11-1989

A Distributed Search Program for the 3x + 1 Problem

Gary T. Leavens
Iowa State University

A Distributed Search Program for the 3x + 1 Problem

TR 89-22

Gary T. Leavens

November, 1989

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011

A Distributed Search Program for the $3x + 1$ Problem

Gary T. Leavens

TR #89-22
November, 1989

Abstract

This report describes the design of a distributed program that searched for peaks in certain measures related to the $3x+1$ problem. The searches for peaks in the number of steps taken, the maximum value reached, and the number of steps before the values of the iterates fall below the starting value exhibit a great deal of parallelism, but there is also some small amount of synchronization necessary. The design of a reliable and long-lived distributed system that searched for such peaks is discussed from the partitioning of the search to more detailed design issues such as ways to limit the search. The search was implemented in the distributed programming language Argus, and a few observations about Argus programming are included. An appendix includes tables of various results from the three years that the search program was running on six or more computers.

Department of Computer Science
Iowa State University
Ames, Iowa 50011-1040, USA

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The distributed system described in this report was designed to do some experimental mathematics and to allow some simple experimentation with the Argus distributed programming language and system [LS83] [LDH+87]. The system as finally implemented ran on six (and sometimes more) computers over several years, with all the computers cooperating in the search for certain numbers. The numbers themselves are related to the $3x + 1$ problem.

The rest of this chapter is devoted to describing the $3x+1$ problem and the objects of the search program itself: peaks in certain derived measures for the $3x + 1$ problem. Chapter 2 discusses the large-scale design issues involved in the search program, focusing on aspects of the problem unique to distributed programming. Chapter 3 discusses smaller-scale efficiency issues, including ways to "cut off" certain computations involved in the search. Chapter 4 draws some lessons about distributed programming and Argus.

Appendix A gives tables of results from the search. Appendix B gives a brief history of the search.

1.1 The $3x + 1$ Problem

The $3x + 1$ problem concerns iterates of the following function:

$$T(n) = \begin{cases} (3n+1)/2, & \text{if } n \equiv 1 \pmod 2, \\ n/2, & \text{if } n \equiv 0 \pmod 2. \end{cases} \tag{1.1}$$

which takes odd integers n to $(3n + 1)/2$ and even integers n to $n/2$ [Lag85, Page 4]. "The $3x + 1$ Conjecture asserts that, starting from any *positive* integer n, repeated iteration of this function eventually produces the value 1" [Lag85, Page 3]. This conjecture, as Lagarias states, is apparently intractable.

The program discussed in this report is not concerned with validating or disproving the $3x + 1$ conjecture. Instead, the program is designed to investigate certain measures related to the iterates of T (and the function H described below).

The iterates of T are simply defined. Let $T^{(0)}(n) = n$, and for all integers $k > 0$, let $T^{(k)}(n) = T(T^{(k-1)}(n))$. The sequences of iterates $(n, T(n), T^{(2)}(n), \ldots)$ is called the trajectory of n. For example, the trajectory of 7 is:

7, 11, 17, 26, 13, 20, 10, 5, 8, 4, 2, 1, 2, 1, 2, 1, ...

The alternative formulation of the function T, does not map odd integers n to $(3n+1)/2$, but rather to $3n+1$

$$H(n) = \begin{cases} 3n+1, & \text{if } n \equiv 1 \pmod 2, \\ n/2, & \text{if } n \equiv 0 \pmod 2. \end{cases} \tag{1.2}$$

The function H is modeled after the so-called hailstone algorithm [Hay84]. Since the article by Hayes was how I originally encountered the problem, the iterates of H figured prominently in the first searches.

One defines the iterates of H in the same way as T. For example, if n is 7, then the sequence of successive iterates of H is:

$$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots$$

Notice how the sequence of iterates for H differs from the sequence for T. The difference is stated precisely in the following lemmas. The first states that the iterates of T are a subsequence of the iterates of H, with the property that every even number in the sequence of iterates of H can be paired with a number in the iterate sequence of T. The second lemma states that every number k in the iterate sequence of H either occurs in the iterate sequence of T or $k/2$ occurs in the iterate sequence of T.

Lemma 1.1.1. *For all $i > 0$, for all $n > 0$, and for all $k > 0$, if $T^{(i)}(n) = k$, then there is some $j \geq 0$ such that, $H^{(j)}(n) = 2k$, and $H^{(j+1)} = k$.*

Proof: (by induction on i).

