A Virtual Laboratory for Study of Algorithms

Thomas E. O'Neil and Scott Kerlin
Computer Science Department
University of North Dakota
Grand Forks, ND 58202-9015
oneil@cs.und.edu

Abstract

Empirical studies of algorithms are highly useful in both instruction and research. This paper describes a software application that facilitates experimentation with algorithms. The application is a virtual laboratory that allows the user to run an algorithm on randomly generated instances of a problem and immediately view a chart that summarizes the results. The application was developed for the PARTITION problem, but its design is sufficiently general to be easily adapted to a wide range of decision and optimization problems. The instance generator and the decision algorithm are essentially plug-ins. The application can easily accommodate any instance generator with two parameters, such as the number of nodes and number of edges for a graph, or the number of clauses and width of clauses for a boolean expression. The decision algorithm can also be replaced with an optimization algorithm, which returns a numeric value instead of a Boolean result.

We anticipate that the virtual algorithms laboratory will be useful for both instruction and research. It can be adapted to present common instructional lessons such as comparison of search and sort algorithms. It can also be used for the analysis and development of improved algorithms for hard problems such as PARTITION, CLIQUE, and SATISFIABILITY.

1 Introduction

The development of efficient algorithms for basic problems in computing is a core activity in Computer Science. Writing and implementing an algorithm is usually the first step in the process. To achieve full understanding of the scope and complexity of the problem, however, we must test the algorithms on a range of automatically generated problem instances. We also inevitably think of variations which improve performance for at least some subsets of the problem space. This creates the need to run the different versions of the algorithm over the same set of problem instances to compare performance. In running algorithms on automatically generated subsets of the problem space, we also may discover that some instances are much easier or harder than the rest. There may well be a critical region in the problem space where all algorithms exhibit their worst-case performance, and meaningful empirical analysis must include critical-region testing.

This paper describes a tool called AlgoLab that is designed to provide high-level support for the development and comparison of algorithms. It encapsulates a set of parameters that define a subset of the problem space, a random generator for instances of the problem, a set of algorithms to be tested, and a visualization panel that captures and charts the results of an experiment. In its current version, AlgoLab is designed around the PARTITION problem. The code is easily adaptable to other problems, and future versions of the software will provide support for common decision and optimization problems over sets of integers, graphs, and Boolean expressions.

2 The PARTITION Problem

The PARTITION problem [2, p. 97] is defined as follows: given a set S of positive integers, determine whether S can be partitioned into two subsets that have the same sum. Alternately, the problem can be defined as a special case of the SUBSET SUM problem: given a set of positive integers $S = \{x_1, x_2, \ldots x_n\}$, determine whether S has a subset whose sum is $(x_1 + x_2 + \ldots + x_n)/2$. Both PARTITION and SUBSET SUM are NP-complete. There is an obvious exponential-time exhaustive search algorithm: successively generate all subsets and compute their sums. If S has n elements, it has 2^n subsets, and the exhaustive search algorithm is $O(p(n)\cdot 2^n)$ for some polynomial function $p(n)$.

There is another approach to the problem that is not so obviously exponential: for each positive sum k less than or equal to the target sum T, determine whether S has a subset with sum k. This can be accomplished by keeping a list of sums L, where L initially contains only the sum 0. For each x in S, and each k in L, the list is progressively extended by adding $x+k$ to the list (if $x+k < T$). Since the length of list cannot exceed T, the number of steps for this algorithm is proportional to $n\cdot T$. Any non-trivial target sum T must be less than the sum of elements in S, so the length of the list never exceeds $n\cdot m$, where m is the maximum number in S. Throw in the observation that n must be less than or equal to m, and the algorithm can be classified as $O(p(n)\cdot m^2)$. Either way, it looks

polynomial, not exponential. And it is in fact polynomial if m is $O(n)$. But the hard instances of the problem have $m = O(2^n)$. In this case, the number of bits in the representation of the set is $O(n^2)$, and the number of steps in the algorithm is $O((2^n)^2)$, which is indeed exponential (see [4] for a discussion of the complexity of PARTITION).

