorting algorithm that was not data-oblivous as a performance comparison. The main result of Goodrich's paper, annealing sort, provides a randomized data-oblivious sorting algorithm that runs in O(nlogn) time and sorts correctly with high probability. I did not discover any implementations of this algorithm in my research, so sought to determine this algorithm's performance and correctness in practice. In addition, I attempted to find performance optimizations to make in order to improve on these metrics.

Algorithms

I tested the following algorithms:

Bubble Sort - A very simple data oblivious sorting algorithm. It makes n passes along the array to be sorted, and at each step, it compares two adjacent elements, and if they are out of order, it swaps them. It makes n comparisons during each traversal of the array, and it is therefore a O(n^2) sorting algorithm, which is its main drawback It is data-oblivious since it makes the same comparisons regardless of the array that is input to it. For more information about Bubble Sort see http://en.wikipedia.org/wiki/Bubble_sort.

Quick Sort - While not a data-oblivious sorting algorithm, I use this to give a performance benchmark of an n log n sorting algorithm. The algorithm works by selecting an element at random to serve as the pivot. It then places each element on the appropriate side of the pivot and recursively sorts each side. It runs in time O(n log n). Python's standard sorting method uses an alternative sorting routine known as Timsort, so I instead opted to use a standard Quick Sort implementation which I found online. For more information about Quick Sort see http://en.wikipedia.org/wiki/Quicksort

Spin the Bottle Sort - Pseudocode from Goodrich [1]

```
while A is not sorted do
  for i = 1 to n do
    Choose s uniformly and independently at random from {1, 2, ..., i − 1, i + 1, ..., n}.
    if (i < s and A[i] > A[s]) or (i > s and A[i] < A[s]) then
      Swap A[i] and A[s].
```

The naive randomized data-oblivious sorting algorithm detailed in Goodrich's paper. It performs a linear scan through the array, and for each element, it selects another element from the array uniformly at random. If the elements are out of order, the algorithm swaps them. The algorithm can terminate when either a) The array is sorted or b) after a number of passes after which it is highly likely that the array is sorted. The second case makes the algorithm data-oblivious, but it also allows for the possibility that the returned result may be wrong. Goodrich shows in his paper that the algorithm runs in time O(n^2logn), which makes it highly impractical in practice.

Annealing Sort - Pseudocode from Goodrich [1]

```
for j = 1 to t do
  for i = 1 to n − 1 do
    for k = 1 to r_j do
      Let s be a random integer in the range [i + 1, min{n, i + T_j}].
      if A[i] > A[s] then
        Swap A[i] and A[s]
  for i = n downto 2 do
    for k = 1 to r_j do
      Let s be a random integer in the range [max{1, i − T_j}, i − 1].
      if A[s] > A[i] then
        Swap A[i] and A[s]
```

This is a refinement of Spin the Bottle Sort that leads to far better results. It mimics the ideas of annealing as it is found in the real world of metallurgy as well as their application to many optimization problems in computer science. It takes as input two sequences, a temperature sequence and a repetition sequence. The temperature sequences defines the range from which the algorithm can select elements to swap with during its execution. The repetition sequence contains the same number of elements as the temperature sequence, and it specifies how many

passes through the array at each temperature should be done. For each element in the temperature sequence, the algorithm makes a pass through the array and for each element, randomly selects one element greater than it and one element that is smaller than it. Each of these selected elements are within t places of the current element, and if the the current element and the selected element are out of place, they are swapped. The idea is to have a temperature sequence that is decreasing over time, which simulates the annealing process.

Questions

After first reading the paper, there were several questions that immediately came to mind as challenges I would have to tackle when implementing the annealing sort algorithm. The first was choice of constants in the annealing schedules. The paper leaves unspecified the choice of constants g,c, and q, and there is another term r that is only specified in big-Theta notation. Together, these constants control both the running time and the accuracy of the algorithm. In addition, I realized that randomness was important to this algorithm, and finding a good random generator would be important to the running of the algorithm. I also realized that I would need to spend time optimizing my loops to make them really tight, since I will spend much of the running time of the algorithm looping through the input array. Finally, I knew that I needed to generate good inputs to test the data on. There are many edges case in sorting that can cause unexpected behavior in sorting algorithms, so I knew I needed to develop a way to generate various types of arrays.

