Input Feature Approximation Using Instance Sampling

Chris Bennett
Department of Computer Science
University of Georgia
benoit@uga.edu

Walter D. Potter
Department of Computer Science
University of Georgia
potter@uga.edu

Abstract

Features (or properties) of problem inputs can provide information not only for classifying input but also for selecting the right algorithm for a particular instance. Using these input properties can help close the gap between problem complexity and algorithm efficiency, but enumerating the features is often at least as costly as solving the problem itself. An approximation of such features can be useful, though. This work defines the notion of group input properties and proposes an efficient solution for their approximation through input sampling. Using common statistical techniques, we show that samples of inputs for sorting and graph problems retain the general properties of the inputs themselves.

1. Introduction

For various computational problems, certain input features can differentiate one instance from another. These properties of an input reflect unique characteristics that can determine algorithm performance, instance classification, and other feature-dependent aspects of a problem. These features can take two forms – individual and group. The former type reflects certain inherent attributes of an element of the input, but the latter type encompasses attributes derived from the interaction or relationships between elements of an input. An input identity is determined by such group properties, and this work focuses on these features of inputs because of their usefulness in many applications such as sorting and graph problems. For sorting, for example, a permutation of orderable keys has several features of *sortedness* defined in the literature [1, 7] where sortedness refers to how ordered a permutation already is. For example, the number of inversions in a permutation is equal to the number of pairs of elements at i and j in a permutation where the ith element is larger than the jth element but $i < j$. In the permutation {2,5,4,3}, there are 3 inversions – {5,4),{5,3}, and {4,3}.

Graphs are also the basic input structures to a large number of computational problems. In a graph, several aggregate properties exist that characterize a particular instance. For example, relatively simple features such as density and the number of strongly connected components can provide a sense of the type of graph as well as help in identifying a more appropriate solution to use for a given problem. For example, in finding a minimum spanning tree (MST) of a graph, two common solutions are Prim's and Kruskal's algorithms. Their relative performance, though, can depend on the density of the graph. Other more complex features such as clustering or the number of cliques are costly to calculate, but a good estimate is often enough information for many tasks. Thus a statistically valid approximation of input properties can provide enough information to improve the efficiency of any feature-dependent problem including algorithm selection and input classification.

Aggregate input features are often expensive to extract relative to the cost of solving the problem. For example, calculating the number of inversions in a permutation is polynomial with respect to the size of the input while the sorting problem itself is only N log N. In addition, an exact enumeration of input properties is frequently unnecessary. That is, an approximation of the aggregate features provides accurate enough information for a particular task such as input classification. This work focuses on feature approximation through input sampling. Given a particular input instance, a model for restricted sampling is outlined such that features can be approximated with certain statistical confidence. We outline the methods in a later section, but certain aspects of the sampling method deserve special mention here. Because we are interested in group attributes of the input, it is critical that the sampling method not introduce new information or relationships between elements of the input. For example, when sampling a permutation to be sorted, the relative order of the keys should be maintained in the sample. A random sample that picks 10% of the keys in random order will yield a result that bears no resemblance to the input. We explain this and the rest of the sampling process in later sections.

Approximations of aggregate feature for permutations of sortable keys and of graphs provide the empirical data for

this research. For sorting, we measure several well-known features of sortedness for inputs and sets of samples drawn from them. Each input represents a permutation type whose sortedness ranges from completely sorted to almost sorted to random and whose features are known to affect the performance of various sorting algorithms such as quicksort, shellsort, mergesort, and heapsort. For graphs, we generate a variety of graph instances of multiple sizes and edge probability. Our feature set includes the density and the probability of edge existence. The sampling and feature approximation processes are described in detail for both kinds of inputs in Sections 3, 4, and 5.

The rest of the paper is organized as follows. Section 2 presents background information and related work on features in specific problem domains and also in general. Section 3 reviews the sampling process, and sections 4 and 5 discuss the results of the sorting and graph experiments, respectively. In section 6, we discuss related and future work, and finally we summarize the research in section 7.