So which approach is better? If m is $O(n)$, we should generate all the sums. If m is $O(2^n)$, however, we may be better off to generate all the subsets. This motivates an empirical comparison of the two algorithms. But at the same time, it has motivated the development of a general tool to support comparison of decision algorithms. AlgoLab is built around the PARTITION problem, but it can easily be adapted to the empirical study of other algorithms.

3 AlgoLab Features

Algolab provides automatic generation of problem instances driven by two integer parameters. This is sufficient to cover a wide variety of problems. For problems involving sets of positive integers, the parameters specify the set size and the maximum value. For graphs, the parameters specify the number of nodes and the number of edges. For Boolean expressions, the parameters specify the number of variables and the number of clauses. Given the parameter values, a random number generator is used to construct the object with components from a uniform distribution.

The AlgoLab interface allows the user to design an experiment in which one or more algorithms are run on a set of problem instances. The instance parameters can be specified to be fixed or variable. If the parameters are both fixed, the algorithm is run on multiple instances of the same size. If either parameter is variable, the problem instances cover a range of sizes. It is possible to have one parameter fixed and the other variable, or to have both parameters variable with one dependent upon the other. For example, we could run a PARTITION algorithm on sets of integers where the size of the set ranges from 10 to 20, and where the maximum number in each set is a function of its size. The interface has an expression parser that allows the user to specify an expression using common arithmetic operators and the names of the parameters.

For each experiment, the user selects one or more algorithms from a list, specifies the instance parameters, and indicates how many replications to run for each kind of instance. The output from a run can be Boolean (for decision problems) or numeric (for optimization problems). The output is plotted on a chart whose x-axis represents the instance size, and whose y-axis is either a percentage of *true* vs. *false* or a numeric value. The user can also choose to plot step counts for either decision or optimization algorithms.

When the user chooses Boolean output, a point series is generated for percent *true* and percent *false*. If more than one algorithm have been selected, the selected algorithms are

run on the same instances, and their results are compared for consistency. This gives the user a way to validate a new version of an algorithm. A new algorithm can be run in parallel with a reliable one, and any inconsistencies in the decision results will be reported. Only one set of results is displayed on the chart, representing the results of all selected algorithms. The instance sizes determine the *x*-axis scale, and the *y*-axis represents the percent of instances for which the algorithms returned *true* or *false*. A green line is drawn to connect the *true* points, and a red line connects the *false* points. Depending on what range of instances is specified, the user may observe the "crossover" phenomenon, where the green and red lines cross. This crossover point characterizes the problem size where the probability of a *true* decision is equal to the probability of a *false* decision. Algorithms that employ exhaustive search with heuristics will typically show their worst performance in the region around this crossover point, and threshold behavior is common as the *x*-values approach this point.

When the user runs a numeric step count series, the selected algorithms are run on the same set of instances and the step counts are plotted on the *y*-axis, with a separate series for each algorithm. This provides immediate performance comparisons for variations of an algorithm over specified regions of the problem space. Data points are connected with lines of different colors for each algorithm. Each point is the average of all replications for the corresponding instance size (the average of all *y*-values for points with the same *x*-value). In addition to the average point series, the result of every instance is displayed as a green or red scatter point for decision algorithms, depending on whether the decision was *true* or *false*. This point display gives the user a sense of the variation in step counts and of the relative difficulty of computing *true* decisions vs. *false* decisions.

The chart panel provides automatic scaling for both the *x*-axis and the *y*-axis. The *x*-axis is scaled to show the range of the instance parameter that the user has specified to be variable and independent. The *y*-axis is dynamically scaled as data points are generated. Because the range of step counts in a numeric series can be very large, the user can manually adjust the *y*-axis by dragging the scale. Dragging the vertical scale upward or downward below the middle of the chart will cause the lowest value on the *y*-axis to decrease or increase, and dragging the vertical scale above the middle of the chart will change the highest value on the *y*-axis. This feature allows the user to "zoom in" to areas of interest in the chart.

