Sorting Feature Retention and Algorithm Selection through Input Sampling

Chris Bennett
Department of Computer Science
University of Georgia
Directed Study with Dr. Don Potter
Fall 2004

Abstract

There are often several algorithms to solve a particular problem, but these solutions can differ in complexity depending on the problem's input. For example, Knuth discusses 25 sorting algorithms that vary in performance depending on the input's features or the hardware's architecture. Various features of the input determine the efficiency of a given algorithm, but these characteristics are often either unknown or more inefficient to enumerate than the solutions to the original problem. Randomly sampling the input, though, can provide a much smaller problem that retains the general features of the much larger input on which to test the algorithms for efficiency for algorithm selection. We show for the sorting problem that using a small sample of a large input not only retains the features of sortedness but also allows accurate ranking of algorithms for the large input. There is often, in fact, a net gain in performance even when considering the extra time for testing the algorithms on the samples.

I. Introduction

Sorting a list of records is a well-researched problem whose most efficient solutions still occupy some 25% of mainframe processing cycles [1]. Despite some of the most thorough investigation of any problem in computer science, sorting remains relevant because there still exist several interesting questions about the subject. This research focuses on two of these topics. First, is there a more efficient way to get a general notion of the features of the input? There exist a multitude of sortedness features as defined by [1,3,8] that enumerate certain characteristics of the input for a sorting problem, but these features only give a general idea about why certain algorithms are more effficient under certain conditions. Furthermore, they require knowledge about the input before sorting can occur, and acquiring this knowledge can often be more costly than sorting itself. Second, is there a profitable way to use such a mechanism to pick the "best" algorithm (within some acceptable margin of error) for a particular problem instance? If an algorithm could be selected such that the cost is negligible with respect to picking the "wrong" algorithm then such a mechanism would be very useful.

This research focuses on solving these problems by sampling a problem's input. We show that these samples not only retain the general features of sortedness of the original input but also that these samples can be used to predict the rank of the performance of various algorithms on the original input. A restricted random sampling technique that attempts to preserve at least the relative order of the selected elements produces smaller inputs on which to race whichever algorithms the user chooses in order to find the best for the larger input. We show that the samples

retain features of sortedness from the input and also that it can be a net gain to use the samples in a race between the best algorithms for various kinds of permutations. Specifically, the ranking of the algorithms on a sample reflects the ranking of the algorithm on the input itself, and the cost of the sampling and the race are less than the difference in the time between the first and second most efficient algorithms on the actual input.

The rest of the paper is organized as follows. First we present related work in sorting through an overview of the various types of sorting algorithms as well as the particular kinds of input for which they are most efficient. In the third section, we discuss our sampling techniques used in the experiments. Section IV shows how samples retain the larger input's features of "sortedness" defined in the literature such as inversions, number of runs, and maximum distance between inversions. Armed with the knowledge that particular algorithms are optimal with respect to certain features of sortedness [3], we race in a head to head competition some of the more efficient algorithms against each other on the sample in order to rank the algorithms for the particular problem instance in section V. We then discuss future work and finish with a summary.

II. Sorting

Sorting is a longstanding and well-researched problem. Given a collection of N items each with key K_i, sorting involves finding the permuation $p(1)p(2)p(3)...p(N)$ of this collection such that the keys of all elements are in nondecreasing order ($K_{p(1)} <= K_{p(2)} <= ... <= K_{p(N)}$) [1]. Over the years, a large number of algorithms and variations upon them have been proposed to solve this problem in the most time and space efficient manner – Knuth discusses about 25 of them [1]. Most of these algorithms fall into one or more of the following classes: insertion sort, exchange sort, selection sort, enumeration sort, or a special purpose sort. Each of course has its strengths and weaknesses with respect to ease of implementation as well as best, worst, and average case efficiency with respect to both time and space. Some well known efficient algorithms include quicksort, heapsort, shellsort, radix sort, and merge sort.

The sorting problem itself has been shown to be of $N \log N$ complexity, but the algorithms often deviate from this in the worst case in actual performace due to various features of the input or the hardware architecture. For example, quicksort has an average performance of $N \log N$, but the run time can degrade to $O(N^2)$ given particular permutations with certain characteristics. Often times an algorithm that outperforms another algorithm on a certain kind of data can be outperformed by that same algorithm on a different kind of data. The notion of having multiple algorithms that perform differently depending on the problem input is known as the algorithm selection problem. For sorting, it is common to simply select an algorithm such as quicksort that performs best in the average case. That is, quicksort may outperform all others nine times out of ten so it is used for all cases. Of course, we would ideally be able to select the most appropriate algorithm for each instance of the problem without adding prohibitive amounts of computation to the process.