2. Features

As described in the introduction, features of an input can differentiate one instance from another. Like fingerprints, they are what make a particular input unique. Features of data sets can be viewed as two types – individual and group. Individual properties do not depend on the interaction between the elements that make up an input, but aggregate properties are dependent on the interaction or relationships between such elements. For example, given a room of 500 people, the average age for a subset of 30 of those people provides an accurate estimate of the average age for the entire group because an individual's age is thought to be independent of another person's age. Determining the number of groups of related people in the room, though, depends on the relationships between the elements. There is no individual feature which you can sum or average to answer such a question because this sort of group property derives its value based on the relationship between the particular elements rather than independent features of the elements. In addition, this interdependence often causes these sorts of properties to be inherently more computationally complex than individual properties. This research attempts to discover whether or not group properties such as these can be estimated through sampling of the input.

Other research has focused on using such input characteristics to analyze and choose algorithms for various problems. Work in sortedness (the notion of how sorted a permutation is) has not only defined several aggregate features such as inversions, the number of runs, and the maximum distance between inversions [1,7], but researchers have also classified various sorting algorithms as optimally efficient with respect to particular features [3]. For example, the Local Insertion Sort algorithm is optimal with respect to inversions – the amount of work grows with the size of the input and its number of inversions.

Feature extraction commonly involves machine learning techniques for document, image, or other complex resource processing. For documents, words are analyzed to construct feature vectors that provide a means to compare documents for similarities. For more complex document processing, Brin introduces the idea of sampling to explore the solution space for rule sets in order to avoid an otherwise prohibitively costly search [10]. Feature extraction in image processing is similar in that it is more specific than a general framework for inputs to a given computational problem. Other work has focused on approximating features in a specific domain as well. Srivastava and others approximate the properties of queries (in particular, the maximum distance between inversions) by using windows in a data stream and extrapolation using a probability distribution [11]. Other research that exploits group properties includes Manilla's work on feature-optimal sorting. These, however, concentrate on the output of a process rather than preprocessing. A general framework for feature extraction and approximation for a given computational problem does not seem to exist.

This work proposes a simple but powerful method of random sampling to extract and estimate properties of inputs of all sorts. Given an unbiased sampling process and careful definition of populations, samples, and estimated attributes, statistical theory allows for accurate approximation within a deterministic degree of error. Properties of samples are shown to reflect the properties of the input relative to size. An important corollary to this is that a sample retains the implicit features of the input. That is, enumeration is not necessary for many applications such as algorithm selection because the samples are structurally similar to the input. In related work, we experiment with using both enumerated and implicit features of samples for algorithm selection [12] and input classification. Several avenues for future work (discussed in section 6) also exist in other problem domains.

3. Sampling

Sampling is used in countless disciplines to estimate various attributes of a population by measuring those same attributes for a small subset of elements from the population. According to statistical theory, the estimates obtained from the samples reliably approximate the true attributes when sound sampling methods are used [9]. Intuitively, the accuracy improves as the number of samples approaches the size of the population, but perhaps

more importantly a surprisingly few number of samples are required for an accurate approximation [9]. Traditionally, these techniques have been used to estimate the mean or total of individual features. This research, however, uses sampling to obtain reliable estimates of group properties for two types of input for common computational problems – sorting and graph problems.

The sampling process can be broken down into five steps.

1. Identify the population
2. Identify the attributes to estimate
3. Identify the sampling technique
4. Extract estimates
5. Review of the sampling process

We will step through each part of the sampling process for both the sorting and graph experiments, but we provide a broad overview here. In defining the population and samples, we must identify the fundamental unit of a set. These individuals are meaningless with respect to a given problem if they are divided any further. For sorting, for example, this unit is a key in the input permutation, and a sample will consist of an (in-order) subset of these keys. We define the attributes to be approximated as any group attribute of the input. This is, of course, a very simple scheme, but the sampling technique itself contains the most crucial part for feature retention and approximation. A modified random sample without replacement is used in order to avoid introducing new relationships between elements that do not exist between them in the population. Because we are measuring group properties, it is critical to maintain the structure of the specific input instance in samples without introducing new information or relationships. Finally, to measure error in our sampling, we calculate the standard deviation for each feature we measure.

Sampling can provide a much smaller input with which to approximate features of the input and also to use for algorithm selection because of the implicit features retained in the sample's structure. Just as important is that these things can be done very efficiently. This powerful technique proves very useful in input classification and 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 in samples with respect to sorting and graphs.