For the basis, suppose $i = 1$. If n is odd, then $T^{(1)}(n) = (3n+1)/2$, $H^{(1)}(n) = 3n+1$, and $H^{(2)}(n) = (3n+1)/2$. If n is even, then $T^{(1)}(n) = n/2$, $H^{(0)}(n) = n$, and $H^{(1)}(n) = n/2$.

For the inductive step, suppose the lemma holds for $i > 0$. Let $n > 0$, and $k > 0$ be given. Let $k_i = T^{(i)}(n)$. By the inductive hypothesis, there is some $j_i \geq 0$ such that, $H^{(j_i)}(n) = 2k_i$, and $H^{(j_i+1)}(n) = k_i$. There are two cases. If k_i is odd, then $T^{(i+1)}(n) = (3k_i+1)/2 = k$, $H^{(j_i+1)}(n) = 3k_i + 1 = 2k$, and $H^{(j_i+2)}(n) = (3k_i+1)/2 = k$. If n is even, then $T^{(i+1)}(n) = k_i/2 = k$, $H^{(j_i)}(n) = k_i = 2k$, and $H^{(j_i+1)}(n) = k_i/2 = k$. ∎

Lemma 1.1.2. *For all $j > 0$, for all $n > 0$, and for all $k > 0$, if $H^{(j)}(n) = k$, then there is some $i \geq 0$ such that either $T^{(i)}(n) = k$ or $T^{(i)}(n) = k/2$.*

Proof: (by induction on j).

For the basis, suppose $j = 1$. If n is odd, then $H^{(1)}(n) = 3n+1$ is even, so $H^{(2)}(n) = (3n+1)/2 = T^{(1)}(n)$. If n is even, then $H^{(1)}(n) = n/2 = T^{(1)}(n)$.

For the inductive step, suppose the lemma holds for $j > 0$. Let $n > 0$, and $k > 0$ be given. Let $k_j = H^{(j)}(n)$. By the inductive hypothesis, there is some $i_j \geq 0$ such that, either $T^{(i_j)}(n) = k_j$ or $T^{(i_j)}(n) = k_j/2$. There are two cases. If k_j is odd, then $H^{(j+1)}(n) = 3k_j + 1$ is even, so $H^{(j+2)}(n) = (3k_j+1)/2 = T^{(i_j+1)}(n)$. If k_j is even, then $H^{(j+1)}(n) = k_j/2 = T^{(i+1)}(n)$. ∎

There are graphs in the article by Hayes that show the wildly erratic and unpredicatable behavior of the iterates of H [Hay84]. The behavior of T is, of course, similarly wild and unpredictable.

1.2 Derived Measures

The computer program described in this report was designed to find peaks in certain measures derived from the iterates of the functions T and H. These derived measures are related to the $3x + 1$ conjecture, since one counts the number of iterations needed to reach 1, or the maximum value reached in a trajectory.

1.2.1 Stopping Times and Steps

As Lagarias states, for $n > 1$, $T^{(k)}(n) = 1$ cannot occur unless there is some m such that $T^{(m)}(n) < n$. The derived measures defined in this section are all concerned with counting the number of iterates needed to reach one of these situations.

The *stopping time* function $\sigma(n)$ is the least whole number k such that $T^{(k)}(n)$ is less than n or one[1]. If there is no such k, then let $\sigma(n)$ be ∞. For example, $\sigma(7) = 7$.

The *total stopping time* function $\sigma_\infty(n)$ is the least whole number k such that $T^{(k)}(n)$ is one[2]. If there is no such k, then let $\sigma_\infty(n)$ be ∞. For example, $\sigma_\infty(7) = 11$.

One can also define similar measures for H.

The steps function $steps(n)$ is the least whole number k such that $H^{(k)}(n)$ is one. If there is no such k, then let $steps(n)$ be ∞. It is the analog of total stopping time for H. For example, $steps(7) = 16$.

1.2.2 Maximum Value

The maximum value function $max\_value(\mathrm{n})$ is the maximum of all the integers reached by iterating H until the value of the iterates reach one. That is,

$$max\_value(n) = \max\{H^{(k)}(n) \mid 0 \leq k \leq steps(n)\}. \tag{1.3}$$

For example, $max\_value(7) = 52$.

Using T instead of H gives an alternative definition of maximum value[3].

$$alt\_max\_value(n) = \max\{T^{(k)}(n) \mid 0 \leq k \leq \sigma_\infty(n)\}. \tag{1.4}$$

For example, $alt\_max\_value(7) = 26$.

The connection between these definitions is discussed below.

1.2.3 Peaks

The distributed system described in this report searches for peaks in the derived measures.

