DIMACS Technical Report 95-27
July 1995

A Formal Framework for Evaluating Heuristic Programs [1]

by

Lenore Cowen[2] Joan Feigenbaum[3] Sampath Kannan[4]

[1]Most of this work first appeared in an AT&T Bell Laboratories Technical Memorandum on December 1, 1994.

[2]DIMACS postdoctoral visitor, 1994. Visit supported by an NSF Postdoctoral Fellowship.
Dept. of Math. Sciences, Johns Hopkins University, Baltimore, MD 21218.
[3]DIMACS permanent member.
AT&T Bell Laboratories Room 2C-473, 600 Mountain Avenue, Murray Hill, NJ 07974.
[4]DIMACS visitor Spring, 1994.
Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104.

DIMACS is a cooperative project of Rutgers University, Princeton University, AT&T Bell Laboratories and Bellcore.

DIMACS is an NSF Science and Technology Center, funded under contract STC–91–19999; and also receives support from the New Jersey Commission on Science and Technology.

ABSTRACT

We address the question of how one evaluates the usefulness of a heuristic program on a particular input. If theoretical tools do not allow us to decide for every instance whether a particular heuristic is fast enough, might we at least write a simple, fast companion program that makes this decision on some inputs of interest? We call such a companion program a *timer* for the heuristic. Timers are related to program checkers, as defined by Blum [3], in the following sense: Checkers are companion programs that check the *correctness* of the output produced by (unproven but bounded-time) programs on particular instances; timers, on the other hand, are companion programs that attempt to *bound the running time* on particular instances of correct programs whose running times have not been fully analyzed. This paper provides a family of definitions that formalize the notion of a timer and some preliminary results that demonstrate the utility of these definitions.

1 Introduction

We address the question of how one evaluates the usefulness of a heuristic program on a particular input of interest. Our intuitive notion of a "heuristic program" is one that is known to produce correct answers but whose running time is not analyzable or has not been analyzed. For example, a heuristic that computes an NP-hard function might, for the problem size at hand, finish in under an hour on some instances, take several hours on some other instances, and run for an entire week on the rest. If our theoretical tools do not allow us to characterize the three classes of instances precisely, might we at least write a companion program that, on some relevant instance, takes five or ten minutes to tell us that we should give up on using this heuristic on this instance, unless we are willing to wait all week? In a related example, we may have code for two different heuristics for the same NP-hard function; can we write a fast program that, on some inputs of interest, tells us that one heuristic will finish significantly sooner than the other? We call such a companion program a *timer*.

This paper proposes a formal framework for the evaluation of heuristic programs and provides initial evidence of the effectiveness of the framework. Let f be a function defined on a domain $D = \cup_{n\geq 0} D_n$ and H be a heuristic program that computes f. The timer E is also defined on domain D. Let d be a "deadline function" defined on the natural numbers. $T_H(x)$ is the running time of H on input x. For a particular $x \in D_n$, we are interested in whether or not $T_H(x) \leq d(n)$. If the timer decides that $T_H(x) > d(n)$, it outputs STOP; if it decides that $T_H(x) \leq d(n)$ or if it cannot decide one way or the other, it outputs GO.

Ideally, a timer would output STOP if and only if $T_H(x) > d(n)$. However, this ideal is not attainable in many realistic situations, and we want the scope of our study to include timers that are useful even though they do not achieve the ideal. We take as our point of departure the following basic principle: A timer E should not render the heuristic program H less useful than H is on its own; therefore, E should not tell us to STOP on instances on which H meets our deadline. On the other hand, E should add some value to H; therefore, on at least some of the instances on which H does not meet the deadline, E should tell us to STOP. Thus there might be "bad" instances for this heuristic that the timer "misses," but it cannot miss them all. At the same time, it never calls a "good" instance bad. A family of definitions that capture this notion formally is presented in Section 2.

The concept of timers is related to the concept of program checking introduced by Blum [3]. A checker is a companion program that checks the *correctness* of the output produced by an (unproven but bounded-time) program on a particular instance. Timers, on the other hand, are companion programs that attempt to *bound the running time* on a particular instance of a correct program whose running time has not been fully analyzed.

Our work on timers is in part a continuation of the research program on checking: Recall that Blum says of his definition of a program checker that

> in the above [definition], it is assumed that any program ... for a problem π halts on all instances of π. This is done in order to help focus on the problem at hand. In general, however, programs do not always halt, and the definition of

'bug' must be extended to cover programming errors that slow a program down or cause it to diverge altogether [3, pp. 2–3].