Several attempts such as machine learning approaches [5] and adaptive sorting techniques [3] have attempted to sort faster by taking advantage of the number of inversions, runs (sorted subsequences), and other input characteristics in order to sort more efficiently. Results often provide a way to assess an algorithm's optimality with respect to certain measures of input sortedness (also called presortedness), but these require foreknowledge of each instance or calculation of such features on the fly. Other research has focused on refining certain algorithms such as the shellsort [2]. Sedgewick and others continue to search for an optimal set of increments for the shellsort and to provide tighter bounds on the efficiency of the sorting algorithm. Our

research focuses on two novel contributions to sorting, though. First, we try to show that these features of sortedness are retained in restricted random sampling of the input (we discuss these techniques in the following section). Second, we show that the performance of an algorithm on the sample of input is a good indication of its performance relative to other algorithms on the input itself. We believe this has applications not only in sorting but in the algorithm selection problem in general.

III. Sampling

A restricted form of random sampling and the law of large numbers provide the framework for the analysis of the sample data. The traditional use of sampling to estimate properties of an underlying population [4] is adapted in our research to estimate features of the input with respect to a particular problem (sorting in this case). By taking enough samples of an input with known features of sortedness, we can accurately measure just how well the samples retain the same features through analysis of their means and standard deviations compared to the populations. Our random sampling is "restricted" in the sense that we attempt not to disrupt the relative ordering of the input. For sorting, this order is fundamental to most notions of sortedness. Similarly for all problems that might use this analysis, random sampling should not introduce new features to the problem.

According to [6], there are five stages for sampling: the definition of the population of concern, the specification of a set of items or events that it is possible to measure, a specification of sampling method for selecting items or events from the frame, the sampling and data collecting, and a review of the sampling process. We define our population of concern as all permutations which contain various features of sortedness to different degrees. For the sortedness tests, details of creating the permutations are given in the next section. The racing experiments add an additional feature (number of runs) that is uniquely determined for each input. That is, a permutation has a varying number of runs, and varying numbers of random flips are made to the numbers to get the final input. We use these problem instances because nearly sorted sequences are very common in practice [9] and because certain algorithms are known to sort random permutations most efficiently. These inputs allow us to measure the retention of features for permutations that are known to make algorithms perform differently. To measure the relative similarity between input and sample, we define the set of items to measure (stage 2) as a subset of sortedness features including the number of inversions, the maximum distance between inversions, the maximum distance to travel to the position in the sorted permutation, and the number of runs. For the algorithm competition discussed later, we define the set of items to measure as the absolute machine time for sorting the permutation and its relative rank to other algorithms on the same input.

Perhaps the most important part of the sampling process for our purposes is the method of sampling. For this particular problem, it is key to retain the relative ordering of individual elements from the input to the sample. Likewise for other problems, the sampling method must not disrupt the input's features that determine the efficiency of a particular algorithm. For both the sortedness retention and racing sampling experiments, we use a restricted random sampling. It is critical to retain a true "snapshot" of the input, and so the sample is taken in order. That is, for a 10% sample, we simply take a random element from every ten elements. If an element k_i comes before element k_j in the input, the elements retain the same relative order in the sample. This is a fast (a fraction with respect to the size of the input) method to acquire a sample that should resemble the input itself with respect to features of sortedness. The fourth stage of the sampling

includes the actual experiments. The feature retention and racing results are discussed in the following section. We use three different sample sizes for each experiment. For feature retention, the sample sizes are 1, 5, and 10%, and for the races the sample sizes include 5, 10, and 25% (for the sake of space, we include results for a subset of these in the paper). The smaller sample sizes are used in the first experiments because we hoped to discover how small a sample could be and still retain sortedness features of the input. As expected, as the sample size approaches the input size, the results for both the sortedness and racing experiments are more accurate. The final stage involves a review of the sampling procedures, and this is an ongoing process.

Sampling can provide a much smaller input which can be used not only to extract the general features for the input but also to use for algorithm selection. Just as important is that these things can be done very efficiently. We imagine this powerful technique could prove very useful in algorithm selection problems where the efficiency of known algorithms varies even more widely. In the next two sections, however, we present our results regarding feature retention and algorithm seleciton with respect to sorting.