The user also has the option to toggle between logarithmic and linear scales for both axes. By default, tick marks are displayed on the y-scale, but numeric values are not. The user can see a display of the (*x*, *y*) values of any point on the chart by moving the mouse over the point with the right mouse button depressed.

4 Experimenting with PARTITION

As mentioned in Section 2 above, there are two distinct approaches for solving the PARTITION problem. One approach enumerates and tests subsets, while the other

approach enumerates possible sums. We will refer to these algorithms as *SubSearch* and *SumSearch*, respectively. We can use AlgoLab to validate the consistency of these two algorithms, to compare their performance, and to discover the critical region of the problem space for PARTITION.

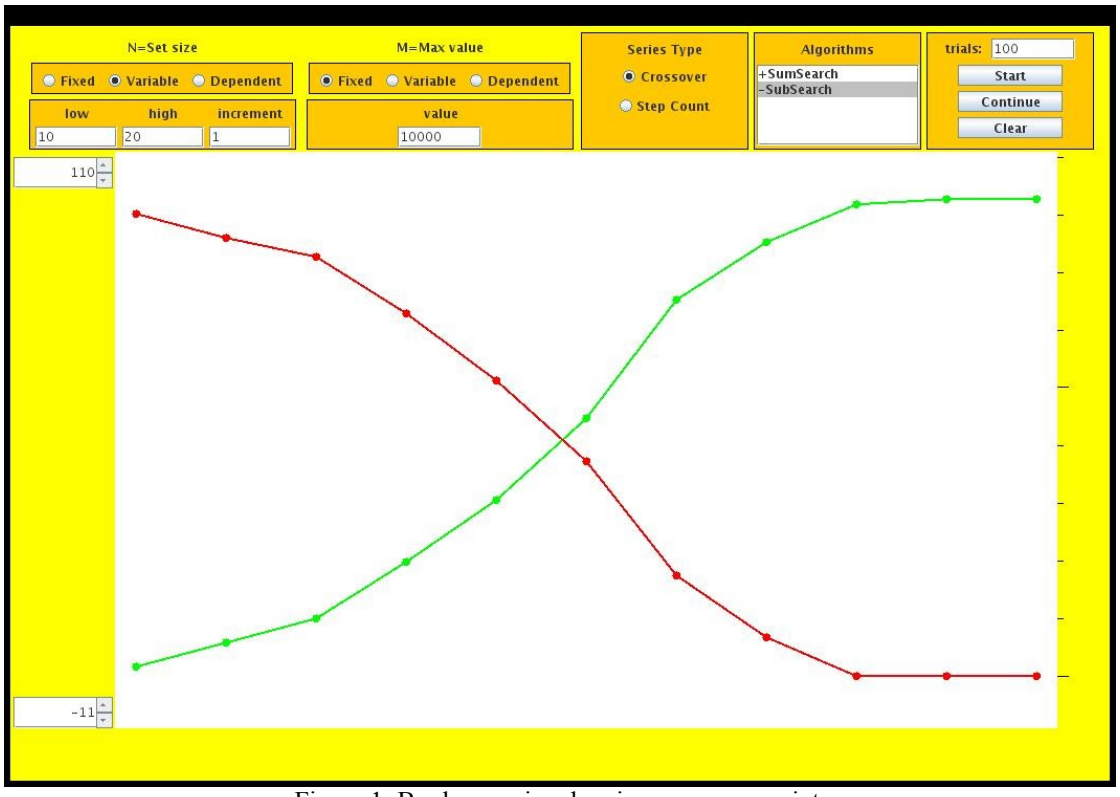

Figure 1: Boolean series showing crossover point.