It is exactly when a timer says STOP that it has detected a "bug" of this form.

Program checking was introduced with a practical motivation, but it has had a profound impact on complexity theory [14, 19, 2]. We hope that the study of timers, also motivated by practical concerns, will lead to interesting theoretical results.

The next section contains our family of definitions. Section 3 gives examples of timers drawn from diverse problem areas in computer science. Finally, in Section 4, we propose directions for future work, including some alternative ways to formalize the intuitive notion of timer.

2 Definitions

Let f, H, E, d, $D = \cup_n D_n$, and $T_H(x)$ be as in Section 1. The heuristic program H is assumed to be correct, but nothing is assumed about its time complexity.

Definition 2.1 Let E and H be deterministic programs. E is a **timer for** $\boldsymbol{(H, d)}$ if

1. For all n and all $x \in D_n$, if $E(x) = \text{STOP}$, then $T_H(x) > d(n)$.
2. For all n, if there is at least one $x \in D_n$ for which $T_H(x) > d(n)$, then there is at least one $x \in D_n$ for which $E(x) = \text{STOP}$.

We note that there are several situations in which timer design is trivial, including the following three.

1. For a deadline function $d(n)$, there is a trivial timer that runs in time $d(n)$: It simply simulates H for $d(n)$ steps. To disallow this, we will insist that the timer run in time $o(d(n))$. If the running time of H is superpolynomial, we may insist on satisfying the stricter requirement that the timer be polynomial-time.
2. If H works by partitioning the input space into "easy cases" and "hard cases," testing in time $o(d(n))$ whether an input is an easy case, and finishing in time less than or equal to $d(n)$ exactly on these cases, then a trivial timer E would simply perform the same test as H and output GO exactly when the input falls into the easy case.
3. If H always (resp. never) finishes in time $d(n)$, then a timer E that always outputs GO (resp. STOP) is a third type of trivial timer.

As discussed in Section 1, timers are in some way analogous to program checkers as defined in [3]. With Definition 2.1 in hand, we can point out two respects in which timers and checkers are fundamentally different. A checker is an oracle machine that calls the program H whose output is being checked, whereas a nontrivial timer cannot call H as a subroutine. Secondly, a checker, by definition, must work for any program H that purports to compute the function f, whereas a timer is, by definition, a companion of a specific heuristic program H.

We regard Definition 2.1 as a version of the weakest possible requirements that a timer must satisfy to be worthy of the name. Such a definition could be useful in proving interesting negative results. In dealing with real heuristic programs, however, we would like to have timers that recognize a substantial fraction of the bad instances in each subdomain, rather than just a single bad instance. This more pragmatic requirement is formalized in Definition 2.2.

Definition 2.2 As in Definition 2.1, E and H are deterministic. Let $g(n)$ be a polynomial. E is a **g-strong timer for (H, d)** if

1. (Same as item 1 of Definition 2.1.)
2. There is a constant $c > 0$ such that, for all n, E has the following property: If the set X of all $x \in D_n$ such that $T_H(x) > g(n)d(n)$ is nonempty, then E says STOP on at least $\max(1, c|X|)$ of the instances in X.

We may interpret Definition 2.2 to mean that there are two thresholds, separated by a (usually small) polynomial multiplicative factor. If $T_H(x)$ is under the first threshold, the timer never says STOP; if it is between the thresholds, the timer can say STOP or GO; finally, among the instances on which it is over the second threshold, the timer says STOP on at least a constant fraction.

Our framework should clearly be able to handle both heuristics and timers that are probabilistic, and the next two definitions formalize the requirements for this case.

Definition 2.3 Suppose that at least one of E and H is probabilistic. The probabilities in items 1 and 2 below are computed over the coin-toss sequences of the relevant programs. E is a **probabilistic timer for (H, d)** if there are polynomials $p(n)$ and $q(n)$ such that

1. For all n and all $x \in D_n$, if $\text{Prob}(E(x) = \text{STOP}) \geq 1/p(n)$, then $\text{Prob}(T_H(x) > d(n)) \geq 1 - 1/q(n)$.
2. For all n, if there is at least one $x \in D_n$ for which $\text{Prob}(T_H(x) > d(n)) \geq 1 - 1/q(n)$, then there is at least one x for which $\text{Prob}(E(x) = \text{STOP}) \geq 1/p(n)$.

We also define timers that actually do satisfy the ideal discussed at the beginning of this section.

Definition 2.4 Suppose that E and H are deterministic. E is a **complete timer for (H, d)** if, for all n and all $x \in D_n$, $E(x) = \text{STOP}$ if and only if $T_H(x) > d(n)$.

