

Monitoring an Algorithm's Execution

DAVID A. BERQUE

MARK K. GOLDBERG

ABSTRACT. Many software systems for Discrete Mathematics incorporate routines that compute NP-hard functions. In spite of their worst-case exponential running time, there are often a wide range of inputs for which these routines run in interactive time. It is generally difficult for a user to distinguish, a priori, the “easy” inputs from those that are “hard”. A *percent-done progress indicator* is a software tool that allows a user to monitor the percentage of a computation that has been completed. Such indicators have been implemented for linear algorithms (e.g., file transfer programs). This paper introduces a paradigm which we call *dynamic bound evaluation* for monitoring the progress of a wide class of recursive algorithms that need not be linear. Further, empirical results are presented that illustrate the effectiveness of the technique as it has been implemented in the *SetPlayer* system for symbolic set manipulation.

1. Introduction and motivation

Many software systems incorporate algorithms that execute very quickly for some inputs while requiring an exceedingly long period of time to terminate for others. This can be unsettling for users, particularly if a good estimate of the total computation time for a particular input is not available. In [9] Myers describes a *percent-done progress indicator* as a software tool that allows a user to monitor the percentage of a computation that has been completed. He notes, however, that relatively few software systems support these indicators. One reason for the scarcity of progress indicators in current software systems is the difficulty in calculating the percentage of a computation that has been completed. For programs that make a fixed number of linear passes over their

1991 Mathematics Subject Classification. Primary 68N99. Secondary 68R05, 68Q40.

The first author was supported in part by the NSA under grant MDA904-90-H-4027. The second author was supported in part by the NSA under grant MDA904-90-H-4027 and by the NSF under grants IRI-8900511 and CDA-8805910. Part of this work was done while the second author was visiting Los Alamos National Laboratory.

This paper is in final form and no version of it will be submitted for publication elsewhere.

data (e.g., file transfer programs) this is a simple calculation; however, for most other types of programs such a calculation can be quite difficult. For this reason most percent-done progress indicators have been found in systems that are driven primarily by linear algorithms (for example the *Macterminal* program that runs on *Macintosh* computers) [9].

Of course, many algorithms do not fall into the “linear pass over data” paradigm. For example, many of the increasingly popular symbolic computation packages (e.g., Maple, Mathematica) rely heavily on non-linear algorithms [6, 12]. Software systems driven by non-linear algorithms are likely to exhibit a greater variation in running time on different inputs than those driven by linear algorithms. Therefore, the need for a tool that can monitor the progress which such a software system has made toward responding to a command is especially acute. This need has been recognized by Caviness and Boyle in the report entitled *Future Directions for Research in Symbolic Computation* where they write that “Carefully designed monitoring aids for software and hardware would be useful in measuring space, time, and progress toward a solution” [5].

The need for a technique that monitors the progress that a software system has made toward responding to a user’s command is especially strong in many software systems that deal with Discrete Mathematics. Since many combinatorial problems are NP-hard, software systems for Discrete Mathematics often incorporate algorithms which have exponential worst case running time. In spite of this, however, there are often a wide range of inputs on which these algorithms perform quite well. Furthermore, it is generally difficult for a user to distinguish, a priori, those inputs for which the algorithm will run quickly from those which will require impractical amounts of time. Knuth illustrates the problems which can arise from these characteristics when he writes that he “...once waited all night for the output ... only to discover that the answers would not be forthcoming for about 10^6 centuries” [8].

The authors’ own motivation for developing a technique for monitoring the progress of an algorithm arises from our experiences in designing and implementing the *SetPlayer* system. The goal of this system is to provide a tool for students and researchers in Discrete Mathematics much as Mathematica provides a tool for students and researchers of other areas of mathematics (see [4] for a complete description of *SetPlayer*). In particular, the system is designed to manipulate sets described symbolically, with the help of the common mathematical expressions 2^S (the power set of S) and $\binom{S}{k}$ (the set of all k -subsets of S).