IV. Sortedness

There has been a great deal of work in identifying features of sortedness [1,3,7,9]. Much of it concentrates on identifying certain features such as the number of inversions or runs in the input as well as finding algorithms that are most efficient with respect to a particular feature (or a set of them). A set of eleven well-researched features are discussed in [3] as an extension of some of the material presented in much of the sorting literature [1,8]. These features lie at the heart of why certain algorithms outperform others in certain problem instances. With respect to certain features, certain algorithms are adaptively optimal [3]. That is, an algorithm may perform the fewest possible comparisons within a constant factor given a particular feature or set of them. If a user knows the features of the data prior to sorting, then he or she can take advantage of the sortedness research to identify the most appropriate sorting algorithm. This information is rarely known, though, and each feature requires at least linear and often polynomial time (or worse) to enumerate so that feature extraction can be more costly than the sorting process itself.

We argue that input sampling can be used to identify the general sorting features of the input in a much more efficient manner. We propose that a sample of an input contains the same general features of the original input. To show this, we sample permutations ranging from sorted to nearly sorted to random that contain a range of features of sortedness such as number of inversions, the greatest distance between two inversions, the maximum distance that an element must travel to its position in the sorted permutation, and the number of runs. These samples were then analyzed for the same features as the original input, and we compare the numbers in Figure 1. This table contains statistics for four different types of permutations and thirty samples taken from each. Second, we propose that the performance of different algorithms on the samples reflects the performance of the same algorithms on the original input. That is, if the features contained in the samples are similar enough to those in the input, whatever algorithm is best for the sample is best for the original input. To show this, we race known competitive algorithms on the samples as well as the original permutations with different features of sortedness. We discuss the sortedness sampling results here and the sampling and racing in greater detail in the following sections.

Each permutation begins as a perfectly sorted list of numbers between 1 and 100,000, and the number of changes represents the number of random flips between elements to get the final permutation. Thus, each successive list used represents an order of magnitude more random flips so that the set goes from almost perfectly sorted to a random permutation. The reader should

interpret the figure with care. For the last three features of sortedness we measured, the rates in the figure present a more accurate barometer of the similarity between the samples and the input because these measures are relative to the size of the list sorted (100,000 for the input but different sizes for the various samples). For the number of inversions, the rate is also a more meaningful measure. However, because the number of inversions can actually be polynomial with respect to the input size, the rates should be scaled to the same size in order to reflect the relationship between the sample and the input accurately. Figure 2 graphs these relative values.

So what do the numbers tell us? The figures show that the samples do indeed contain accurate reflections of the measured sortedness features for the original permutation. As the size of the sample grows, the feature retention becomes even more accurate. Likewise, as the permutation type goes from almost nearly sorted to 10,000 random perturbations, the features become more and more similar between the input and samples. In the tests for a collection of 100,000 elements that is nearly sorted (only 10 randomly flipped elements), the features measured are more skewed because it is likely few if any of the randomly flipped elements show up in the sample and thus make it completely sorted. There is much more deviation in the measures for the samples. Overall, though, for each permutation and set of sortedness features, the features of the samples become a more and more accurate reflection of the input's features as the sample size grows.