Similarly, we also define ***g*-complete**, ***g*-strong probabilistic**, **strong**, and **complete probabilistic** timers, and we give examples of some of these in Section 3. Other variations on the notion are possible. For example, each type of timer that we have defined has an "infinitely often" form in which the second property is required to hold for infinitely many n, instead of for all n. Which formal definition one should satisfy depends on the circumstances. Analogously there are different definitions of "one-way function," some useful in complexity theory [6, 13] and some in cryptography [8, 5, 7, 15, 18].

Another useful analogy can be drawn with the study of "reactive systems," such as process controllers, communication protocols, and operating systems; formal treatments of such systems always identify "safety" and "liveness" properties. In all of the definitions we have presented for timers, item 1 corresponds to safety and item 2 to liveness. Just as in the study of reactive systems, the correct formulation of safety and liveness depends on context.

One straightforward class of timers arises as follows. If there are easy-to-compute implicit parameters, such the number of edges of a graph, the diameter of a graph, etc., on which the running time of a heuristic H depends, then a simple strategy for a timer is to evaluate these implicit parameters and decide whether to run H or not. In Sections 3.4 below, we exhibit timers that *approximate* implicit parameters that govern the running time of the heuristic but may be hard to compute.

We conclude this section with some basic negative results about timers. First we exhibit a heuristic for which there is no nontrivial complete timer.

Proposition 2.5 Let d be a fully time-constructible deadline function. Then there is a heuristic H such that any complete timer for (H, d) must be trivial.

Proof: Consider the "universal heuristic" H that takes as input a (program, input) pair (P, x) and simulates P on x. Simple diagonalization shows that there cannot be a nontrivial complete timer E for (H, d). Suppose there were such an E. Because it is nontrivial, its running time is $o(d(n))$. Consider the program P_E that behaves as follows on input x. P_E first computes $E((P_E, x))$. If E outputs GO (i.e., indicates that $T_H((P_E, x)) \leq d(n)$), then P_E runs for an additional $d(n)$ steps; if E outputs STOP (i.e., indicates that $T_H((P_E, x)) > d(n)$), then P_E halts. E cannot be a complete timer, because it is incorrect on (P_E, x): It outputs STOP exactly when $T_H((P_E, x)) \leq d(n)$. ∎

The reason that the construction in Proposition 2.5 does not provide a counterexample to the weaker Definition 2.1 is that there is no particular input length on which the timer is always wrong. An encoding trick is used in Proposition 2.6 to overcome this.

Proposition 2.6 Let d be a fully time-constructible deadline function. Then there is a heuristic H such that any timer for (H, d) must be trivial.

Proof: Let $\{P_i\}_{i \geq 1}$ be an enumeration of all programs and $f : Z^+ \times Z^+ \rightarrow Z^+$ be a one-to-one function. Let H, E, and P_E be as in Proposition 2.5, except that an input (P_i, x), where x is of length n, must be encoded as a string of length $f(i, n)$ before it is presented to H. If j is the index of the diagonalizing program P_E, then E will be wrong on all inputs of length $f(j, \cdot)$ and hence will not satisfy Definition 2.1. ∎

3 Examples of Timers

In this section, we describe several examples of timers. Our examples are chosen to satisfy a variety of the definitions given in the previous section.

3.1 Bubble Sort

Let H be a standard implementation of Bubble Sort, such as the one given in Knuth [12]. Let $d(n) = \Omega(n^2)$. (This is only interesting, of course, if $d(n)$ is less than the worst-case running time of bubble sort; if it's not, then there's a trivial timer for H that just says GO on all inputs.) For input sequence $(x_1, x_2, \ldots, x_n)$, we denote by $b(i)$ the *inversion number* of x_i, i.e., the number of indices j such that $j < i$ and $x_j > x_i$. Let $M(i) \equiv \sum_{k=0}^{b(i)-1} i - k$ and $M \equiv \max_{1 \leq i \leq n} M(i)$. Then it is clear from the description of H given in [12] that M is a lower bound on the running time of H on input $(x_1, x_2, \ldots, x_n)$. We use this fact to define a linear-time, deterministic, $O(1)$-complete timer E for (H, d).

Let $c \geq 2$ be a constant. E considers c segments of input elements, namely $(x_1, \ldots, x_{\frac{n}{c}})$, $(x_{\frac{n}{c}+1}, \ldots, x_{\frac{2n}{c}})$, …, $(x_{n-\frac{n}{c}+1}, \ldots, x_n)$. (If n is not a multiple of c, the last segment can be shorter than the rest.) For $1 \leq l \leq c$, E first finds the minimum element x_{i_l} in the l^{th} segment; it then computes $M(i_l)$. Let M' be the maximum, over l, of $M(i_l)$. E says STOP if and only if $M' \geq d(n)$.