4. Sorting and Feature Approximation

Sorting is one of the most thoroughly researched problems in computer science. Given a collection of N items each with key K_i, sorting involves finding the permutation $p(1)p(2)p(3)...p(N)$ of this collection such that the keys of all elements are in non-decreasing order ($K_{p(1)} <= K_{p(2)} <= ... <= K_{p(N)}$) [1]. Over the years, researchers have proposed a large number of algorithms and variations upon them to solve this problem in the most time and space efficient manner – Knuth discusses over 20 of them. Most of these algorithms fall into one or more of the following classes: insertion sort, exchange sort, selection sort, and enumeration sort, or a special purpose sort. Each has its positives and negatives with respect to ease of implementation as well as time and space efficiency. Some well known comparison-based efficient algorithms include quicksort, heapsort, shellsort, and merge sort.

The sorting problem has been shown to be of $N \log N$ complexity (for comparson-based sorting), but the algorithms themselves often deviate from this in the worst case in actual performance due to various features of the input. For example, quicksort has an average performance of $N \log N$, but the run time can degrade to 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 like 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 input features such as the number of inversions, runs, 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 presortedness, but these require foreknowledge of each instance or calculation of such properties on the fly. Other research has focused on refining certain algorithms such as the shellsort [2]. It is not entirely clear what causes certain increment sets to perform differently for different types of inputs, but Sedgewick and others continue to search for optimal sets of increments for shellsort and to provide tighter bounds on the efficiency of the sorting algorithm.

Our research shows that these features of sortedness are retained in restricted random sampling of the input. 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 relative to its size. To show this, we sample large 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 are then analyzed for the same features as the original input relative to size, and we compare the numbers in Figure 2. This table contains statistics for four different types of permutations and thirty samples taken from each, and it is discussed in detail in the results subsection. First, though, we review the sampling procedure for this input type.

4.1. Population and Samples

Because we seek to find estimates for group features, the sampling process must avoid introducing new relationships between elements of the population that did not previously exist. For permutations of sortable keys, the population consists of all elements in a given permutation, and samples consist of a subset of these keys. For the sorting problem, though, the relative order of these elements is crucial to maintain in the sampling process. Aggregate features such as inversions and numbers of runs depend on this relative ordering, and it must be preserved in the samples. By defining each element as a key in the input instance, we are able to maintain this relative ordering in samples because of the sampling procedure.

4.2. Group Properties

Properties of sortedness that we measure include the number of inversions, the number of runs, the maximum distance between any inverted pair, and the maximum distance that an element must travel to its final sorted spot. Each of these is explained in more detail and with examples below. Because these features are measured with respect to the size of the input, results are relative to the sample and input sizes where appropriate.

4.3. Sampling

We use a modified form of random sampling. 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. For example, for a 50% sample, we simply take a random element from every two elements. In figure 1, we can see how four elements {1,4,5,8} are sampled from the input {1,2,4,3,5,6,7,8} while their relative order is maintained. 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 resembles the input itself with respect to features of sortedness. For the experiments, each input 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. We have chosen sample rates of 1, 5, and 10% in order to get an accurate picture of the tradeoff between accuracy in retention and the efficiency in sampling and extraction. Each successive input represents an order of magnitude more random flips so that the set goes from almost perfectly sorted to a random permutation, and the various sample sizes provide a wide range of measurements for feature approximation and retention.

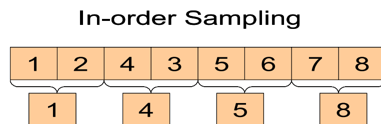

Figure 1. Sampling permutations.

4.4. Results

Figures 2 and 3 present a summary of the empirical data. We will walk through a specific example in a moment, but first we describe the general results. The figures show that the samples accurately retain the measures of sortedness from the original permutation. As expected, as the sample size 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. These reductions in variation for both larger samples and more randomized input are intuitively rooted in the idea that deviations from sortedness are more likely to show up when either more of the input is sampled or there are more perturbations from which to pick. In the tests for a nearly sorted collection of 100,000 elements (with 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. This would of course make the sample completely sorted. There is more deviation in the measures for the smaller samples, but this is expected. Rates relative to the size of the input are also listed in the table because these sortedness features are measured with respect to the size of the input (or sample). Overall, 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.