Changes	Size	# Inversion	Max Dist	Max Trvl Final	# Runs	Inv Rate	Dist Rate	Trvl Rate	Run Rate
10	**100000**	**762002**	**94192**	**87858**	**20**	**7.62002**	**0.94192**	**0.87858**	**0.0002**
	1000	86.533	79.867	79.867	0.267	0.087	0.080	0.080	0.000
	(std dev)	*176.816*	*162.982*	*162.982*	*0.521*	*0.177*	*0.163*	*0.163*	*0.001*
	5000	1691.633	1559.233	1559.233	0.867	0.338	0.312	0.312	0.000
	(std dev)	*1625.978*	*1554.877*	*1554.877*	*0.819*	*0.325*	*0.311*	*0.311*	*0.000*
	10000	8234.567	4551.167	4467.833	2.233	0.823	0.455	0.447	0.000
	(std dev)	*6931.570*	*2772.530*	*2707.316*	*1.524*	*0.693*	*0.277*	*0.271*	*0.000*
100	**100000**	**6132214**	**97762**	**84560**	**199**	**61.322**	**0.978**	**0.846**	**0.002**
	1000	626.767	364.733	357.600	1.867	0.627	0.365	0.358	0.002
	(std dev)	*584.912*	*269.665*	*264.275*	*1.432*	*0.585*	*0.270*	*0.264*	*0.001*
	5000	13724.633	3654.233	3263.700	8.800	2.745	0.731	0.653	0.002
	(std dev)	*5173.969*	*771.579*	*719.648*	*3.178*	*1.035*	*0.154*	*0.144*	*0.001*
	10000	63597.600	8369.833	7641.767	19.933	6.360	0.837	0.764	0.002
	(std dev)	*14333.335*	*905.874*	*843.215*	*3.750*	*1.433*	*0.091*	*0.084*	*0.000*
1000	**100000**	**66726712**	**99644**	**95658**	**1963**	**667.267**	**0.996**	**0.957**	**0.020**
	1000	7333.600	891.233	823.400	19.767	7.334	0.891	0.823	0.020
	(std dev)	*2312.302*	*76.130*	*91.200*	*4.408*	*2.312*	*0.076*	*0.091*	*0.004*
	5000	169803.367	4877.533	4471.400	100.067	33.961	0.976	0.894	0.020
	(std dev)	*17490.349*	*84.557*	*210.473*	*8.851*	*3.498*	*0.017*	*0.042*	*0.002*
	10000	666638.867	9857.467	9216.567	196.467	66.664	0.986	0.922	0.020
	(std dev)	*45244.563*	*65.241*	*304.339*	*13.093*	*4.524*	*0.007*	*0.030*	*0.001*
10000	**100000**	**572251590**	**99987**	**99727**	**16442**	**5722.516**	**1.000**	**0.997**	**0.164**
	1000	58784.733	986.467	935.600	168.233	58.785	0.986	0.936	0.168
	(std dev)	*4013.646*	*8.966*	*28.686*	*9.744*	*4.014*	*0.009*	*0.029*	*0.010*
	5000	1431734.867	4983.533	4816.333	818.800	286.347	0.997	0.963	0.164
	(std dev)	*48904.094*	*11.907*	*84.899*	*22.813*	*9.781*	*0.002*	*0.017*	*0.005*
	10000	5804830.200	9987.633	9809.267	1663.367	580.483	0.999	0.981	0.166
	(std dev)	*133782.807*	*8.401*	*130.002*	*28.026*	*13.378*	*0.001*	*0.013*	*0.003*

Figure 1

Let us examine the values gathered for the permuation with 1,000 random swaps more closely. For three of the measures of sortedness (the maximum distance between two inverted elements, the maximum distance an element must travel to get to its position in the sorted list, and the number of

runs in the list), the relative rates in the right-hand side of the table show exactly how a sample's features closely resemble the features of the input relative to its size. Given an input with rates of 0.996, 0.957, and 0.02 for the features just mentioned, the resemblance of the samples' features to the input grows more stark (and the deviation within the sample population decreases). In fact, the average relative rates for a sample of just 10% are 0.986, 0.922, and 0.02. For the other feature of sortedness (number of inversions), the graphs in Figure 2 better illustrate the relationship between the samples and the input because unlike the other features this measure can be polynomial with respect to the size of the input. If again we look at the chart for the permutation with 1,000 random swaps, we see that the number of inversions in the sample is closely related to the number of inversions in the input itself.