It is clear that E runs in linear time. To prove that E is an $O(1)$-complete timer for (H, d), note first that, if E says STOP, then the real running time of H on input $(x_1, x_2, \ldots, x_n)$ is at least $M \geq M' \geq d(n)$. Next, we must show that there is a constant c' such that E says STOP whenever the real running time is at least $c'd(n)$. It suffices to show that M' is at least a constant multiple of M, provided that M and M' are both $\Omega(n^2)$ (as they will be when $M' \geq d(n)$). Suppose that the i for which $M = M(i)$ is in the l^{th} segment. Then $M(i_l) \leq M'$. The input element x_{i_l} is less than or equal to x_i, and thus any element that comes earlier in the input than x_{i_l} and contributes to $b(i)$ also contributes to $b(i_l)$. More precisely, $i_l \geq i - n/c$ and $b(i_l) \geq b(i) - n/c$. Thus $M' \geq M(i_l) \geq M - (n/c)^2$, which is what we wanted to show.

The constant implied in the statement that E is an $O(1)$-complete timer depends on the constant c; similarly, the meaningful range of values for c depends on the implied constant in $d(n) = \Omega(n^2)$.

3.2 Euclid's Algorithm for GCD

Suppose x and y are n-bit numbers with $x \geq y$, and the heuristic H is Euclid's algorithm for finding the gcd. It is well-known that H terminates in $O(n)$ iterations. Let $D(n)$ be the time required to divide two n-bit numbers and $A(n)$ be the time required to add two such numbers. Each iteration of Euclid's algorithm takes $\Theta(D(n))$ time.

Suppose we are given a deadline $kD(n)$. Are there inputs on which we can quickly determine that H takes at least k iterations? Let us focus on instances (x, y) such that the quotient at every iteration is 1 and such that the gcd is 1. In the sequence of values computed by Euclid's algorithm, the last two values computed are $v_m = 1$ and $v_{m-1} = a$, for some positive integers a and m, and $v_i = v_{i+1} + v_{i+2}$ for $i < m-1$. Let $x = v_1$ and $y = v_2$. Then $x = F_{m-1}a + F_{m-2}$ and $y = F_{m-2}a + F_{m-3}$, where F_i is the i^{th} Fibonacci number.

Given a deadline of $kD(n)$, we could find out whether a pair (x, y) will take exactly k iterations by solving $x = F_{k-1}a + F_{k-2}$ for a and seeing whether that value of a plugged into the equation for y checks out. Whenever (x, y) is a pair satisfying these simultaneous equations, the gcd computation will take exactly k iterations.

Next, we can also handle certain kinds of pairs (x, y) such that the gcd computation takes at least k iterations. Note once again that, if the quotients in the iterations are all 1 and the computation takes $m > k$ iterations, then $x = F_{m-1}a + F_{m-2}$ and $y = F_{m-2}a + F_{m-3}$. This can be rewritten as $x = F_{k-1}c + F_{k-2}d$ and $y = F_{k-2}c + F_{k-3}d$, where c and d are again positive integers. If the solutions c and d of these simultaneous equations are both positive, then again we have shown that the computation requires at least k iterations, and we can stay STOP.

Finally, we can remove the restriction that x and y be a pair with gcd 1. If they have gcd e, then the last two numbers (last one first) will be e and ke, for some positive integer k, and the Fibonacci argument works again.

Thus, the timer will output STOP on any pair (x, y) that takes too many iterations and has a computation where every quotient is 1. It clearly will not output STOP on any computation that finishes within the deadline. Also the timer runs in time $kA(n)$, which is $o(kD(n))$. Currently we do not know how to construct a strong timer for Euclid's algorithm.

3.3 Proving primality

Let D_n be the set of n-bit integers and H be a probabilistic program that, when given an integer x, searches for a proof that x is prime. Suppose that H proceeds by running the sophisticated algorithm of Adleman and Huang [1] for n^c steps[1] and then, if no proof of primality is found, switching to a simple-minded trial-division algorithm that takes exponential time but always decides correctly whether a number is prime or composite. Let $d(n) = n^c$. A program E that runs the Miller-Rabin compositeness test on x (which takes time $O(n^4)$) and outputs STOP if and only if the test finds a proof of compositeness is a complete probabilistic timer for (H, d).