The graphs in figure 3 show the measures of sortedness for different samples as well as the input relative to their actual sizes. Because each feature is measured with respect to the size of the input (or sample size in the case of the samples), a relative measurement of the estimates provides a better picture of how well various sample sizes perform for different features. For the last three features of sortedness we measure, the rates in the figure present a more accurate indicator 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 meaningful measure. Because the number of inversions can actually be polynomial of the input size, though, the rates should be scaled to the same size in order to reflect the relationship between the sample and the input accurately. Progressing through the graphs from nearly sorted (10 changes) to random (10,000 changes), we see how each feature is more closely approximated by all sample sizes. In addition, we can also see how in general the progressively larger samples also more closely estimate the measured properties.

Changes	Size	#Inversion	Max Dist	Max Trvl Final	#Runs	Inv Rate	Dist Rate	Trvl Rate	Run Rate
10	**100000**	**762002**	**94192**	**87858**	**20**	**7.62**	**0.94**	**0.88**	**0**
	1000	86.533	79.867	79.867	0.267	8.653	0.080	0.080	0.000
	(std dev)	*176.816*	*162.982*	*162.982*	*0.521*	*17.682*	*0.163*	*0.163*	*0.001*
	5000	1691.633	1559.233	1559.233	0.867	6.767	0.312	0.312	0.000
	(std dev)	*1625.978*	*1554.877*	*1554.877*	*0.819*	*6.504*	*0.311*	*0.311*	*0.000*
	10000	8234.567	4551.167	4467.833	2.233	8.235	0.455	0.447	0.000
	(std dev)	*6931.570*	*2772.530*	*2707.316*	*1.524*	*6.932*	*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	62.677	0.365	0.358	0.002
	(std dev)	*584.912*	*269.665*	*264.275*	*1.432*	*58.491*	*0.270*	*0.264*	*0.001*
	5000	13724.633	3654.233	3263.700	8.800	54.899	0.731	0.653	0.002
	(std dev)	*5173.969*	*771.579*	*719.648*	*3.178*	*20.696*	*0.154*	*0.144*	*0.001*
	10000	63597.600	8369.833	7641.767	19.933	63.598	0.837	0.764	0.002
	(std dev)	*14333.335*	*905.874*	*843.215*	*3.750*	*14.333*	*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	733.360	0.891	0.823	0.020
	(std dev)	*2312.302*	*76.130*	*91.200*	*4.408*	*231.230*	*0.076*	*0.091*	*0.004*
	5000	169803.367	4877.533	4471.400	100.067	679.213	0.976	0.894	0.020
	(std dev)	*17490.349*	*84.557*	*210.473*	*8.851*	*69.961*	*0.017*	*0.042*	*0.002*
	10000	666638.867	9857.467	9216.567	196.467	666.639	0.986	0.922	0.020
	(std dev)	*45244.563*	*65.241*	*304.339*	*13.093*	*45.245*	*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	5878.473	0.986	0.936	0.168
	(std dev)	*4013.646*	*8.966*	*28.686*	*9.744*	*401.365*	*0.009*	*0.029*	*0.010*
	5000	1431734.867	4983.533	4816.333	818.800	5726.939	0.997	0.963	0.164
	(std dev)	*48904.094*	*11.907*	*84.899*	*22.813*	*195.616*	*0.002*	*0.017*	*0.005*
	10000	5804830.200	9987.633	9809.267	1663.367	5804.830	0.999	0.981	0.166
	(std dev)	*133782.807*	*8.401*	*130.002*	*28.026*	*133.783*	*0.001*	*0.013*	*0.003*

Figure 2. Summary statistical data from sorting experiments.

An example will help make sense of the numbers. Before specifics, though, we need to understand how to compare the number of inversions for different sizes of permutations. Because inversions are polynomial with respect to the size of the input (that is, a sequence with 10 times as many elements can have approximately 100 times as many inversions), we must multiply the samples' inversions by the square of the sampling factor (10^2 for a 10% sample, 100^2 for a 1% sample). Other properties such as the maximum distance between inversions are linear with respect to the size of the permutation so that no additional scaling is required.