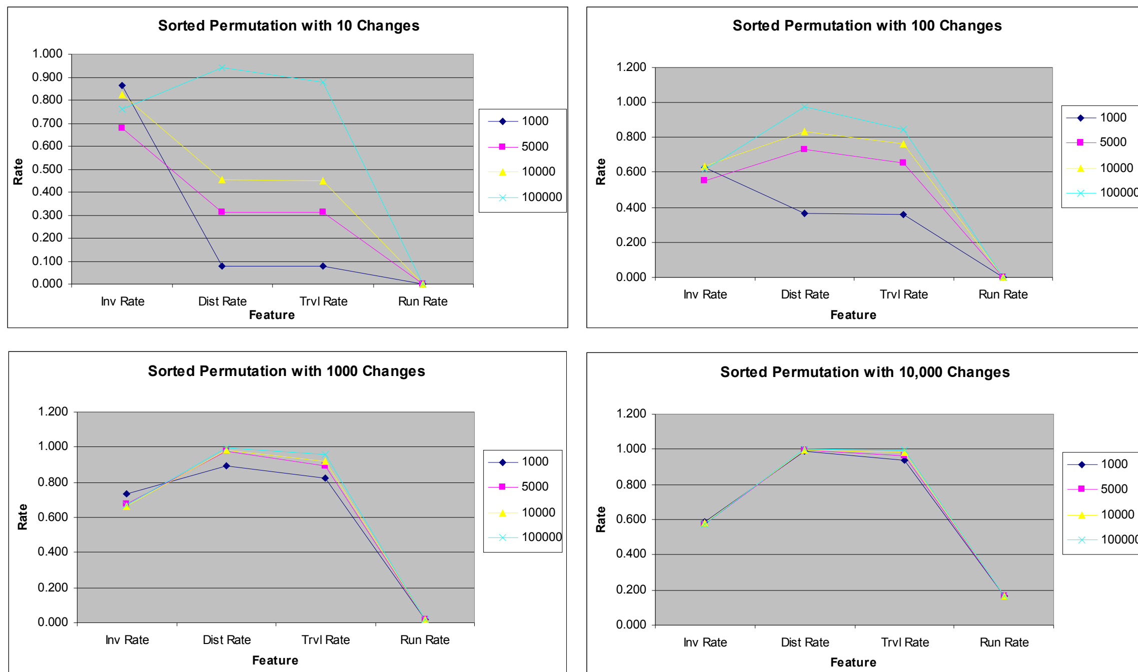

Figure 2

V. Algorithm Selection through Racing

In section 4, we showed that samples of an input to a sorting problem retain the features of sortedness found in the input itself. These results are interesting in and of themselves, but a practical application of this information provides a more compelling use of this research. Because we know that random samples of an input reflect the same inherent features of the problem, we investigated whether or not these samples could be used to predict the rank in performance of various algorithms on the much larger input. Our results in two different experiments show that the relative performance on the samples by the algorithms is a very good indication of the relative performance on the original problem. These experiments were run on a Sun Ultra 5 workstation with no other competing processes for computation time. It should be noted here that what we are racing is particular implementations of algorithms on a particular machine. It is very important to distinguish that this research makes no claims about a given algorithm but rather about a particular implementation of it on a particular hardware setup at a given time. That is, a particular

implementation of mergesort may outperform quicksort only on a specific architecture. Given any other hardware setup, the same implementations may perform differently. The context of the algorithm selection based on samples must be the same as for the original input.

We conducted two experiments to test our hypothesis. First, we ran our sample race with four different algorithms that represent different approaches to sorting – quicksort, mergesort, shellsort (with Knuth's increments), and heapsort. The input permutation types included ones ranging from already sorted input to nearly sorted to random permutations with different numbers of runs. Second, we tested five different shellsort increment sets on similar types of permutations. The results are shown in Figures 3 and 4 and discussed below.

Quicksort, Mergesort, Shellsort, and Heapsort

The absolute performance numbers for all four sorting algorithms refelect well-established efficiency expectations. More importantly, however, the similiarity between the relative performance of the algorithms on the sample versus the relative performance on the input itself is striking. As the numbers show, the outcome of a race on several samples is a very good indicator of the outcome of the race on the input itself. The process allows for accurate ranking of algorithm performance because the problem features for sorting are retained in the samples. For the sake of space, Figure 3 shows the statistics for 10% samples of permutations of size 1,000,000, but as one would expect, the accuracy becomes only more pronounced as the sample size approaches the input size. Smaller samples can be used as well, but there is the tradeoff with accuracy of feature retention versus race performance.

Let us examine and explain some specific numbers from the data before we discuss the broader implications. In the case where there is only one run in the input and the number of random swaps varies from zero to 10,000, we see how shellsort with Knuth's increments is the only solution whose performance generally degrades from just under 2.6 seconds to over 6 seconds as the input becomes more random. We also see how its performance on the samples grows slower as well (196 ms to 355 ms). For the remainder of the algorithms (quicksort, mergesort, and heapsort), the overall performance is fairly static. In fact, the relative rankings of these three algorithms with respect to the samples and the actual input is exact (with the exception of the case where there were 4,000 random swaps made). Shellsort falls down the rankings from first to third as the input (and its samples) grow more random. There is a fuzzy area when a few thousand swaps are made where the ranking for shellsort on the samples misrepresents its performance on the input, but the performance on the input is extremely similar for the other top algorithms. That is, the best algorithms are virtually tied to a certain degree.