The crucial fact about this example is that the best-known algorithms for proving primality are considerably slower than the best-known algorithms for proving compositeness. The idea can be generalized to any language $L \in \mathrm{RP} \cap \mathrm{coRP}$ with RP algorithm A and coRP algorithm B such that one of A or B is significantly faster than the other.

[1]The expression n^c is used here as a symbolic representation of the running time of the Adleman-Huang algorithm. We tried to find out what the exponent c is and instead discovered, in correspondence with the authors of [1], that it has never been calculated precisely. The reason it has not been calculated is that it is "huge" enough to ensure that the algorithm will not be used; the authors told us that $c > 50$; for purposes of this discussion, it suffices that $c > 4$.

3.4 Timing Enumeration Algorithms

In this section, we present a general method of building timers for "enumeration" (or "listing") programs. A (deterministic) *listing program* for a parameterized family S of combinatorial structures is a program that takes as input a parameter value p and gives as output a list $S(p)$. For example, the program could take as input a graph G and output the list $S(G)$ of all perfect matchings in G; in this example, the family S is the set of all perfect matchings, and the parameter values are graphs G. Similarly, a listing program could take as input a graph G and output the list of all spanning trees of G. For an excellent introduction to the theory of listing, see Goldberg [4].

We restrict attention to listing programs that run in *polynomial total time*, i.e., in time polynomial in n (the length of the input) and C (the length of the output). This restriction is imposed in order to rule out certain simple-minded listing programs that have trivial complete timers. (For example, a listing program for perfect matchings could simply try all partitions of the vertices into n pairs and output only those in which each pair is an edge; this program takes exponential time on *all* inputs and hence, for any polynomially bounded deadline function d, has a trivial complete timer that always says STOP.) The two listing problems that we examine in detail happen to have algorithms with the *polynomial delay* property, a more stringent property first defined by Johnson, Papadimitriou, and Yannakakis [10]. In a polynomial-delay algorithm, the time it takes to generate the first output configuration and the time between any two consecutive output configurations are both bounded by a polynomial in the size of the input. This stricter property is not needed for our statements about timers to be meaningful. These and other measures of efficiency are discussed in [4].

Listing programs conform to our intuitive notion of "heuristics," because the running time of such a program on input p is in general very hard to calculate; the length of the list $S(p)$ is obviously a lower bound on this running time, but this length is often hard to compute. The number of spanning trees of a graph G can be computed exactly in deterministic polynomial time [11], but the number of perfect matchings is a $\#P$-complete function [20]. A general method of building timers for listing programs is to compute (either exactly or approximately) the length l of the list $S(p)$ and then to output STOP if and only if the estimate is significantly greater than $d(|p|)$. Both the type of timer that the method yields and the meaning of "significantly" depend on the particular listing problem.

Let H be any *polynomial total time* listing program[2] for spanning trees (e.g., the one of Read and Tarjan [17]); this means that the running time of H is $poly(n, l)$, where n is the size of the input graph G, and l is the number of spanning trees. Let A be the algorithm of [11] that computes l in time $M(n)$, where $M(n)$ is the time to compute the determinant of an $n \times n$ matrix. For any deadline function $d(n)$ such that $M(n) = o(d(n))$, the algorithm

[2] We require H to be polynomial total time in order to avoid listing programs that have trivial timers. For example, many combinatorial listing problems can be solved by simple-minded programs that *always* take exponential time, even on instances in which the length of the list is subexponential; if H is such a program, the algorithm that always says STOP is a trivial timer for (H, d), where d is any subexponential deadline function.

that runs A and outputs GO if and only if $A(G) \leq d(n)$ is a nontrivial timer for (H, d).

The timer E that we give for programs that list perfect matchings uses the same basic idea as the one for spanning trees, but it differs in some details. Jerrum and Sinclair [9] give a probabilistic method for approximating the number of perfect matchings in a $2n$ vertex graph that runs in time $O(q^3 n^5 \log^2 n)$, where q is a known upper bound on M_{n-1}/M_n, the ratio of near-perfect matchings to perfect matchings. Even if a good upper bound on q is not known a priori, [9] shows how, given a candidate upper bound c_1, the algorithm can be modified to halt within a small constant factor of the time bounds reported above, with q replaced by c_1; with high probability, the modified algorithm either produces a good estimate for the number of perfect matchings or reports that M_{n-1}/M_n is greater than c_1 and halts. The graphs that pass the test for a bound on the ratio M_{n-1}/M_n (or contain 0 perfect matchings) are called *q-amenable.*