SetPlayer supports more than sixty commands, many of which are driven by algorithms that run in polynomial time (all of the polynomials have low degree, and the algorithms all have small implementation constants). However, several of *SetPlayer*’s commands depend on algorithms that compute the cardinality of the sets $2^{A_1} \cup 2^{A_2} \dots 2^{A_m}$ and $\binom{A_1}{k} \cup \binom{A_2}{k} \cup \dots \cup \binom{A_m}{k}$, given the sets $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$ ($m \geq 0$) and the integer $k \geq 0$. The problems associated with computing these cardinalities are called *Card* and *Card.k* respectively.

SetPlayer computes *Card* and *Card.k* by computing the functions *Trs* and *Trs.k* defined as follows. The input to *Trs* (resp. *Trs.k*) is a set U and a collection of sets $\mathcal{A} = \{A_1, A_2, \dots, A_m\}$ for $m \geq 0$ (resp. the same as for *Trs* and an integer $k \geq 0$). The output is the number of subsets (resp. subsets of size k) of U that have non-empty intersections with every $A_i \in \mathcal{A}$ ($1 \leq i \leq m$). The following identities establish that *Card* (respectively, *Card.k*) is computationally equivalent to *Trs* (respectively, *Trs.k*). Let $U = \bigcup_{i=1}^m A_i$, $\overline{A}_i = U - A_i$ ($i = 1, \dots, m$), and $\overline{\mathcal{A}} = \{\overline{A}_1, \dots, \overline{A}_m\}$; then

$$\text{Card}(\mathcal{A}) = 2^{|U|} - \text{Trs}(U, \overline{\mathcal{A}}); \quad \text{Card.k}(\mathcal{A}) = \binom{|U|}{k} - \text{Trs.k}(U, \overline{\mathcal{A}}).$$

Obviously, the special case of *Trs* that occurs when all the input sets have cardinality 2 is the *Monotone 2-SAT Problem*, which is proved to be #P-complete in [11]. Note also that if $G = (V, E)$ is a graph then computing *Trs.k*(V, E) is identical to computing the number of vertex covers of size k in G , which is a well-known #P-complete problem [7].

The details of the algorithm that *SetPlayer* uses to compute *Trs* are beyond the scope of this paper (the interested reader is referred to [3]). However, it is useful to point out that the algorithm follows a basic strategy that is employed by many recursive algorithms. First, a *simplification stage* checks to see if the problem instance can be replaced by a simpler instance or can be solved directly. If the simplification stage is not able to solve the problem instance, then a *recursive stage* of the algorithm divides this instance into two smaller subproblems. The answer to the original instance is then the sum of the answers to these subproblems.

Experiments show that for many collections \mathcal{A} with identical “simple” parameters, the running time of our algorithms for computing *Card* and *Card.k* can be drastically different. Table 1 shows the time required to compute *Card* on three collections $\mathcal{A}_1, \mathcal{A}_2, \mathcal{A}_3$. Each of the collections contains 120 subsets of size 108 of a set of size 110. Moreover, all three collections have the same degree sequence of elements. The latter means that there is one-to-one correspondence between the elements of any pair of collections such that the corresponding elements belong to the same number of sets. The computations were performed using a Sun SPARCstation-1.

<i>Input</i>	<i>Time Required</i>
\mathcal{A}_1	4.5 hours
\mathcal{A}_2	19.7 seconds
\mathcal{A}_3	4.8 seconds

Table 1: Time Required to Compute the *Card* Function

Since the standard quantitative measures of the collections are identical, it would be very difficult for a user to predict that one of the collections will require

significantly more processing time than the others. However, with the help of a monitoring tool, a user might be able to make this determination after a small percentage of the computation had completed. The user could then decide to continue the computation interactively, to finish the computation as a batch job, or to abort the computation entirely.

2. Dynamic bound evaluation

2.1. Background. In [8] Knuth discusses a Monte-Carlo technique for *estimating* the run-time of backtrack programs. The approach requires carrying out a separate estimation procedure *before* starting the primary computation. The estimation procedure provides a static pre-estimate on the required computation time. This differs considerably from the dynamic output provided by the percent-done progress indicators described above.

In the remainder of this section we present a paradigm for dynamically monitoring the progress of a wide class of recursive algorithms. The paradigm is presented from an practical (rather than mathematically rigorous) point of view. In the final section of the paper we present empirical results that illustrate the effectiveness of the technique as it has been implemented in the *SetPlayer* system.