From the data, it is obvious quicksort and shellsort with Knuth’s increment set are the most competitive algorithms on average for the permutations chosen. If we were to use just these algorithms in the race, we would indeed see a net gain in performance compared to the common practice of just using a single sorting algorithm for all problem instances. For example, when the input is nearly sorted, shellsort outperforms quicksort anywhere from a second to two seconds. The total time to race these algorithms on samples is around .5 seconds, and the sampling time itself is negligible (milliseconds). Thus the one-half second cost of algorithm selection far outweighs the cost of choosing the wrong algorithm. These benefits only grow as the input grows larger and larger.

1 Run		QS	MS	SS	HS
Sorted	Input	4694.00	5030.00	2599.00	7060.00
	Samp Avg	350.00	396.20	196.13	546.43
	Std Dev	52.35	28.41	12.74	91.88
100 Changes	Input	4226.00	4959.00	3256.00	6090.00
	Samp Avg	332.13	404.40	224.53	518.60
	Std Dev	22.68	15.40	8.63	24.10
1000 Changes	Input	4561.00	5108.00	3878.00	6032.00
	Samp Avg	343.70	429.47	290.13	526.90
	Std Dev	26.12	18.67	16.39	14.94
2000 Changes	Input	4374.00	5001.00	4527.00	6094.00
	Samp Avg	375.73	449.03	323.27	563.03
	Std Dev	90.59	83.61	44.54	91.51
3000 Changes	Input	4382.00	4985.00	4361.00	6520.00
	Samp Avg	333.10	405.07	307.63	517.37
	Std Dev	26.74	14.01	17.68	23.10
4000 Changes	Input	5379.00	4985.00	5242.00	6071.00
	Samp Avg	333.83	409.70	318.97	524.40
	Std Dev	24.43	23.06	13.79	55.18
5000 Changes	Input	4123.00	5227.00	5561.00	6095.00
	Samp Avg	336.93	430.00	321.90	522.37
	Std Dev	28.53	53.57	13.23	19.38
10000 Changes	Input	4248.00	5294.00	6222.00	6196.00
	Samp Avg	326.07	424.63	355.03	497.13
	Std Dev	18.61	16.97	13.38	15.78

2 Runs		QS	MS	SS	HS
Sorted	Input	4232.00	4647.00	3053.00	6478.00
	Samp Avg	329.47	385.53	251.70	496.27
	Std Dev	18.03	15.48	14.06	13.70
100 Changes	Input	4363.00	4867.00	3712.00	5931.00
	Samp Avg	335.97	393.97	269.33	499.70
	Std Dev	19.04	12.89	8.57	15.65
1000 Changes	Input	4040.00	4952.00	4223.00	5928.00
	Samp Avg	329.30	415.10	311.63	497.33
	Std Dev	19.00	11.78	9.59	11.66
2000 Changes	Input	4076.00	4869.00	4517.00	5953.00
	Samp Avg	339.83	413.57	325.90	496.13
	Std Dev	19.22	17.81	9.93	11.88
3000 Changes	Input	4244.00	4975.00	4790.00	5868.00
	Samp Avg	335.20	419.93	338.97	499.07
	Std Dev	20.02	19.99	11.19	16.64
4000 Changes	Input	4097.00	5263.00	5239.00	5806.00
	Samp Avg	334.40	412.07	344.30	500.73
	Std Dev	19.40	24.58	11.49	11.67
5000 Changes	Input	4504.00	5105.00	5206.00	5957.00
	Samp Avg	340.13	428.87	352.50	499.17
	Std Dev	21.84	11.27	11.45	16.11
10000 Changes	Input	4596	5112	5627	5861
	Samp Avg	332.73	423.43	378.70	496.93
	Std Dev	15.66	12.73	15.19	12.60

Figure 3

These numbers are very encouraging. Not only do they seem to predict the ranking of performance on a problem instance for competing algorithms, but they also provide proof that race outcomes can lead efficient algorithm selection that can often improve overall performance. This has many implications in algorithm selection in general and in particular could prove useful in problems where algorithms on large inputs run in polynomial or exponential time. We discuss this further in future work.

