IMPLEMENTING PROGRESS INDICATORS FOR RECURSIVE ALGORITHMS

Dave A. Berque[*] *Jeffrey A. Edmonds*[†]

Math & Computer Science Department
DePauw University
Greencastle, IN 46135
dberque@depauw.bitnet
jedmonds@depauw.bitnet

Mark K. Goldberg[‡]

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180
goldberg@cs.rpi.edu

ABSTRACT
A percent-done progress indicator is a software tool that allows a user to monitor the progress that a software system has made toward responding to a command. Formal experiments have demonstrated that users prefer software systems that incorporate percent-done progress indicators over those that do not. Obviously, a system must be able to calculate (or at least estimate) the percentage of a computation that has been completed in order to support a percent-done progress indicator. This is generally straightforward for programs that make a fixed number of linear passes over their input and then terminate (e.g., file transfer programs). Most previous percent-done progress indicators have been limited to such programs. This paper presents a new technique called "dynamic bound evaluation" that allows percent-done progress indicators to be routinely incorporated into a wide range of programs that are driven by recursive algorithms that are not required to follow the "linear pass over data" paradigm. In addition, the paper discusses the application of this technique to the standard Quicksort algorithm as well as to an algorithm for computing the number of vertex covers in a graph.

KEYWORDS: Percent-done progress indicators, recursive algorithms, mathematical software.

INTRODUCTION

Many software systems incorporate algorithms that execute almost instantly for some inputs while requiring a very long period of time to terminate for other inputs. This can be unsettling for users, particularly if a good estimate on the total computation time is not available. A percent-done progress indicator is a software tool that allows a user to monitor the progress that a software system has made toward responding to a command. This information can be displayed in a variety of textual or graphical formats depending on the particular interface style of the application.

For example, Myers [9] shows how a percent-done progress indicator can display its output as the rising "mercury" in a thermometer. An example of this type of display is shown in Figure 1. Much like the thermometers used in public fund-raising drives, the shading in the percent-done thermometer expands from bottom to top as the task progresses.

Figure 1: Percent-done Progress Indicator Output

Myers [9,10] has demonstrated formally that users prefer software systems that incorporate percent-done progress indicators over identical systems that do not. He notes, however, that relatively few software systems support these indicators. One reason for the scarcity of progress indicators in current software systems stems from 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 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

[*]Supported in part by the NSA under grant MDA904-90-H-4027.

[†]Supported in part by the Howard Hughes Medical Institute under grant 71108-503301.

[‡]Supported in part by the NSA under grant MDA904-90-H-4027 and by the NSF under grants IRI-8900511 and CDA-8805910.

found in systems that are primarily driven by linear algorithms [9]. Of course, many algorithms do not fall into the "linear pass over data" paradigm. For example, many of the the increasingly popular symbolic computation packages (e.g., Maple, Mathematica) rely heavily on non-linear algorithms [7, 11]. Since non-linear algorithms are likely to exhibit a greater variation in running time on different inputs than their linear counterparts, the need for progress indicators in software systems driven by this type of algorithm is especially acute.

To illustrate this point, we briefly consider the *SetPlayer* system (see [4] for a complete description of *SetPlayer*) which has motivated much of this work. The goal of this system is to provide a tool for students and researchers in Discrete Mathematics much as Mathematica [11] and Maple [7] provide tools for students and researchers in calculus and algebra. One of *SetPlayer's* functions, which we will refer to as *VC*, takes a graph G as input and displays the number of *vertex covers* of G as output.

A *vertex cover* of a graph is a subset of the vertices of the graph having the property that every edge of the graph has at least one endpoint in the subset. For example, the graph G shown in Figure 2 has the following six vertex covers: $\{A, C\}$, $\{A, B, C\}$, $\{A, B, D\}$, $\{A, C, D\}$, $\{B, C, D\}$, and $\{A, B, C, D\}$. Problems related to vertex covers arise frequently in Mathematics, and theoretical Computer Science. Many of these problems, including the problem of computing the *VC* function defined above, are NP-hard.

Figure 2: Graph G

Even for graphs with the same number of vertices, the same number of edges, and identical degree sequences, the time required to compute *VC(G)* can vary drastically. Table 1 shows the time required to compute *VC* on three graphs G_1, G_2, and G_3. The computation was performed using a Sun SPARCstation-1.