Procedure E

```
{
   if (G has no perfect matching) Output GO
   else
      Choose a polynomial q(n) such that the running time of the
         Jerrum-Sinclair algorithm is less than d(n).
         if (no such q exists) Output GO
      if (G is NOT q-amenable) Output GO.
      else
         Run the Jerrum-Sinclair algorithm for O(q^3 n^5 log^2 n) steps
         if (Estimated number of perfect matchings > 2d(n)) Output STOP
         else Output GO.
}
```

Theorem 3.1 Let H be any listing program for perfect matchings that runs in total time $Cg(n)$, where C is the number of matchings in the input graph, and let $d(n) = \Omega(n^{35} \log^2 n)$. Then E is a g-strong probabilistic timer for (H, d).

Proof: We first argue that E is a nontrivial probabilistic timer for (H, d). E says STOP only if the estimate for the number of perfect matchings is greater than $2d(n)$. With high probability this estimate is within a factor of 2. Since the number of matchings is a lower bound on the time it takes to list them, H will run for more than $d(n)$ steps with high probability. On the other hand, for n large enough, we know that there exists a graph G such that H will not finish by deadline $d(n)$ on input G, because there exist graphs with an exponential (in n) number of matchings. So it remains to show that there exists a G for which E answers STOP with high probability. In order to show this, we need that there exist a family of graphs with more than a polynomial number of matchings, for which the ratio M_{n-1}/M_n is not too large. This is satisfied by the simple observation in [9] that all bipartite graphs on n vertices with minimal degree $n/4$ are q-amenable, for $q(n) = n^2$. It is easy to construct such graphs with a superpolynomial number of perfect matchings. Thus choosing $d(n)$ to be $\Omega(n^{11}\log^2 n)$ makes E a nontrivial timer.

To show that E is a strong timer, we recall the following result of Jerrum and Sinclair about the fraction of graphs that have a bounded ratio M_{n-1}/M_n: If $p \geq (1+\epsilon)n^{-1}\log n$, then with probability $1 - O(n^{-k})$ (where k is a constant depending on ϵ) the random graph $G_{n,p}$ is $q(n)$-amenable, where $q(n) = n^{10}$. Thus E with deadline $\Omega(n^{35}\log^2 n)$ will output STOP on almost all graphs that have more than $d(n)$ matchings. Because of the bound on the total running time of H, any instance on which H takes time $d(n)g(n)$ or greater must have at least $d(n)$ perfect matchings. Because E outputs STOP on almost all such graphs, E is a g-strong probabilistic timer. ▮

A listing program for perfect matchings that runs in time $Cg(n)$, for some polynomial g, can be obtained using the "recursive listing" technique described in [4, §2.1.1]. Finally, we remark that the constants and the degrees of the polynomials in this example clearly render it impractical; it is of theoretical interest, however, because it provides a strong probabilistic timer for a class of heuristics that do not seem to have timers that are complete or deterministic.

3.5 Timing Iterative Numerical Algorithms

We now present an example of a timer from the realm of numerical analysis. Our example involves one of the simplest iterative methods; however, it is easy to see how to generalize to other iterative methods. The timer will come from a lower bound on the rate of convergence.

Let $F : R \rightarrow R$ be a continuously differentiable contraction function, with $1 > c' \geq |F'(x)| \geq c > 0$, for all $x \in R$. (See [16] for definitions.) Further assume that $F(0)$ is not 0.

Define H to be the following algorithm that takes F, ϵ, and an initial point $x^{(0)}$ as input and computes an ϵ-approximation to the (unique) fixed point of F. Define the iterative sequence

$$x^{(k+1)} = F(x^{(k)}) \ \ (k = 0, 1, \ldots)$$

H computes this sequence, beginning with $x^{(0)}$, until $|x^{(k+1)} - x^{(k)}| < \epsilon$.

That this process will converge to the fixed point of F and that this point is unique, is a well-known fixed-point theorem [16].

Now consider the following procedure E. We will show, for a certain class of functions F, that E is in fact a timer. Let d be the given deadline.

Procedure E

{ $l_F =$ `(known) lower bound on the amount of time`
        `it takes to compute` $F(x)$`, for any` x `in the domain`
  $k = d/l_F$

  `Find` c `and` c'`, with` $1 > c' \geq |F'(x)| \geq c > 0$ `for all` x `in` R
  $\alpha = \frac{1}{1-c'}|F(0)|$

```
    if (x^(0) > α and c^k(|x^(0)| − α) > ε) Output STOP
    else Output GO
}
```

Notice that we have set k to be an upper bound on the number of iterations of algorithm H that can be performed before the deadline d. We first show that, if E says STOP, then H will need strictly greater than k iterations to get within ϵ of the fixed point.