Four Increment Sets for Shellsort

Shellsort is a family of sorting algorithms where the performance of different implementations depends almost exclusively on the selection of the increment set [1,2]. The shellsort acts as a multi-pass insertion sort where at the i-th pass, elements that are increment[i] distance apart are sorted. The final increment is 1, and so the final pass is a full insertion sort that compares all neighbors. Much of the research is focused on finding the most optimal increment set [2], but we have selected five increment sets that have different known efficiencies to test in the race. These include Shell, Knuth, Sedgewick, Hibbard, and Gonnet.

The absolute and relative performance numbers are interesting for the shellsort for several reasons. First, the overall numbers show that again the race on samples is a good indicator of the relative performance on the actual input. Like the previous experiments with four general sorting algorithms, the absolute numbers reflect expectations for performance for each increment set. Where there are mis-predictions in the rankings, it is usually at the millisecond level of error. Also,

the numbers show exactly when a "bad" increment set (Shell and Gonnet) actually can be useful. When an input is almost nearly sorted, the supposedly poorly performing shellsorts are actually more efficient than the more optimal increment sets.

		Time (ms)					Rank				
1 Run	Sorted	Shell	Knuth	Sedgewick	Hibbard	Gonnet	Shell	Knuth	Sedgewick	Hibbard	Gonnet
Sorted	Input	1288.0	2340.0	2950.0	2307.0	566.0	2	4	5	3	1
	Samp Avg	80.6	192.1	273.1	185.0	60.2	2	4	5	3	1
	Std Dev	5.9	13.6	15.2	3.7	11.5					
100	Input	2473.0	3131.0	3873.0	2970.0	2394.0	2	4	5	3	1
	Samp Avg	113.9	226.8	309.8	219.4	90.2	2	4	5	3	1
	Std Dev	12.8	13.2	19.1	14.7	9.7					
1000	Input	14417.0	4080.0	4238.0	3822.0	14488.0	4	2	3	1	5
	Samp Avg	224.5	275.5	356.6	261.5	207.9	2	4	5	3	1
	Std Dev	19.7	13.8	17.6	14.3	15.4					
2000	Input	26005.0	4225.0	5105.0	4003.0	26906.0	4	2	3	1	5
	Samp Avg	336.0	300.3	371.0	281.2	321.8	4	2	5	1	3
	Std Dev	18.1	11.3	17.8	11.9	22.7					
3000	Input	39637.0	4434.0	5296.0	4141.0	41287.0	4	2	3	1	5
	Samp Avg	429.5	315.1	383.5	291.2	419.6	5	2	3	1	4
	Std Dev	33.1	17.5	19.2	12.5	22.4					
4000	Input	50070.0	4828.0	4636.0	4367.0	50733.0	4	3	2	1	5
	Samp Avg	517.0	320.4	379.7	314.2	511.4	5	2	3	1	4
	Std Dev	29.0	9.3	15.2	48.7	22.4					
5000	Input	68838.0	4942.0	5378.0	4441.0	64692.0	5	2	3	1	4
	Samp Avg	621.4	330.9	385.8	308.4	615.5	5	2	3	1	4
	Std Dev	38.3	14.7	23.0	14.0	39.7					
10000	Input	137645.0	5502.0	4942.0	5273.0	132965.0	5	3	1	2	4
	Samp Avg	1144.9	368.4	420.9	341.4	1155.6	4	2	3	1	5
	Std Dev	68.8	24.0	56.8	25.2	64.3					