Each of the graphs from Table 1 is on 110 vertices and has 120 edges. Since the standard quantitative measures of the graphs are identical, there is no easy way for the user to predict ahead of time that one of the graphs will require significantly more processing time

Graph	*Time Required*
G_1	4.5 hours
G_2	19.7 seconds
G_3	4.8 seconds

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

than the others. However, with the help of a percent-done progress indicator, a user might be able to make this determination after a small percentage of the computation has 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.

The dramatic variance in time illustrated in Table 1 is by no means limited to the algorithm for computing the *VC* function. For example, the well-known Quicksort algorithm sorts an n-element array in $O(n \log n)$ time on average but requires $O(n^2)$ time in the worst case [1]. In [2] Bentley describes a situation where users were worried that their sorting utility was stuck in an infinite loop: "...a sort they expected to take a few minutes would in fact require a few weeks of CPU time ...". Clearly a progress indicator would have been helpful to these users.

In the domain of symbolic computation systems, the need for progress indicators has been clearly acknowledged by Caviness and Boyle when they write in *Future Directions for Research in Symbolic Computation* that "Carefully designed monitoring aids for software and hardware would be useful in measuring space, time, and progress toward a solution" [6]. Similarly, Knuth has recognized the need for progress indicators in backtracking programs. To emphasize his point, Knuth writes that he "...once waited all night for the output from such a program only to discover that the answers would not be forthcoming for about 10^6 centuries" [8].

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 next section of this paper we present a technique called *dynamic bound evaluation* that allows percent-done progress indicators to be incorporated into a wide range of software systems that are driven by recursive algorithms which are not limited to following the "linear pass over data" paradigm. We have applied this technique to the standard Quicksort algorithm as well as to *SetPlayer's VC* algorithm. The details of the Quicksort application are presented in this section. The final section of the paper presents empirical results that illustrate the effectiveness of the technique

as it has been applied to the Quicksort algorithm as well as to *SetPlayer's* algorithm for computing the *VC* function.

DYNAMIC BOUND EVALUATION

The Basic Scheme

The dynamic bound evaluation technique, which we present in this section, 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. As an example of viewing the execution of an algorithm in terms of the associated computation tree, we consider the well-known Quicksort algorithm (see for example [1]). The root of the Quicksort computation tree is labeled by the original array to be sorted. Whenever an array labeling node p of the computation tree is divided into two smaller arrays, these new arrays label the nodes of the tree which are the children of p. If the array labeling node p can be sorted without further subdivision, then p is a leaf of the computation 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 the case of sorting algorithms it is common to measure the work done by the algorithm in terms of the number of key comparisons that the algorithm makes (see, for example [1].) The dynamic bound evaluation technique requires the existence of at least one of the following functions which provide bounds on the amount of work that will be required to apply an algorithm to a particular input.

- The function $ub(S)$ which gives an upper bound on the amount of work that will be performed by the algorithm when it is applied to 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 amount of work that will be performed by the algorithm when it is applied to 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 subproblems. Further, the value of $ub(S)$ decreases as n decreases. Similarly, the value of $lb(S)$ increases as n decreases.

If at some early stage of the computation we can compute (1) the number of units of work the algorithm has completed, and (2) the number of units of work that will be required before the algorithm terminates, then we can estimate the percentage of the computation that has been completed. Computing (1) is a routine task; the program can simply count each unit of work as it is performed. 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 then 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 work the algorithm will carry out. 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 $S_1, S_2, \ldots, S_n$, and i units of work have already been performed. ¿From the definitions of the functions $ub(S)$ and $lb(S)$, it is clear that an upper bound, u, on the total work the algorithm will perform is given by

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

and similarly, a lower bound, l, on the total work the algorithm will perform is given by

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

Since i gives the amount of work that has already been performed, we can easily compute the percentage of the computation that has completed in the best and worst cases.