With combinatorial problems, the first few experiments should be probes to determine what range of instance sizes will be feasible. Exponential-time algorithms do not scale up well, so it's good to know where the threshold between a few seconds per instance to several minutes or hours per instance will be found. The *SubSearch* algorithm for PARTITION literally tries all subsets with few time-saving heuristics, so we cannot expect to run it on sets of 100 numbers. Waiting for execution of $O(2^{100})$ steps is not feasible. So we expect to stick to sets of size 50 or less. Anticipating that the *SumSearch* algorithm will be faster than *SubSearch*, we start by running *SumSearch* on sets of size 10 to 50 with an increment of 5, each set with a fixed maximum integer value of 1,000,000. We set the number of trials per instance size to 20, and we select the numeric step count series. This experiment takes about 10 minutes, and we observe a maximum step count of more than 25,000,000. We repeat the experiment, changing the maximum number in each set from 1,000,000 to 1,000. We expect that the algorithm that enumerates sums will show significantly better performance if the target sum is smaller. This experiment takes only a few seconds, with a maximum step count of about 77,000. So we have determined that we can feasibly test set sizes of up to 50 numbers, as long as the maximum numbers in the sets don't get too big.

Next we decide to look for a crossover point. We repeat the previous experiment with Boolean output instead of step counts. We get immediate results, and we see a crossover in the lowest *x* range: the percent *true* jumps from about 20% for $n = 10$ to 100% for $n = 15$, and it stays at 100% for the remainder of the *n*-values. When the maximum value in the set is 1,000, crossover occurs almost immediately as the set size grows beyond 10. To get a closer look, we change the set sizes to range from 10 to 20, and try the experiment with maximum values of $m = 1000$, 10,000, and 100,000, while raising the number of trials to 100 to smooth out the curves. We see that for maximum $m = 1000$, crossover occurs just beyond $n = 11$, for $m =$ 10,000 it occurs just before $n = 15$ (see Figure 1), and for maximum $m = 100{,}000$ it occurs at $n = 18$. We decide to try again with $m = 1{,}000{,}000$, and this time we find no crossover – the percent *true* rises from 0 to about 20% at $n = 20$.

Figure 2: Boolean series in critical region.

We now have enough data to take a guess at the relationship between *n* and *m* that defines the critical region for the problem (see [1]). The first three crossovers occurred when *m* was about 2^{n-1}. So we predict that crossover for $m = 1{,}000{,}000$ will occur when *n* is about 21, and we change our set sizes to range from 15 to 25 to test our hypothesis. We have to wait a minute or two for this result, but as predicted, crossover occurs between $n = 21$ and $n = 22$. We can further confirm our hypothesis by making the maximum value *m* dependent on the set size *n*. To accomplish this, we redefine the *m* parameter to be dependent, and type in the formula 2^(*n*-1). This should cause the

problem instances to stay in the critical region as *n* grows, with the result that percent *true* will remain fixed at about 50%. We also expect this to take substantially more time, so we lower the number of trials per instance to 20. After a minute or so, we see that the red and green lines, indeed, do not cross. But the percent *true* looks closer to 15% than 50%, and the line is jagged (see Figure 2). We run more trials to smooth out the line, and confirm that % *true* is steady at about 15%. We have found the critical region, but we would have to scale up our formula for *m* slightly to find the middle of the critical region.