2.2. The Basic Scheme. The dynamic bound evaluation technique is applicable to a wide class of recursive algorithms that work by building a tree of subproblems. We will refer to this tree as a *computation tree*. The root of a computation tree is labeled by the original input problem and the leaves of the tree are labeled by problems that can be solved directly without further subdivision. We will refer to the number of nodes of a computation tree as the *size* of the tree.

Below we review some of the properties that are common to algorithms belonging to this class.

- The algorithm starts with an initial problem instance I as the “current” problem.
- At each step, the current problem is either solved directly or divided into one or more subproblems. One of these subproblems becomes the current problem. The remaining subproblems are stored for later processing.
- When a problem S is divided into several subproblems, the solution to problem S is some combination of the solutions to each of its subproblems.
- The individual subproblems are solved independently of each other (they do not communicate in any way). Thus, the order in which they are evaluated is not important.

In addition to the properties described above, the dynamic bound evaluation technique requires the existence of at least one of the following functions.

- The function $ub(S)$ which gives an upper bound on the size of the computation tree generated by executing the algorithm on problem S . For sufficiently small instances of S the upper bound should be strict.
- The function $lb(S)$ which gives a lower bound on the size of the computation tree generated by executing the algorithm on problem S . For sufficiently small instances of S the lower bound should be strict.

The effectiveness of the technique will be enhanced if the following conditions are met.

- Both (rather than just one) of the functions $ub(S)$, and $lb(S)$ are available.
- The functions $ub(S)$ and $lb(S)$ are based on some parameter n , of S , that tends to decrease as problems are divided into smaller problems. Further, the value of $ub(S)$ decreases as n decreases. Similarly, the value of $lb(S)$ increases as n decreases.

Note that the running time of this type of algorithm on a particular input is roughly proportional to the number of nodes in the associated computation tree. Therefore, if at some early stage of the computation we can compute (1) the number of nodes of the computation tree that have already been constructed, and (2) upper and lower bounds on the number of nodes that the completed computation tree will eventually contain, then we can determine the percentage of the computation that has been completed in the best and worst cases. Computing (1) is a routine task; the program can simply count the nodes as they are completed. Computing (2) is the subject of the remainder of this section.

Suppose an algorithm which meets the conditions described above has been implemented to solve a particular problem P without regard to monitoring the progress of the computation. A common technique for managing the partially completed subproblems is to store them on a programmer controlled stack (rather than letting the machine manage the recursion automatically on the run-time stack) and we assume that this technique has been used. In what follows, we explain how such an implementation can be easily modified so that its progress can be monitored using the functions $ub(S)$ and $lb(S)$ to dynamically compute upper and lower bounds on the size of the computation tree that represents the computation at hand. The main idea behind the technique we present is that even if the functions $ub(S)$ and $lb(S)$ do not give tight bounds, the values of these functions will converge as the sizes of the uncompleted subproblems decrease.

Suppose at some moment during the computation the current problem is S , the stack contains subproblems S_1, S_2, \dots, S_n , and i nodes of the computation tree have already been constructed. From the definitions of the functions $ub(S)$

and $lb(S)$, it is clear that an upper bound, u , on the total size of the computation tree is given by

$$u = i + ub(S) + \sum_{i=1}^n ub(S_i)$$

and similarly, a lower bound, l , on the total size of the computation tree is given by

$$l = i + lb(S) + \sum_{i=1}^n lb(S_i).$$

Since i gives the number of nodes of the tree that have already been constructed, we can easily compute the percentage of the tree that has been constructed in the best and worst cases.

2.3. Partial versus Complete Percent Done Feedback. The dynamic bound evaluation technique described above can produce two distinct types of percent-done information. During early stages of the computation, it is likely that the lower bound on the size of the computation tree will be significantly smaller than the upper bound on this size. In this case the technique only computes the percentage of the computation tree that has been constructed in the *best* and *worst* cases. We refer to this type of percent-done feedback as *partial percent-done feedback*. However, as the computation proceeds, the remaining subproblems become smaller and the upper and lower bounds converge. Once the bounds become equal, the dynamic bound evaluation technique will compute the exact percentage of the computation tree that has been constructed. We refer to this type of percent-done feedback as *complete percent-done feedback*. The empirical results presented in the final section of this paper indicate that the transition from partial to complete percent-done feedback generally occurs at a reasonably early stage of the computation.