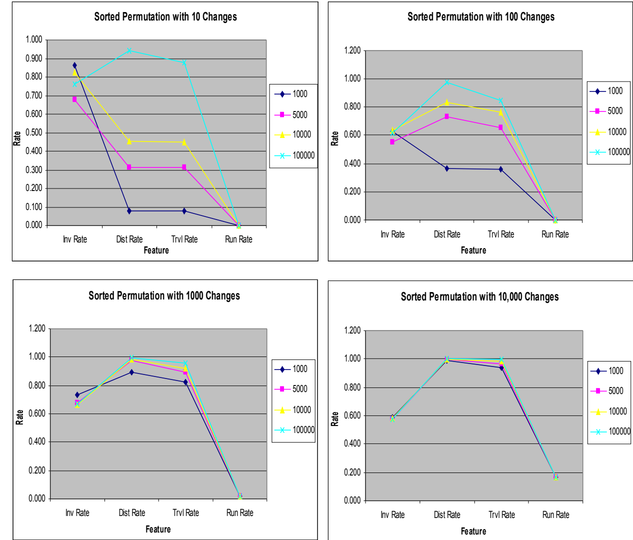

Figure 3.

Let us compare two types of numbers from the sample data in figure 2 – intra-permutation where numbers from different sample rates of the same permutation are compared and intra-permutation where numbers from the same sample rate but from different permutation types are compared. For the former, the number of inversions for the 1 and 10% samples for the 1,000 perturbation input provides a good example. The original input has 66,726,712 inversions. The average number of inversions for the smaller sample size is 7,333.6 while the larger sample averages 666,638.9 inversions (73,336,000 and 66,663,890 scaled to the same size as the original input). Both of these numbers provide an accurate estimate of the number of inversions expected in the original input (and the average of the larger sample proves to be very accurate). In addition, the variation with the sample set is more stable in the larger sample. If scaled to input size, the standard deviations of 2,312.3 and 45,244.6 (not scaled) show us that any given sample with a 10% sample rate is more likely to be closer to the mean than a sample with a smaller sample rate.

The number of inversions for the 5% samples of the 10-flip and 1,000-flip inputs provides an illustrative example for inter-permutation comparisons. For each input, there are 762,002 and 66,726,712 inversions respectively. For the nearly sorted input, the 5% sample averages 1,691.6 (scaled value of 676,653.2) inversions while the random permutation averages 169,803.4 inversions (scaled value of 67,921,346.8). Both 5% samples provide good estimates for the number of actual inversions in their respective original inputs, but the sample for the more random permutation is most accurate. The variance

within the sample set measures the accuracy in a different way. The scaled standard deviations for the nearly sorted input (650,391.2) and for the 1,000-flip (6,996,196) show that variation from the expected case for any given sample is much larger in the 10-flip input. Both of these inter-permutation comparisons reveal a common characteristic about sortedness features and sampling – the more sorted an input is, the more deviation there will in the estimate for that feature.

Both the specific numbers as well as the general trends in the sorting feature experiments reveal two facts. First, as expected, larger samples do a better job of approximating the measure properties of the inputs, but even small samples can be very accurate in their estimates. This of course makes sense when carried to the extreme of larger and larger sample rates that eventually end with 100% sampling. Second, samples more accurately estimate features of the input when there is more disorder in the input. That is, the more perturbations from sorted, the more likely these deviations will show up in a given sample. For example, a 5% sample of a list that is perfectly sorted except for one element has only a 1 in 20 chance of detecting that single out of order element. This is an interesting effect that is explored further in work that uses sampling techniques for algorithm selection [12].

5. Graphs and Feature Approximation

Another common input to a vast number of computation problems is the graph. Stored in many forms such as adjacency lists and adjacency matrices, the graph simply defined is a collection of nodes connected by edges. Of course, graphs can be much more complicated than that (we point the reader to any standard textbook on graph theory for further reading). For these experiments, though, we consider only directed graphs that use adjacency lists for storage. Methods for sampling undirected graphs as well as those stored with other methods could easily be adapted since such cases are reducible from our general directed graph case.