Theorem 3.2 Let s be the unique fixed point of F, satisfying the above conditions. If $x^{(0)} > \alpha$ and $c^k(|x^{(0)}| - \alpha) > \epsilon$, then $|x^{(k)} - s| > \epsilon$.

Proof: By the triangle inequality, $|s| - |F(0)| = |F(s)| - |F(0)| \leq |F(s) - F(0)|$. The assumption that $c' \geq |F'|$ implies that $|F(0) - F(s)| \leq c'|s|$. Solving for $|s|$, we get $|s| \leq \frac{1}{1-c'}|F(0)|$. By definition, the righthand side is just α; so $|s| \leq \alpha$.

Now let $\epsilon_0 = |x^{(0)} - s|$, and similarly, $\epsilon_i = |x^{(i)} - s|$. Take $x^{(0)} > \alpha$. Then $|\epsilon_{i+1}| = |F(x^{(i)}) - s| = |F(x^{(i)}) - F(s)|$. By the fundamental theorem of calculus, this is equal to $|\int_s^{x_i} F'(x)dx| \geq c|x_i - s| = c\epsilon_i$ by the assumption that $|F'| \geq c$. Iterating, we have $|\epsilon_k| \geq c^k|\epsilon_0| = c^k|x^{(0)} - s|$. Because we have shown that $|s| \leq \alpha$, we get

$$|x^{(k)} - s| \geq c^k|x^{(0)} - s| \geq c^k(|x^{(0)}| - \alpha) > \epsilon,$$

by assumption. ▮

To complete the formal proof that E is a timer, we need to show that, if there are some bad instances, there are some that E finds. However, E is not completely specified at this point. We assumed in step (2) that E finds some c and c', with $1 > c' \geq \max_{x \in R} |F'(x)| \geq c > 0$. However, we did not specify *how* E was to do this, nor did we give any restriction on the quality of these bounds. How successful E can be in accomplishing step (2), and how successful E will be as a timer, will vary enormously depending on the class of functions F considered.

As an example, we now look at the following class of functions. For $x > 0$, let $f(x) = e^{-c_1 x} + c_2 x$, for $1 > c_2 > c_1 > 0$. The input to the fixed point algorithm we wish to time are the values of c_2, c_1, an initial point $x^{(0)}$ and ϵ. Since $0 < c_2 - c_1 \leq f'(x) \leq c_2 < 1$, the timer will run procedure E with $c = c_2 - c_1$, and $c' = c_2$.

Now Theorem 3.2 implies the timer satisfies (1) in definition 2.1. It is also easy to see that there are instances on which E answers STOP and thus condition (2) is satisfied. Finally, the timer E will be nontrivial for most reasonable values of the deadline d, since each step involves just a few multiplications and divisions, whereas the fixed-point algorithm computes F once in each iteration, with high precision.

4 Discussion and Future Directions

In Section 2, we presented a family of definitions that capture many of the properties that one naturally wants in timers. However, one could present essentially the same family of definitions but make slightly different choices on certain details. For example, one could choose different thresholds for parameters such as the fraction (currently required to be a constant) of bad instances on which a strong timer must say STOP or the probability (currently required to be inverse-polynomial) with which, if a probabilistic timer says STOP, it must really be timing a bad instance. The choices presented in Section 2 should be reevaluated as more timers are exhibited.

More fundamentally, one could view timers from an overall perspective that is dual to the one we've presented. The guiding principle for the family of definitions presented in Section 2 is that a timer may not err when it says STOP but may err when it says GO. This viewpoint makes sense in scenarios in which the heuristic is run "offline," and its being slow on a particular instance is undesirable but not fatal. If the heuristic were part of a real-time system, then the effect of its running too long on the instance at hand might indeed be fatal, and one would only want to use it if one could guarantee that it would finish before its deadline. In that case, timers should be defined so that they may not err when they say GO but may err when they say STOP. It is clear how to alter the definitions presented in Section 2 so that they capture this dual notion.

As the terms "timer" and "deadline" suggest, we have so far focussed on whether a heuristic program finishes in an acceptable amount of time on the given input. It is straightforward to extend our framework so that the focus is on whether the heuristic uses an acceptable amount of space, communication bandwidth, number of processors, or any other crucial resource.