Bounds for the Quicksort Algorithm

We measure the amount of work that the quicksort algorithm performs by the number of comparisons of key elements that the algorithm makes. We assume an implementation of Quicksort in which key comparisons are made only during the partitioning phase of the algorithm. Further, we assume that n such comparisons are required to partition an array of size n into two subarrays (see [1] for a description of the Quicksort algorithm). It is well-known that the worst case for Quicksort occurs when the element selected as the pivot during the partitioning phase is a maximum or minimum value over all data elements. In this case an array of size n is partitioned into two subarrays with sizes $n-1$ and 0 (recall that the pivot element is placed in neither subarray). Since arrays of size less than two need no further processing, it follows that in the worst case the total amount of work performed by the Quicksort algorithm when applied to an array of size n is given by the recurrence:

$$w(n) = \begin{cases} 0 & \text{if } n < 2 \\ n + w(0) + w(n-1) & \text{otherwise} \end{cases}$$

Similarly, the best case for Quicksort occurs when the two subarrays produced by the partitioning phase are as equal in size as possible. Therefore, in the best case the total amount of work performed by the Quicksort algorithm when applied to an array of size n is given by the recurrence:

$$w'(n) = \begin{cases} 0 & \text{if } n < 2 \\ n + 2 * w'((n-1)/2) & \text{if } n \geq 2 \wedge n \text{ odd} \\ n + w'(\lfloor (n-1)/2 \rfloor) + & \\ \quad w'(\lceil (n-1)/2 \rceil) & \text{otherwise} \end{cases}$$

These functions can be used directly by the percent-done progress indicator, or can first be expressed in closed form. For example, the recurrence for $w(n)$ given above can be re-written in closed form as: $w(n) = (n^2+n)/2 - 1$. Letting $|S|$ be the size of the array representing subproblem S, the functions $ub(S)$ and $lb(S)$ that are needed to drive the progress indicator are defined by $ub(S) = w(|S|)$ and $lb(S) = w'(|S|)$ respectively.

Partial vs. 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 total work will be significantly smaller than the corresponding upper bound. In this case the technique only computes the percentage of work that has been completed 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 work that has been completed. 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 may occur at an 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 one minute 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* twenty minutes of computation time will be required to complete the task.

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 $S_1, S_2, \ldots, S_n$. 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 amount of work that will required for the algorithm to complete 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 become equal.

As an initial attempt at accomplishing this goal we can try storing the uncompleted subproblems in a priority queue. Before enqueing 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[1]. 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.

EMPIRICAL RESULTS

In this section we present examples of some data which indicate the effectiveness of the dynamic bound evaluation technique. More detailed data which better illustrate the rate at which the upper and lower bounds may be expected to converge can be found in [5].

We have tested the dynamic bound evaluation technique on the Quicksort algorithm, as well as on *SetPlayer's* algorithm for computing the *VC* function. We begin our presentation with data collected from the Quicksort algorithm. The entries in Table 2 show the percentage of the computation that was actually completed when the dynamic bound evaluation technique produced bounds satisfying the conditions given at the top of the corresponding column. Specifically, the column labeled *Partial* denotes the percentage of the computation that was complete when the upper and lower bounds satisfied the condition $lb = ub/10$. When the bounds satisfy this condition, we say the total amount of work required is known to within an order of magnitude. The column labeled *Total* denotes the percentage of the computation that was complete when the upper and lower bounds first became equal. As the data indicates the transition to complete percent-done feedback occurs at a late stage of the computation (we will see that the bounds for the Vertex Cover algorithm converge much more quickly). However, after approximately one tenth of the computation is complete, the user is informed of the remaining computation time to within an order of magnitude. This information lets the user estimate whether the remaining computation will require a matter of seconds, minutes, hours, or days.

N	*Partial*	*Complete*
500	7.9 %	99.9%
1,000	8.4 %	99.9%
5,000	10.4%	99.9%

Table 2: Transition from Partial to Complete Percent-done Feedback when Applying Quicksort to an Array of Size N

We next consider data for the *VC* function. As described previously, the input to this function is a graph G, and the output is the number of vertex covers of G. Recall the values shown in Table 1 which illustrate that the time needed to compute this function can vary greatly even for two inputs of identical size. This makes the function an ideal test-bed for dynamic bound evaluation.

The algorithm that drives *SetPlayer's VC* command is recursive and belongs to the class of algorithms that builds a tree of subproblems as it executes. We have developed functions $ub(G)$ and $lb(G)$ that evaluate to strict (as opposed to asymptotic) upper and lower bounds on the work that will be performed by *SetPlayer's* algorithm for computing *VC* when applied to graph G. Although complete derivations of these functions are beyond the scope of this paper, they are similar in spirit those presented for the Quicksort algorithm. In brief, the derivations 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 [3,4] for a complete description of *SetPlayer's* algorithm for computing *VC*, as well as a discussion of the recurrence relations that model the algorithms and the solutions to these recurrences.