Although it is clear that complete percent-done feedback is more useful to the user than partial percent-done feedback, there are many situations in which even partial percent-done feedback can be extremely helpful. For example, suppose a user issues a command and, after five seconds of computation time have elapsed, the percent-done progress indicator reports that in the best case the computation is 90% complete while in the worst case it is only 50% complete. Although this is only partial percent-done feedback, the user can determine that even in the worst case only a few more seconds of computation time will be required. The user, thus assured that this is not a computation that will require hours of computation time, will probably decide to continue the computation.

Suppose now that the user requests the system to carry out a second command. This time, after ten minutes of computation time has elapsed, the percent-done progress indicator reports that in the best case the computation is 5% complete while in the worst case it is 1% complete. The user might well decide to continue the computation in the background (or to abort it entirely) since *even in the best case* several hours of computation time will be required to complete the task.

2.4. An Improvement to the Basic Scheme. The transition from partial to complete percent-done feedback occurs when the upper and lower bounds on the size of the computation tree become equal. In this section we consider a variation of the basic dynamic bound evaluation technique that increases the rate at which these bounds converge.

Suppose that at some moment during the computation the stack contains uncompleted subproblems S_1, S_2, \dots, S_n . For a particular subproblem S_i we can compute the values of $ub(S_i)$ and $lb(S_i)$. We refer to the quantity $ub(S_i) - lb(S_i)$ as the *uncertainty* of subproblem S_i . The smaller the uncertainty value for a particular subproblem is, the more complete information we have about the size of the computation tree that will result from applying our algorithm to that subproblem.

When the uncompleted subproblems are stored in a stack they are subdivided in a last-in, first-out fashion. If we break away from this purely LIFO ordering, we can process the problems based on their uncertainty value. In particular, by selecting problems with high uncertainty values for early subdivision we can increase the rate at which the upper and lower bounds on the size of the associated computation tree become equal.

As an initial attempt at accomplishing this goal we can try storing the uncompleted subproblems in a priority queue. Before enqueueing a subproblem S , the priority of S is computed according to the formula $pri(S) = 1/(ub(S) - lb(S))$. Using the standard convention that lower numeric priority values have higher priority, it follows that subproblems with greatest uncertainty will be given highest priority and will be subdivided first. Thus, the upper and lower bounds can be expected to become equal at an earlier stage of the computation.

When using priority queues to manage the collection of uncompleted problems, it is possible for the size of the priority queue to become exponentially large with respect to the size of the original input problem. In addition to the obvious storage problems this creates, the time needed to enqueue and dequeue subproblems can become unacceptably long. To eliminate these difficulties while retaining the improved convergence rate that the priority queue allows, we use *both* a priority queue and a stack to manage the uncompleted subproblems. The priority queue is used as the primary storage mechanism; however, the stack is used to temporarily handle overflow should the number of problems in the priority queue exceed some predetermined constant C_p .

If the value of C_p is increased, more memory is required to store the uncompleted subproblems and more time is needed to enqueue and dequeue individual subproblems. On the other hand, the upper and lower bounds may converge at an earlier stage of the computation.

3. Empirical results

Earlier sections of this paper have shown situations in which partial percent-done feedback can be extremely useful to the user. It is clear that this partial percent-done feedback becomes more useful as the computation proceeds and the difference between the lower and upper bounds on the size of the completed computation tree decreases. When this difference reaches zero, the partial percent-done feedback becomes complete-done feedback and is of maximum value to the user.

In this section we present data illustrating the rate at which the upper and lower bounds on the computation tree size may be expected to converge. The data presented in this section were obtained using the dynamic bound evaluation technique as it has been implemented to monitor the progress of *SetPlayer's* command for computing *Trs*.