Testing

The arrays the algorithms were tested on consisted of arrays of n integers ranging from 0 to n-1. While Bubble sort is stable, the randomized algorithms tested here are not, so the lack of duplicates does not effect our results. The lack of negative numbers or non-integer numbers also does not affect the running time of the algorithms.

Sorted - An array that is already sorted.

Reversed - An array that is in reversed order.

Random Permutation - An array of specified length that is randomly selected from all n! possible permutations. This makes use of Python's random.shuffle method, which may introduce complications.

Within - An array where each element is at most d places away from its correctly sorted position. The generator works by splitting the array up into chunks of size d+1 and then randomly shuffling within each chunk. While this clearly does not generate arrays where each element is equally likely to be any location within d places of its correct location, it does provide a general sense of "close to sorted" that I am looking for. In testing the algorithms, when testing within I used log n for d, where n is the length of the array.

Uniform - An array containing values selected uniformly at random from the range [0,1). Computed using Python's random() function.

ZeroOne - Arrays consisting only of 0's and 1's, where each element is zero with probability one-half and one with probability one-half. While in practice this could be easily sorted with some type of Counting Sort, I wanted to test the way the algorithms handled duplicate values.

Technical Note

All development was done on my MacBook Pro Retina running OS X 10.9.5 using Python. The code was run and tested using the PyPy runtime, which offers significant performance improvements over the standard Python distribution (http://pypy.org/). The code consists of an Inputs class, which is used to generate sequences for testing, a Sorts class, which implements the actual sorting algorithms, as well as a Tests class, which contains methods that test the sorting algorithms. The first two classes also contain comprehensive unit tests. All code associated with this project can be found on this Github page: https://github.com/godarderik/cs223finalproject

Results - Small Arrays

The full results for small array sizes are found in the results.txt folder. Each sorting algorithm was run one hundred times on arrays of sizes 10,50,100,500,1000,10000. The exception to this was Spin the Bottle Sort, which was too slow to run on arrays of size 10,000. The following graphs provide the highlights of the results. Note that red denotes Bubble Sort, Green denotes Spin the Bottle Sort, Blue denotes Quick Sort, and Orange denotes Annealing Sort.