Our presentation of data for the *VC* algorithm begins by returning to the graphs G_1 and G_2 presented in Ta-

[1] This cannot happen with the Quicksort algorithm, but it can happen for algorithms which may produce an exponentially large number of subproblems

ble 1 in the Introduction of this paper. Recall that both of these graphs were on 110 vertices and 120 edges. Although the standard quantitative measures of these graphs are the same, as reported in Table 1, *SetPlayer* required approximately 4.5 hours to compute VC on G_1 but required only approximately 20 seconds to compute the same function on G_2.

Tables 3 and 4 illustrate that the dynamic bound evaluation technique was able to detect the drastic variation in time required to compute VC on these two graphs. Table 3 shows that after only one unit of work had been completed, the upper and lower bounds on the total work required 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.

Nodes Completed	*Lower Bound*	*Upper Bound*
1	2,097,151	2,097,151

Table 3: Percent-Done Feedback for $VC(G_1)$

Nodes Completed	*Lower Bound*	*Upper Bound*
1	57	786,431
50	293	1,951
127	1,535	1,535

Table 4: Percent-Done Feedback for $VC(G_2)$

For the case of graph G_2, the convergence was somewhat slower as illustrated in Table 4. This table shows that after 127 units of work had been constructed, the upper and lower bounds on the total work required 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 3 and 4 were gathered using a value of $C_p = 2,200$.

We conclude this section by presenting data illustrating the effect of varying the value of the constant C_p. Table 5 presents data obtained by computing VC of a randomly generated 3-regular graph on 60 vertices (a graph is said to be k-regular iff every vertex of the graph is incident to precisely k edges).

The entries in Table 5 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

C_p	$lb = ub/10$	$lb = 9ub/10$	$lb = ub$
3,600	4.3%	50.3%	60.8%
4,800	4.3%	35.1%	50.0%
6,000	4.3%	24.1%	38.6%
7,200	4.3%	22.2%	36.2%

Table 5: Convergence of Upper and Lower Bounds when Computing VC on a 3-Regular Graph on 60 Vertices Using Various Values of C_p

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 total work required was one-tenth of the upper bound (in this case the total amount of work required 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 $VC(G)$ is computed, *SetPlayer* automatically adjusts the value of C_p based on the size of graph 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.

Previously, percent-done progress indicators have been limited to linear algorithms. However, the data presented in this section indicate that the dynamic bound evaluation technique is a viable approach to incorporating percent-done progress indicators into a large class of recursive algorithms. The problem of incorporating percent-done progress indicators into algorithms that follow other paradigms remains open.

REFERENCES

1. Aho, Alfred V., Hopcroft, John E., and Ullman, Jeff D. *Data Structures and Algorithms.* Addison-Wesley Publishing Co., Reading Massachusetts, 1982.

2. Bentley, Jon. Software Exploratorium: The Trouble with Qsort. *UNIX Review*, Vol. 10, February 1992, pages 85-93.

3. Berque, Dave. *Implicit Set Manipulation: Theory and Practice.* Ph.D. Dissertation, Rensselaer Polytechnic Institute, Department of Computer Science, July 1991. UMI Dissertation Service Publishers, Ann Arbor, Michigan.

4. Berque, Dave, Cecchini, Ron, Goldberg, Mark and Rivenburg, Reid. The *SetPlayer* System for Symbolic Computation on Power Sets. To appear in the *Journal of Symbolic Computation.*

5. Berque, Dave and Goldberg, Mark. Monitoring an Algorithm's Execution. Submitted to *Proceedings of the DIMACS Workshop on Computational Support for Discrete Mathematics*, American Mathematical Society.

6. Caviness, B. F. and Boyle, A (editors). Report of a Workshop on Symbolic and Algebraic Computation, Society of Industrial and Applied Mathematics, August 1989.

7. Char, Bruce W. *First Leaves: A Tutorial Introduction to Maple.* WATCOM Publications, Waterloo, Ontario, 1988.

8. Knuth, Donald E. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, January 1975.

9. Myers, Brad A. The importance of percent-done progress indicators for computer-human interfaces. In *CHI '85 Proceedings*, pages 11–17, April 1985.

10. Myers, Brad A. Using percent-done indicators to enhance user interfaces. In *Graphics Interface '85*, pages 167–170, May 1985.

11. Wolfram, Stephen. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley Publishing Company, Redwood City, California, 1988.