2 Runs	Sorted	Shell	Knuth	Sedgewick	Hibbard	Gonnet	Shell	Knuth	Sedgewick	Hibbard	Gonnet
Sorted	Input	3425269.0	3032.0	3676.0	5869.0	3231933.0	5	1	2	3	4
	Samp Avg	17221.4	255.0	346.1	314.0	17452.8	4	1	3	2	5
	Std Dev	567.1	14.9	13.1	13.4	577.1					
100	Input	3159905.0	3744.0	4254.0	6449.0	3232610.0	4	1	2	3	5
	Samp Avg	17226.0	282.1	370.2	337.4	17488.1	4	1	3	2	5
	Std Dev	505.8	17.2	22.8	16.8	510.8					
1000	Input	3185335.0	4296.0	4503.0	6936.0	3526836.0	4	1	2	3	5
	Samp Avg	17373.9	321.3	396.7	366.2	17607.8	4	1	3	2	5
	Std Dev	550.3	12.5	13.9	11.7	570.2					
2000	Input	3153525.0	4543.0	5406.0	7188.0	3534194.0	4	1	2	3	5
	Samp Avg	17209.4	338.4	408.9	380.2	17687.2	4	1	3	2	5
	Std Dev	554.9	14.9	17.6	14.9	671.0					
3000	Input	3129788.0	4854.0	5644.0	7457.0	3217149.0	4	1	2	3	5
	Samp Avg	17466.9	348.0	414.0	391.9	17711.5	4	1	3	2	5
	Std Dev	525.5	15.5	20.7	18.4	598.6					
4000	Input	3393631.0	4947.0	4842.0	7685.0	3260314.0	5	2	1	3	4
	Samp Avg	17713.0	369.7	434.8	410.7	17827.8	4	1	3	2	5
	Std Dev	713.0	23.3	55.8	25.1	565.7					
5000	Input	3691560.0	5040.0	5023.0	7717.0	3532517.0	5	2	1	3	4
	Samp Avg	17597.7	361.3	422.5	407.2	17855.7	4	1	3	2	5
	Std Dev	791.0	16.0	16.3	14.8	729.3					
10000	Input	3556984.0	5691.0	5431.0	8197.0	3559411.0	4	2	1	3	5
	Samp Avg	17943.2	388.6	437.7	432.4	18003.9	4	1	3	2	5
	Std Dev	727.5	21.5	18.8	16.5	808.9					

Figure 4

A final interesting issue the numbers bring up is the performance of the bad increment sets when the permutation is transitioning from nearly sorted to a more random input. At this point, the

poorly performing shellsorts actually perform well on the samples. This is caused by some features of the insertion sort and the increment sets themselves. Where the inputs are in between nearly sorted and random, insertion sort performs an order of magnitude better on the samples (scaled to the actual size of the input) because the samples are small enough that the percentage of random flips that make it through are not enough to send the performance into polynomial time. On the large input, however, the insertion sort degrades to polynomial time. The "good" increment sets avoid this degradation, though. Further research into this "zone of degradation" could yield interesting results with respect to optimal increment sets.

VI. Future Work

We believe input sampling applies to problem domains beyond sorting. Specifically, input sampling and racing could allow a user to rank and pick the most efficient algorithm for a given input on a given machine for any problem for which known algorithms vary in performance depending on input features. Future work includes expansion of these sampling and racing ideas to other problems that require polynomial or even exponential time. We present initial results about the family of shellsorts and various increments used for it in section 5, but work in finding optimal increment sets might benefit from this research as well because of the relationship between performance on samples and input for shellsorts. In addition, there is much work left to be done to define a solid theoretical foundation for exactly why restricted random sampling preserves features of a problem's input.

VII. Summary

We have attempted to show the value of input sampling with respect to two problems in sorting. First, we are able to estimate the features of an input by extracting the same features from a restricted random sample of the input. Because features are retained by these samples and certain algorithms perform differently according to these features, we show that the relative performance of an algorithm on the samples is a very good indicator of the relative performance on the original input. This is a novel and efficient way not only to compare sorting algorithm performance on a given input but also to extract generalized sortedness features of large inputs. There are also much broader implications, though. Many difficult problems for whom certain features of the input determine the most efficient algorithm could take advantage of this technique to significantly reduce computation time and the cost of selecting the wrong algorithm.

IX. References

[1] Knuth, D., *The Art of Computer Programming*, Volume 3. 1975.

[2] Sedgewick R., *Analysis of Shellsort and Related Algorithms.*

[3] Estivill-Castro and Wood, D., *A Survey of Adaptive Sorting Algorithms,* ACM Computing Surveys Vol. 24 No. 4. 1992.

[4] http://mathworld.wolfram.com/Sampling.html

[5] Guo, H., *Algorithm Selection for Sorting and Probabilistic Inference: A Machine Learning-*

Based Approach. Doctoral Dissertation, Kansas State University. 2003.

[6] http://en.wikipedia.org/wiki/Law_of_large_numbers

[7] Bansal, S.S. B. Tech Project Report, Part II. *Sorting using Presortedness of Input Sequence*.

[8] Manilla, H., *Measures of Presortedness and Optimal Sorting Algorithms*, IEEE Transactions on Computers, (Vol. C-34. No. 4, April 1985).

[9] Burge, W.H., "Sorting, trees, and Measures of Order", *Information and Control*. 1958.