Graphs are used to model an endless number of real-world situations, and the problems to be solved are often very computationally complex. For example, several NP-complete problems require exponential time to search the solution space. Modeling using graphs often requires input classification, but such analysis can also be very costly. We hope to show there are several applications using graphs where sampling can prove very useful. By approximating graphs with structurally similar but significantly smaller samples, we (intend to) show that certain problems including input classification can be performed with a deterministic rate of error.

To create our graphs, we use a standard model for random graphs generation. Given a certain number of nodes and a probability that an edge exists between any two given nodes, we iterate through every pair of nodes and randomly decide if an edge indeed exists between them. For our graphs, we then remove any nodes from the input that have no edges from or to them. Because sampling only makes sense for large graphs, we generate large graphs of varying densities and edge probabilities. Because of memory constraints, we are able to generate graphs with 1,000, 2,500, and 5,000 nodes with edge probabilities of 0.001, 0.01, and 0.1 so that we have nine input graphs. For sampling, we use sample rates of 5, 10, and 25%. We will first discuss the sampling procedure followed by a discussion of the results.

Population and Samples We define the population for sampling graphs as the set of edges and their associated nodes for a particular input. Two types of entities (edges and nodes) complicate the sampling procedure because we must account for the effect that both have on the structure of the graph. A node can be associated with multiple edges in two capacities (as the source or target nodes), and samples must contain only relationships (edges) that exist between nodes in the original input.

Group Properties For graphs, we analyze the inputs and their samples for density as well as the edge probability. Because the density is polynomial with respect to the number of nodes, the measurements must be scaled for clearer meaning. Edge probability (defined previously as the probability that an edge exists between two nodes) should remain approximately the same from input to sample. This statistic, though, is for the graph as a whole. As mentioned in the definition of the samples, an edge can only appear in the sample if there is an edge between the same nodes in the input.

Sampling A modified random sample is used for the graphs. In order to avoid introducing new information into any samples, we simply need to avoid creating new edges between nodes. The hard part of the sampling is maintaining the general structure for a graph for different kinds of graphs. There are three sampling methods shown in Figure 4. We use two different kinds of graphs to show how the first two sampling methods are faulty while the combined method works better.

There are two types of input graphs shown. The first is a “hub and spoke” graph where all outer nodes are connected to a hub node in the middle, and the second graph is a fully connected graph with five nodes. Let us assume our sample rate is 50% for this example. One sampling method could be simply to select 50% of the nodes in the graph, and use any nodes that begin in those nodes. We can immediately see why this would not be a good method for the hug graph – the only way any edges even make it into the sample is if the hub is selected, and there is a good chance it will not be. For the fully

connected graph, say we pick the three nodes randomly. If we add all edges from those nodes (and thus implicitly the target nodes of those edges), we are left with a sample that contains all the nodes but is missing several edges. This new graph is very different from the input in features and structure.

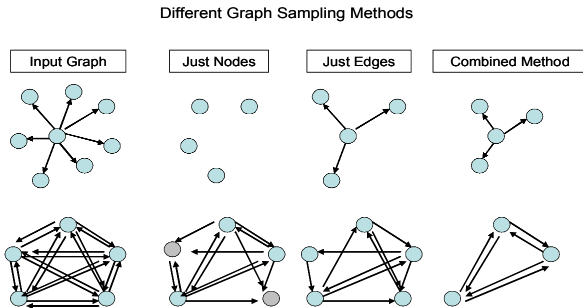

Figure 4.

A second sampling method would involve simply selecting random edges. For the first graph, this seems to work well. The hub node now stands an excellent chance of making it into the sample, and in fact the result seems to resemble the input structurally. For the fully connected graph, though, this sampling method begins to break down. By selecting 10 of the 20 edges, we will most likely include all of the nodes in the graph but only half the edges. This output has the same problems as the first sampling method – the structure and features of the input are lost. The hybrid method, though, where both nodes and edges are taken into consideration works well. First we select enough nodes to satisfy the sample rate by examining the edge list. We randomly select edges, and we add the source and target nodes of that edge if they are not already in the node list for the sample. Once there are enough nodes, we traverse each one's adjacency list to find edges which we can add to the edge list of the sample. If the target node of the edge is one of the nodes randomly selected in the previous step, then we add this edge to the sample's edge list. This method not only allows for making the chance of a node being in the sample proportional to the number of edges in which it participates, but it also limits the number of nodes in the sample while maintaining certain structure from the input. This sampling method generalizes well for many types of graphs.