We have developed functions $ub(H)$ and $lb(H)$ that evaluate to strict (as opposed to asymptotic) upper and lower bounds on the size of the computation tree that will be generated by applying *SetPlayer's* algorithm for computing *Trs* to a problem instance H . The derivations of these functions are based on modeling the best and worst-case behavior of the algorithm with recurrence relations, and then finding closed-form solutions to these relations. The interested reader is referred to [2] for a complete description of *SetPlayer's* algorithm for computing *Trs*, as well as a discussion of the recurrence relations that model the algorithms and the solutions to these recurrences.

The three collections of sets $\mathcal{A}_1, \mathcal{A}_2$, and \mathcal{A}_3 mentioned in the introduction were used to develop three inputs to *Trs* with identical simple parameters. These inputs are graphs $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 , each with 110 vertices, 120 edges, and identical degree sequences. The running times given in Table 1 for computing *Card* on $\mathcal{A}_1, \mathcal{A}_2$, and \mathcal{A}_3 are also applicable for computing *Trs* on $\mathcal{G}_1, \mathcal{G}_2$, and \mathcal{G}_3 respectively. Tables 2 and 3 illustrate that the monitoring routine was able to detect the drastic variation in time required to compute *Trs* on the graphs \mathcal{G}_1 and \mathcal{G}_2 . Table 2 shows that after only one node of the computation tree had been constructed, the upper and lower bounds on the size of the completed tree were both 2,097,151. Since the values were equal, the percent-done progress indicator was able to give complete (rather than partial) percent-done feedback even at the earliest stages of this computation.

<i>Nodes Completed</i>	<i>Lower Bound</i>	<i>Upper Bound</i>
1	2,097,151	2,097,151

Table 2: Percent-Done Feedback for $Trs(\mathcal{G}_1)$

For the case of graph \mathcal{G}_2 , the convergence was somewhat slower as illustrated in Table 3. This table shows that after 127 nodes of the computation tree had been constructed, the upper and lower bounds on the size of the completed

<i>Nodes Completed</i>	<i>Lower Bound</i>	<i>Upper Bound</i>
1	57	786,431
50	293	1,951
127	1,535	1,535

Table 3: Percent-Done Feedback for $Trs(\mathcal{G}_2)$

computation tree had converged to the value 1,535. Thus, after approximately 8% of the computation had completed, the percent-done progress indicator was able to make the transition from partial to complete percent-done feedback.

The previous section of this paper described how the effectiveness of the dynamic bound evaluation technique is altered by the value of C_p which controls the number of subproblems that the priority queue can hold before storing additional subproblems on the stack. The data shown in Tables 2 and 3 were gathered using a value of $C_p = 2,200$.

Table 4 contains data illustrating the effect of varying the value of the constant C_p . In particular, the table presents data obtained by computing Trs of a randomly generated 3-regular graph on 60 vertices with various values of C_p .

C_p	$lb = ub/10$	$lb = ub/2$	$lb = 9ub/10$	$lb = ub$
1,200	91.0%	95.4%	97.3%	99.3%
2,400	14.2%	59.7%	80.8%	88.1%
3,600	4.3%	40.4%	67.5%	75.9%
4,800	4.3%	23.1%	50.3%	60.8%
6,000	4.3%	12.4%	35.1%	50.0%
7,200	4.3%	12.4%	24.1%	38.6%
8,400	4.3%	12.4%	22.2%	36.2%
9,600	4.3%	12.4%	22.2%	36.2%

Table 4: Convergence of Upper and Lower Bounds when Computing Trs on a 3-Regular Graph on 60 Vertices Using Various Values of C_p

The entries in Table 4 indicate the percentage of the computation that was actually complete when the dynamic bound evaluation technique produced upper and lower bounds satisfying the conditions given at the top of the corresponding column. For example, the entry for the row labeled 3,600 and the column labeled $lb = ub/10$ is 4.3%. This means that when 4.3% of the computation had actually been completed, the lower bound on the size of the completed computation tree was one-tenth of the upper bound (in this case the size of the completed computation tree was known to within an order of magnitude). The right-most column of the table is labeled by the equation $lb = ub$. Therefore, the values in this column indicate the percentage of the computation that was

complete when the percent-done progress indicator made the transition from providing partial to complete percent-done feedback.