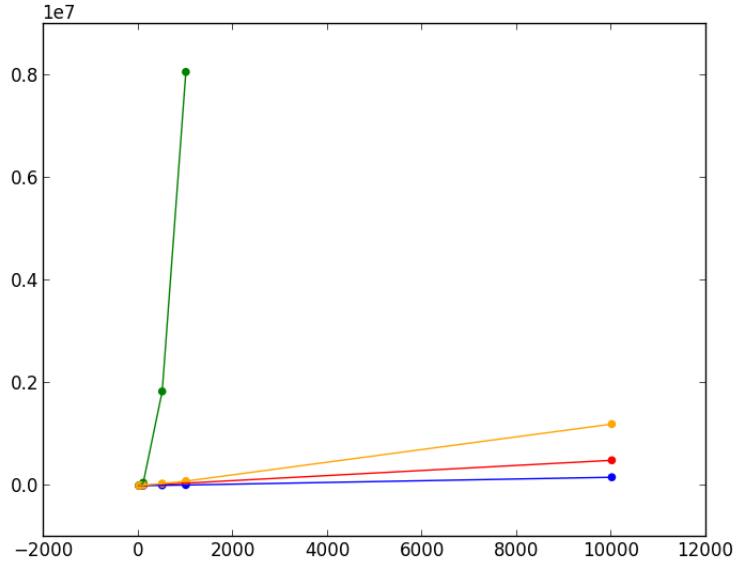

Figure 1: Comparisons Within

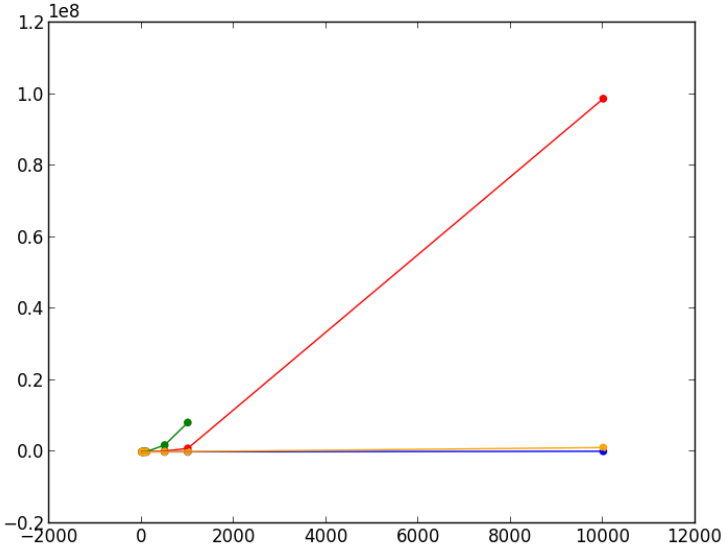

Figure 2: Comparisons Random

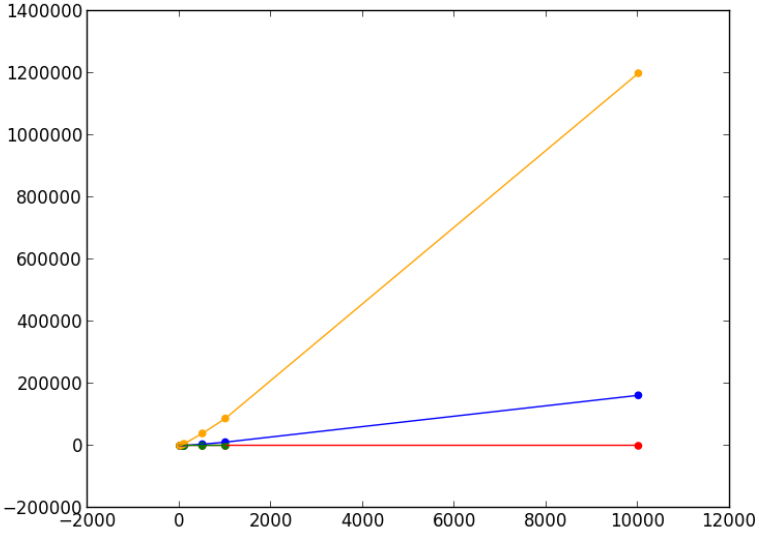

Figure 3: Comparisons Sorted

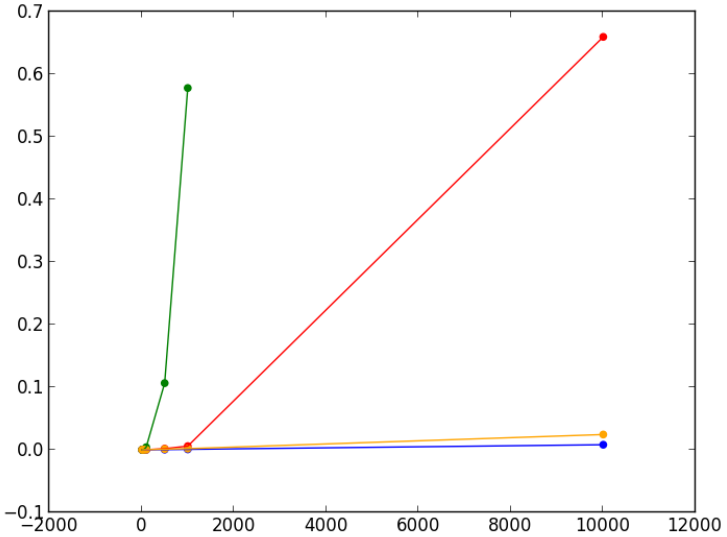

Figure 4: Time Random

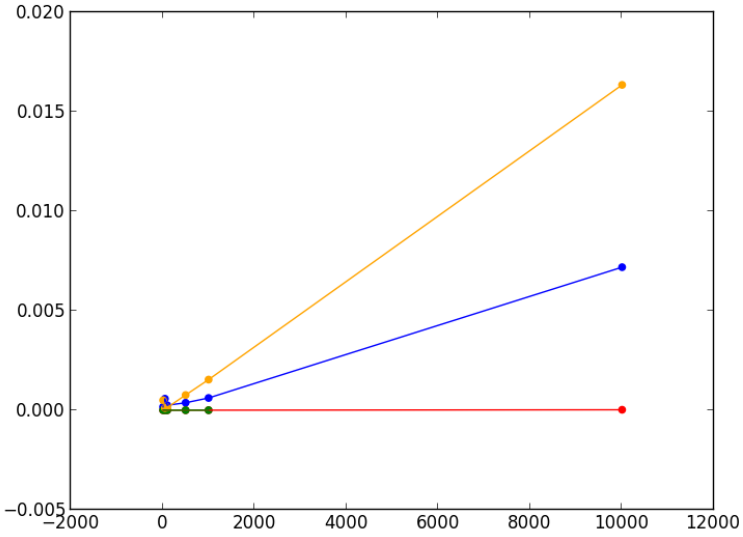

Figure 5: Time Sorted

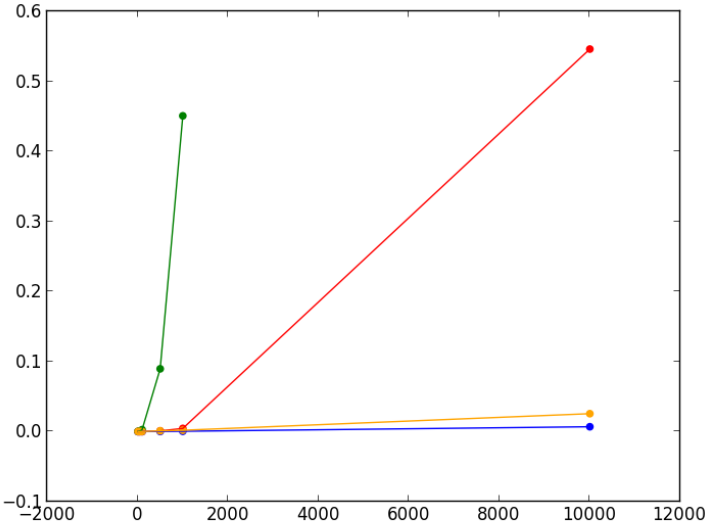

Figure 6: Time Reversed

Results - Large Arrays

Testing for large arrays was done by running algorithms on arrays of one million elements for one hundred times each. Spin the Bottle Sort was far too inefficient to be run on arrays of this size, and Bubble Sort could only be run on arrays that were already sorted or nearly sorted.

Sorted	Sorted Correctly	Time per Sort (sec)	Comparisons per Sort
Bubble Sort	1.0	0.00775487661362	0.0
Quick Sort	1.0	1.26330007553	25389846.9
Annealing Sort	1.0	3.78813307762	167999832.0

Random	Sorted Correctly	Time per Sort (sec)	Comparisons per Sort
Quick Sort	1.0	1.02843230009	25401829.1
Annealing Sort	1.0	5.77418357849	167999832.0

Within	Sorted Correctly	Time per Sort (sec)	Comparisons per Sort
Bubble Sort	1.0	0.116811516285	12999987.0
Quick Sort	1.0	1.12336622953	25431864.2
Annealing Sort	1.0	5.7689218688	167999832.0

Zero One	Sorted Correctly	Time per Sort (sec)	Comparisons per Sort
Quick Sort	1.0	0.140487535	1499959.3
Annealing Sort	1.0	6.00099039078	167999832.0

Reversed	Sorted Correctly	Time per Sort (sec)	Comparisons per Sort
Quick Sort	1.0	1.04661799431	25427998.4
Annealing Sort	1.0	5.13579659224	167999832.0

Uniform	Sorted Correctly	Time per Sort (sec)	Comparisons per Sort
Quick Sort	1.0	1.2569059515	25526290.8
Annealing Sort	1.0	6.22775731564	167999832.0