More concretely, we believe that numerical analysis is a natural application domain in which to use timers. For example, there are many iterative methods for the solution of systems of linear equations. Standard numerical analysis texts show how to *upper* bound the number of iterations that various methods will require to solve $Ax = b$ in terms of, e.g., the spectral radius of A. For timer design, however, we need to compute *lower* bounds on the number of iterations, and these bounds may depend crucially on the initial iterate $x^{(0)}$. The construction of such timers is an important goal, both practically (because the timers could be deployed in numerical linear algebra packages and help guide users' choices of iterative methods for particular problem instances) and theoretically (because the lower bounds may require new analytical results).

5 Acknowledgements

We thank Robert Cowen, Nick Reingold, and Bart Selman for fruitful discussions during the formative stages of this work. We thank Jong-Shi Pang for assistance with the proof of Theorem 3.2 and Leslie Goldberg for references on listing algorithms.

The work of the first author was supported in part by an NSF Mathematical Sciences Postdoctoral Fellowship and a consulting agreement with AT&T Bell Laboratories and was done in part at AT&T Bell Laboratories, the Institute for Mathematics and its Applications, and the DIMACS Center at Rutgers University.

References

[1] L. Adleman and M. Huang, *Recognizing primes in random polynomial time*, Proc. 19th Symposium on Theory of Computing, ACM, New York, 1987, pp. 462–469.

[2] L. Babai, L. Fortnow, and C. Lund, *Nondeterministic Exponential Time has Two-Prover Interactive Protocols*, Computational Complexity, 1 (1991), pp. 3–40.

[3] M. Blum, *Program Result Checking: A New Approach to Making Programs More Reliable*, Proc. 20th International Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 700, Springer, Berlin, 1993, pp. 2–14. First appeared in preliminary form in International Computer Science Institute Technical Report 88-009, Berkeley CA, 1988.

[4] L. Goldberg, *Efficient Algorithms for Listing Combinatorial Structures*, Cambridge University Press, Cambridge UK, 1993.

[5] S. Goldwasser and S. Micali, *Probabilistic Encryption*, Journal of Computer and System Sciences, 28 (1984), pp. 270–299.

[6] J. Grollman and A. Selman, *Complexity Measures for Public-Key Cryptosystems*, SIAM Journal on Computing, 17 (1988), pp. 309–335.

[7] J. Hastad, *Pseudo-Random Generators under Uniform Assumptions*, Proc. 22nd Symposium on the Theory of Computing, ACM, New York, 1990, pp. 395–404.

[8] R. Impagliazzo, L. Levin, and M. Luby, *Pseudo-Random Generation from One-Way Functions*, Proc. 21st Symposium on the Theory of Computing, ACM, New York, 1989, pp. 12–24.

[9] M. Jerrum and A. Sinclair, *Approximating the Permanent*, SIAM Journal on Computing, 18 (1989), pp. 1149–1178.

[10] D. Johnson, C. Papadimitriou, and M. Yannakakis, *On Generating All Maximal Independent Sets*, Information Processing Letters, 27 (1988), pp. 119–123.

[11] G. Kirchoff, *Uber die Auflosung der Gleichungen, auf welche man bei der Untersuchung der linearen Verteilung galvanische Strome gefuhrt wird*, Ann. Phys. Chem., 72 (1847), pp. 497–508.

[12] D. Knuth, *Sorting and Searching*, The Art of Computer Programming, vol. 3, Addison-Wesley, Reading, 1973.

[13] K. Ko, T. Long, and D. Du, *On One-Way Functions and Polynomial-time Isomorphisms*, Theoretical Computer Science, 47 (1986), pp. 263–276.

[14] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, *Algebraic Methods for Interactive Proof Systems*, Journal of the ACM, 39 (1992), pp. 859–868.

[15] M. Naor and M. Yung, *Universal One-Way Hash Functions and their Cryptographic Applications*, Proc. 21st Symposium on the Theory of Computing, ACM, New York, 1989, pp. 33–43.

[16] J.R. Munkres, *Topology: A First Course*, Prentice-Hall, Englewood Cliffs, 1975.

[17] R. Read and R. Tarjan, *Bounds on Backtrack Algorithms for Listing Cycles, Paths, and Spanning Trees*, Networks, 5 (1975), pp. 237–252.

[18] J. Rompel, *One-Way Functions are Necessary and Sufficient for Secure Signatures*, Proc. 22nd Symposium on the Theory of Computing, ACM, New York, 1990, pp. 387–394.

[19] A. Shamir, *IP = PSPACE*, Journal of the ACM, 39 (1992), pp. 869–877.

[20] L. Valiant, *The Complexity of Computing the Permanent*, Theoretical Computer Science, 8 (1979), pp. 189–201.