Results The data for sampled graph features show a strong correlation between the properties of carefully sampled subgraphs of the original input. Both density and edge probability are approximated well by the samples. The table in figure 5 provides the numbers for analysis while the graphs in figure 6 show the relative performances for each sample size and input graph. For the sake of space, we show the average densities and edge probabilities for the various input graphs without showing the standard deviation data.

EP 0.001	1000 Nodes			2500 Nodes			5000 Nodes		
	Density	EP	# Edges	Density	EP	# Edges	Density	EP	# Edges
5%	0.5423723	0.0170391	17.138965	0.6320264	0.00552	72.461832	0.7572667	0.0030498	187.9536
10%	0.5939401	0.0093577	37.537017	0.7505797	0.0032729	172.10793	1.0191296	0.0020524	505.89591
25%	0.690087	0.0043557	109.03375	1.1260888	0.001964	645.53042	1.7844783	0.0014375	2214.5375
Input	1.6012658	0.002533		2.726123	0.0011889		5.0235697	0.001012	

EP 0.01	1000 Nodes			2500 Nodes			5000 Nodes		
	Density	EP	# Edges	Density	EP	# Edges	Density	EP	# Edges
5%	0.9828889	0.0195053	49.144444	1.7630349	0.0140522	220.37937	2.9970598	0.0119659	749.26494
10%	1.4896964	0.0148015	148.96964	3.0175208	0.0120443	754.38021	5.5373047	0.0110687	2768.6524
25%	3.0399352	0.0121404	759.9838	6.8113066	0.0108929	4257.0666	13.09204	0.0104709	16365.05
Input	9.928	0.009928		25.066	0.0100264		50.095	0.010019	

EP 0.1	1000 Nodes			2500 Nodes			5000 Nodes		
	Density	EP	# Edges	Density	EP	# Edges	Density	EP	# Edges
5%	5.531098	0.1095847	276.5549	13.04087	0.1040228	1630.1087	25.476549	0.1017575	6369.1372
10%	10.400211	0.103453	1040.0211	25.542156	0.1020593	6385.539	50.515432	0.1009502	25257.716
25%	25.564891	0.1020963	6391.2228	62.97451	0.1007001	39359.069	125.42974	0.100317	156787.17
Input	99.818	0.099818		250.208	0.1000832		499.7124	0.0999425	

Figure 5.

We can see again that larger samples naturally approximate the properties of the input more accurately. In addition, graphs with more edges per node also sample well. That is, small samples on sparser graphs can vary more widely in feature measurements for similar reasons as in sampling for sorting. For larger graphs, though, even smaller samples do a good job of approximation.

In figure 6, the graphs show how larger samples do indeed approximate features better. In addition we also see how larger graphs yield better estimates across all edge probabilities. For example, in all three input types (edge probability of 0.001, 0.01, and 0.1), all three sample sizes prove remarkably more accurate in predicting the edge probability of the input for graphs of size 5,000. Even for 2,500 nodes, the 5% sample does a reasonable job. For smaller inputs, though, small samples deviate more widely from the properties of the original input. We also see how with graphs of lower connectedness, smaller samples over-estimate features such as edge probabilities. For the first input graph, the 5% sample overestimates by nearly an order of magnitude. For the subsequently denser graphs, though, even the small samples become more and more accurate. In the final graph, the average of the 5% samples varies from the input by only 10% of the original value. For larger graphs, the accuracy only improves.

Future work will focus on creating not only larger random graphs but testing very large graphs that reflect real-world situations in order to test applications of the sampling process.