Analysis

Examining the results, one thing that is clear is that Spin the Bottle Sort is terribly inefficient, and indeed its running time is O(n^2 log n). Indeed, this makes it quite surprising that it's refinement, Annealing Sort is so effective. Looking at the above data, it is clear that in situations when the data is already sorted or nearly sorted, Bubble Sort is clearly the best choice. It runs nearly ten time faster than Quick Sort on these inputs. However, when the data is even remotely out of order, Bubble Sort performs very badly. On the other hand, in almost all of these cases, Annealing Sort with the optimizations I made performs on par with Quick Sort. It suffers from no correctness issues, sorting correctly in all of the instances on which I tested it. It runs is roughly 5-6 times the running time of Quick Sort on similar inputs. However, this implementation of Annealing Sort still poses several problems. While I observed no errors in my testing, I used a different annealing schedule than that given in the paper, which means that I know of no bound on the probability of error. In addition, while I have greatly optimized the constant factors as much as possible, it still runs noticeably slower than Quick Sort, which may pose problems in some applications. Another issues is the stability of the sorting algorithm; while it works well in the Zero - One case, it does not preserve the ordering of identical keys, which also may pose problems in some contexts. However, overall, the performance of Annealing Sort seems like a reasonable tradeoff for a data-oblivious sort, and indeed it is far better than Bubble Sort on most inputs.

One other issue is that my current implementations of Bubble Sort and Spin the Bottle Sort are not really data-oblivious. Since they terminate when the array is sorted, the sequence of compare exchange operations is dependent on the input — the algorithm will stop comparing and exchanging once the array is sorted. This is a really serious deficiency, since the whole point of implementing these algorithms is to obtain a sorting algorithm that is data-oblivious. Indeed, any sorting algorithm that waits until the input is sorted to halt cannot be data-oblivious, since its compare exchange operations depend on the state of the array. However, this just makes Annealing Sort that much better — not only is it more efficient than these algorithms in most instances, it is also data-oblivious. In a sense, this is a fundamental difficulty of data-oblivious sorting - we can't make any optimizations based on the state of the input array.

Improvements Made

Annealing Schedule - Goodrich proposes the following three phase annealing schedule

- ***Phase 1.*** For this phase, let $\mathcal{T}_1 = (2n, 2n, n, n, n/2, n/2, n/4, n/4 \ldots, q \log^6 n, q \log^6 n)$ be the temperature sequence and let $\mathcal{R}_1 = (c, c, \ldots, c)$ be an equal-length repetition sequence (of all c's), where $q \geq 1$ and $c > 1$ are constants.
- ***Phase 2.*** For this phase, let $\mathcal{T}_2 = (q \log^6 n, (q/2) \log^6 n, (q/4) \log^6 n, \ldots, g \log n)$ be the temperature sequence and let $\mathcal{R}_2 = (r, r, \ldots, r)$ be an equal-length repetition sequence, where q is the constant from Phase 1, $g \geq 1$ is a constant determined in the analysis, and r is $\Theta(\log n / \log \log n)$.
- ***Phase 3.*** For this phase, let $\mathcal{T}_3$ and $\mathcal{R}_3$ be sequences of length $g \log n$ of all 1's.

I started by setting the constants c, q, and g to 2,1, and 1 respectively, and then setting r to log n/(log log n). While this worked decently on smaller arrays, I started having problems with correctness on arrays with greater than 1,000 elements, where correctness would dip below 80%. I tried increasing c,q, and r, but while increasing c helped on smaller arrays, I still could not correctly sort arrays with 1,000,000 elements at all. The key insight that helped me achieve nearly 100% correctness on arrays of arbitrary size with multiplying r by a constant factor. In the paper, r is given as theta(log n/(log log n)). I found that multiplying this number by two led to dramatic improvements in the accuracy of the algorithm.