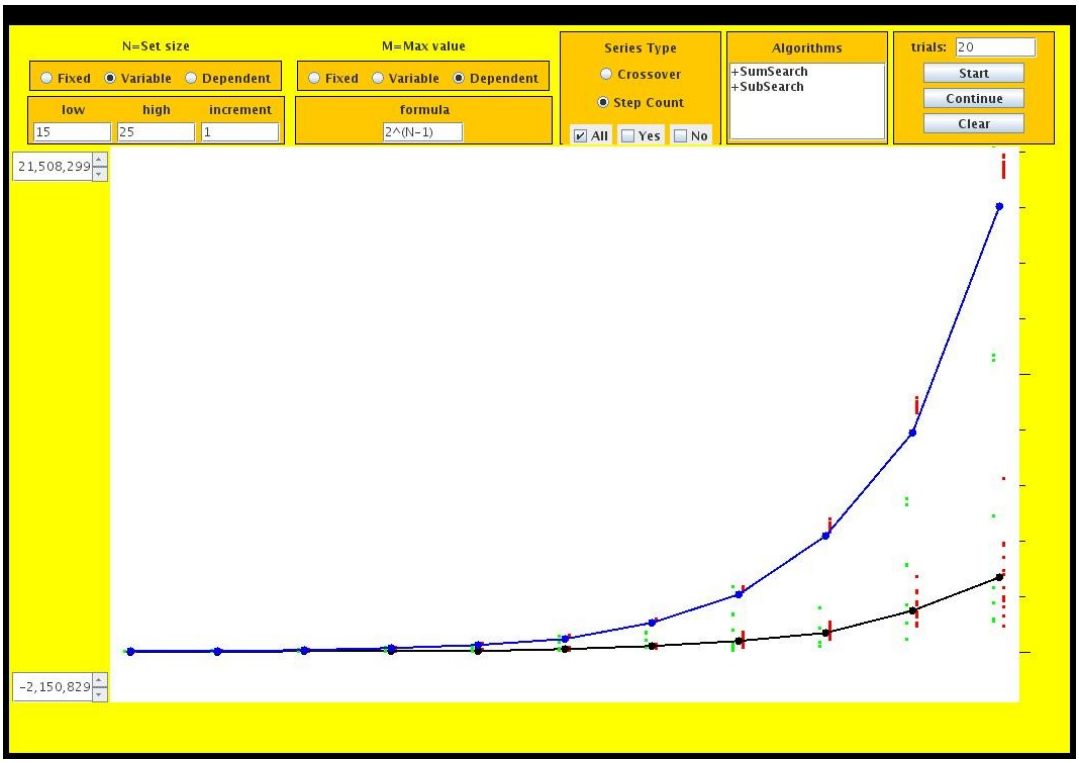

Figure 3: Algorithm comparison showing exponential complexity in critical region.

Now that we know where the critical region is, we can compare the performance of our two algorithms. We select both *SumSearch* and *SubSearch* from the algorithm list and change the series back to step counts. This experiment takes a bit longer, but after 10 minutes or so, we see that *SubSearch* has substantially higher step counts than *SumSearch*, with an average step count over 20,000,000 for $n = 25$ (see Figure 3). The curve for *SubSearch* looks classically exponential, and when we zoom in to focus on the *SumSearch* curve, it also looks exponential, with an average step count over 3,000,000 for $n = 25$. We've learned something about the PARTITION problem, we have confirmed that enumerating sums is a better approach than enumerating subsets, and we are confident that we know how to compare PARTITION algorithms over a meaningful range of problem instances.

5 Conclusion and Further Development

AlgoLab provides a tool for rapid experimentation with algorithms. It can effectively be used to find critical regions in the problem space, validate the correctness of algorithms, and compare the performance of algorithms. While the current version encapsulates the PARTITION problem, the code is abstractly designed to be easily adaptable to a variety of decision and optimization problems over integer sets, graphs, and Boolean expressions.

Future versions of AlgoLab will allow instance parameters, instance generators, and algorithms to be loaded dynamically, enabling the user to switch problems without modifying the source code. Archiving and output functions for the results of experiments will be added, as well as an interface for viewing and manually solving individual problem instances (such as VisiDLL for SATISFIABILITY as described in [3]). We envision a robust, time-saving, and flexible tool for numerous applications in both research and instruction.

References

[1] B. Hayes, "The Easiest Hard Problem," *American Scientist* 90(2), p. 116, March-April 2002.

[2] R. Karp, "Reducibility Among Combinatorial Problems," in *Complexity and Computer Computations*, ed. R. E. Miller and J. W. Thatcher, pp. 85-103, Plenum Press, New York (1972).

[3] O'Neil, T. E., "Classics Illustrated: A Visualization Tool for Theorem-Proving Procedures," *Proceedings of the 38*th *Midwest Instruction and Computing Symposium* (MICS '05), Eau Claire, Wisconsin, April 2005.

[4] R. Stearns and H. Hunt, "Power Indices and Easier Hard Problems," *Mathematical Systems Theory* 23 (1990), pp. 209-225.