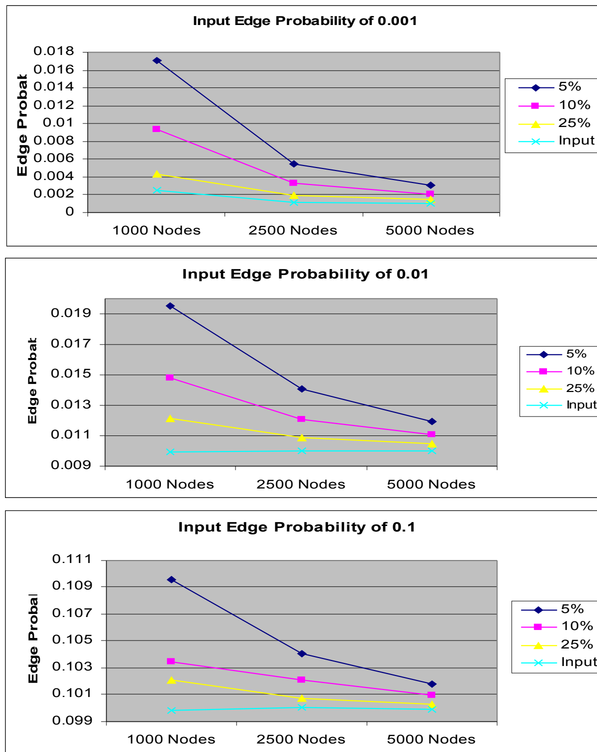

Figure 6. It is important to note the order of magnitude difference in scale between each successive graph. The similarity in the graph trends belie the significant differences in relative deviation for each graph.

6. Related and Future Work

Related work attempts to exploit input sampling techniques for the sorting algorithm selection problem [12]. We compare the performance of common sorting techniques on various inputs as well as their samples in order to find out if the best algorithm for a particular input can be determined by the inherent features in a sample of it. In addition, various increment sets for shellsort are analyzed for performance both on large inputs and their samples for insight into increment set optimality. Other future work using these sampling techniques concentrates on classifying large graphs in addition to other algorithm selection problems with inputs such as graphs. Finally, we would like to explore using sampling results as seed inputs for heuristic algorithms for computationally complex problems to improve both runtime as well as accuracy.

7. Summary

We have shown that several features of two common problem inputs (permutations of sortable keys and graphs) are retained in samples of those inputs. For the sorting problem, our techniques provide accurate estimates of several properties of sortedness such as the number of inversions. Intuitively, larger sample rates provide more accurate predictions for each feature, but even smaller (and thus quicker) samples provide good estimates on average. For graphs, the density and probability of connection between any two nodes is also accurately approximated even with relatively small samples of large graphs. In several situations, estimates of input properties provide accurate enough information for algorithm selection or classification, for example. We explore various applications of sampling in other research.

References

[1] Knuth, D. Knuth. (1998). *The Art of Computer Programming: Sorting and Searching*, Volume 3. Reading, Massachusetts: Addison-Wesley.

[2] Sedgewick, R. (1996). “Analysis of Shellsort and Related Algorithms”. Fourth European Symposium on Algorithms, Barcelona.

[3] Estivill-Castro, V., & Wood, D. (1992). “A Survey of Adaptive Sorting Algorithms”. *ACM Computing Surveys,* Vol. 24, No. 4, 441-476.

[5] H. Guo. (2003). *Algorithm Selection for Sorting and Probabilistic Inference: A Machine Learning-Based Approach*. Doctoral Dissertation, Kansas State University.

[6] S.S. Bansal. (2002). “Sorting using Presortedness of Input Sequence. Project Report, Part II”. Retrieved September 20th, 2004, from: http://www.cse.iitk.ac.in/research/btp2002/98357.html

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

[8] W.H. Burge. (1958). “Sorting, trees, and Measures of Order”, *Information and Control*.

[9] S.K. Thompson. (1992). *Sampling*. New York, NY: John Wiley & Sons, Inc.

[10] Brin, S. and Page, L. (1999). “Dynamic Data Mining: Exploring Large Rule Spaces by Sampling”. Retrieved February 5th, 2005, from http://dbpubs.stanford.edu:8090/pub/1999-68

[11] Srivastava, U., Babu, S., & Widom, J. (2003). “Monitoring Stream Properties for Continuous Query Processing”. Technical Report, Stanford University. 2003. Retrieved January 17th, 2005 from http://dbpubs.stanford.edu:8090/pub/2003-23

[12] Bennett, Chris. (2005) “Sorting Algorithm Selection Using Input Sampling”. Technical Report, University of Georgia. 2005.