However, I further experimented with constant values and annealing schedules, and I managed to come up with a one phase annealing schedule that sorts correctly with virtually 100% accuracy. If q is set equal to zero, then the algorithm spends all of its time in Phase 1, so the temperature schedule becomes T = (2n, 2n, n, n, n/2, n/2, …, 2, 2, 1, 1). I found that with this temperature schedule, I could set c = 2, so R = (2,2,2,2,…,2), and achieve a successful sort virtually every time. While perhaps in theory one is not able to give a proof of correctness that shows the algorithm succeeds with high probability using this annealing schedule, I found that in practice this simple annealing schedule was all that was needed to sort any array correctly. In my tests of sorting 1,000 element arrays, I found that there were no incorrect sorts in 100,000 trials using this annealing schedule (results.txt). The running time of this annealing schedule is also much better than Goodrich's. While both run in time O (n log n), Goodrich's annealing schedule suffers from high constant factors that dramatically limit the algorithm's performance. Before

switching to this annealing schedule, the sorts took between 15 - 20 seconds to complete, and used upwards of five hundred million comparisons. The results shown above indicate that the current results are much better.

Loops - Goodrich's paper outlines this pseudocode for looping through the array:

for $i = 1$ to $n - 1$ **do**
  for $k = 1$ to r_j **do**
    Let s be a random integer in the range $[i + 1, \min\{n, i + T_j\}]$.
    if $A[i] > A[s]$ **then**
      Swap $A[i]$ and $A[s]$
for $i = n$ downto 2 **do**
  for $k = 1$ to r_j **do**
    Let s be a random integer in the range $[\max\{1, i - T_j\}, i - 1]$.
    if $A[s] > A[i]$ **then**
      Swap $A[i]$ and $A[s]$

While I originally implemented the algorithm following this specification exactly, I realized that first looping through the array in increasing order and then in decreasing order required extra work. While the analysis of the algorithm may require that the swaps with elements above be done before swaps with elements below, I found that in practice this made no difference. I combined the selection of an element above with the selection of an element below into one loop, and I found that this resulted in a several second speedup in my code.

Random Number Generation - I tried disabling the random number generation in my algorithm and timing how long it for my algorithm to run through its loops, and I found that generating random numbers was consuming over two-thirds of the running time of my algorithm. Previously, I had been using Python's randrange function, which given two numbers a,b, generated a number in the range [a, b). While this provided an acceptable degree of randomness, the fact that I potentially had to call it hundreds of millions of times in the course of the algorithm severely hindered the performance of the algorithm. I decided that I didn't need random numbers that were as "random" as what randrange was giving me. Instead, I opted to

precompute a list of random numbers between 0 and 1 and store them in a file. At the start of the algorithm, I then read this file into memory and then loop over the array of random numbers to select my random numbers. I transform these numbers to be in the range that I need, and when I reach the end of the array, I simply start back at then beginning. This does not change the fact that the algorithm is data-oblivious, since the compare exchange operations are still performed independent of input. After making this improvement, the accuracy of the algorithm was unchanged, but the time needed to run the algorithm decreased noticeably. I started by precomputing one million elements and then gradually reduced it until finding that my algorithm was still effective storing only ten thousand random numbers.

Further Improvements & Conclusion

While my implementation sort was able to achieve performance that is similar to that of Quick Sort, and by extension, other standard n log n sorting algorithms, there are still areas of my implementation that could use improvement. If I were ever to use this algorithm in an actual application, I would likely implement it in a language C++ that would make it easier to obtain better performance. As it stands, the five to six seconds the algorithm takes to sort may be to high in most circumstances. I would also like to investigate alternative annealing schedules that may yield fewer comparisons than my current one. As can be seen in my data tables, the rate-limiting step at this point seems to be the number of comparisons that I am making, which is seven to eight times the number that Quick Sort makes on random arrays. I experimented with other choices of constants in my annealing schedules, but the one I am currently using outperformed the rest. I wonder if there is an alternative annealing schedule that yield superior performance. In addition, I think it may be useful to investigate a way to perform a stable data-oblivious sort. Overall, I felt that I learned a great deal about optimizing randomized algorithms by doing this project.

References

[1] M. T. Goodrich. Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons. Algorithmica, pages 1–24, 2012.