As we would expect, the rate of convergence for the upper and lower bounds increases with the value of C_p . Each time $Trs(\mathcal{G})$ is computed, *SetPlayer* automatically adjusts the value of C_p based on the size of graph \mathcal{G} . The system attempts to allocate enough memory to ensure a reasonable convergence rate, but not so much memory that the system's performance degrades. However, the user can alter the value of C_p set by the system if convergence rate appears to be too slow.

We conclude the paper with a final example illustrating the rate at which the upper and lower bounds on the size of the computation tree converge. Table 5 presents data obtained by computing Trs of a randomly generated graph with edge probability of 0.1 on 80 vertices for various values of C_p .

C_p	$lb = ub/10$	$lb = ub/2$	$lb = 9ub/10$	$lb = ub$
4,000	74.0%	85.1%	91.5%	96.7%
8,000	55.0%	75.4%	86.8%	93.6%
12,000	37.6%	66.5%	82.2%	90.2%
16,000	25.1%	59.3%	78.1%	87.2%
20,000	24.0%	58.6%	77.7%	86.9%
24,000	8.2%	47.1%	69.6%	81.0%
28,000	8.2%	42.2%	65.6%	77.8%
32,000	8.2%	36.9%	61.2%	74.3%
36,000	8.2%	33.4%	57.3%	71.3%
40,000	8.2%	29.0%	52.9%	67.6%
44,000	8.2%	24.4%	50.0%	65.0%
48,000	8.2%	20.2%	47.7%	62.7%
52,000	8.2%	19.3%	44.3%	59.3%
56,000	8.2%	19.3%	40.3%	55.3%
60,000	8.2%	19.3%	39.4%	54.4%
64,000	8.2%	19.3%	39.4%	54.4%

Table 5: Convergence of Upper and Lower Bounds when Computing Trs on a Graph with Edge Probability 0.1 on 80 Vertices Using Various Values of C_p

References

- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.
- [2] David Berque, *Implicit Set Manipulation: Theory and Practice*. PhD thesis, Rensselaer Polytechnic Institute, 1991. UMI Dissertation Service Publishers, Ann Arbor, Michigan.

- [3] David Berque, Ron Cecchini, Mark Goldberg, and Reid Rivenburg, The *SetPlayer* System for Symbolic Computation on Power Sets. *J. of Symbolic Computation*, **14**, (1992), pp.645 - 662.
- [4] ———, The *SetPlayer* System: An Overview and a User Manual. Technical Report 91-17, Rensselaer Polytechnic Institute, Department of Computer Science, 1991.
- [5] B. F. Caviness and A. Boyle (editors), Future Directions for Research in Symbolic Computation. Report of a Workshop on Symbolic and Algebraic Computation, Society of Industrial and Applied Mathematics, August 1989.
- [6] Bruce W. Char, *First Leaves: A Tutorial Introduction to Maple*. WATCOM Publications, Waterloo, Ontario, 1988.
- [7] M. R. Garey, and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W.M. Freeman and Co, 1979.
- [8] Donald E. Knuth, Estimating the Efficiency of Backtrack Programs. *Mathematics of Computation*, **29**(129):121–136, January 1975.
- [9] Brad A. Myers, The Importance of Percent-done Progress Indicators for Computer-human Interfaces. In *CHI '85 Proceedings*, pages 11–17, April 1985.
- [10] ———, Using Percent-done Indicators to Enhance User Interfaces. In *Graphics Interface '85*, pages 167–170, May 1985.
- [11] Leslie G. Valiant, The Complexity of Enumeration and Reliability Problems. *SIAM Journal of Computing*, **8**(3):410–421, 1979.
- [12] Stephen Wolfram, *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley Publishing Company, Redwood City, California, 1988.

David A. Berque
 DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE, DEPAUW UNIVERSITY, GREENCASTLE, INDIANA 46135
E-mail address: dberque@depauw.bitnet

Mark K. Goldberg
 DEPARTMENT OF COMPUTER SCIENCE, RENSSELAER POLYTECHNIC INSTITUTE, TROY, NEW YORK 12180
E-mail address: goldberg@cs.rpi.edu