An integer $n > 0$ is a *peak* in a derived measure f, if and only if for all $0 < m < n$, $f(m) < f(n)$. These numbers are called peaks because if one graphs the positive integers

[1] This definition differs from Lagarias's in the treatment of the number one. In Lagarias's paper [Lag85], $\sigma(1) = \infty$, but in the above definition $\sigma(1) = 0$.

[2] Again, this definition differs from Lagarias's for the number one, since in Lagarias's paper $\sigma_\infty(1) = 2$ but in the above definition $\sigma_\infty(1) = 0$.

[3] This definition also differs slightly from Lagarias's, since by my definition $alt\_max\_value(1) = 1$, not 2. However, the difference is immaterial for all other positive integers.

on the x-axis and the value of some derived measure on the y-axis, then each peak will be a point higher than has been reached before. For example, 3 is a peak in *max_value*, because $max\_value(3) = 16$, $max\_value(2) = 2$, and $max\_value(1) = 1$.

Peaks in a derived measure are also interesting because stating that there are no other peaks in that measure between m and n says a great deal about the values of the derived measure on all the numbers between m and n. Peaks appear more and more rarely as one tests larger numbers, as the tables in Appendix A bear out. While it is easy to verify the value of $steps(n)$ or $max\_value(n)$ for any particular n, it is very expensive to verify that n is a peak in either derived measure, because this involves showing that all numbers less than n have a smaller value for *steps* or *max_value*.

1.2.4 Relationships Between Peaks in Derived Measures

The following summarizes the relationships between peaks in the various derived measures. The question is whether peaks in one derived measure must be peaks in some other derived measure.

Peaks in stopping time seem unrelated to peaks in any of the other measures. But for the other kinds of peaks, there are relationships that result from the similar definitions of H and T. The main connections are summarized above in lemmas 1.1.1 and 1.1.2.

Max_Value and Alt_Max_Value

To show the relationship between *max_value* and *alt_max_value*, the following lemma is needed. It says that if a value occurs in the iterate sequence of a number greater than two, then it must occur before the iterates reach one.

Lemma 1.2.1. *For all $n > 2$, and for all k, if there is some $i \geq 0$ such that $H^{(i)}(n) = k$, then there is some $0 \leq j \leq steps(n)$ such that $H^{(j)}(n) = k$; furthermore, if there is some $i \geq 0$ such that $T^{(i)}(n) = k$, then there is some $0 \leq j \leq \sigma_\infty(n)$ such that $T^{(j)}(n) = k$.*

Proof: If $steps(n)$ is infinite, then the result follows with $j = i$.

So suppose $i > steps(n)$ and $H^{(i)}(n) = k$. Since $n > 2$, $steps(n) \geq 2$, since $steps(3) = 7$ and for all $n \geq 4$, at least two steps are needed to reach 1. But if $H^{(m)}(n) = 1$ and $m \geq 2$, then $H^{(m-1)}(n) = 2$. (If $H^{(m-1)}(n)$ is odd, then $1 = H^{(m)}(n) = 3(H^{(m-1)}(n)) + 1 > H^{(m-1)}(n)$, which is a contradiction.) Similarly, if $H^{(m)}(n) = 1$ and $m \geq 2$, then $H^{(m-2)}(n) = 4$. Since $H^{(steps(n))}(n) = 1$ by definition, k must be either 4, 2, or 1. So if $k = 1$, take $j = steps(n)$. If $k = 2$, take $j = steps(n) - 1$. Finally, if $k = 4$, take $j = steps(n) - 2$.

The proof for T is similar to the above proof for H. ∎

Peaks in *alt_max_value* are directly related to peaks in *max_value*. This is shown in the following lemma.

Lemma 1.2.2. *If $n > 2$, then $max\_value(n) = 2 \cdot alt\_max\_value(n)$.*

Proof: Suppose $alt\_max\_value(n) = k$. Let $i \leq \sigma_\infty(n)$ be such that $T^{(i)}(n) = k$. Then by Lemma 1.1.1, there is some $j \geq 0$ such that $H^{(j)}(n) = 2k$. By Lemma 1.2.1, one can choose $j \leq steps(n)$. Thus $max\_value(n) \geq 2 \cdot alt\_max\_value(n)$.

Suppose $max\_value(n) = m$. Let $j \leq steps(n)$ be such that $H^{(j)}(n) = m$. Then m is even, because otherwise $H^{(j+1)}(n) = 3m + 1 > m$. By Lemma 1.1.2, there is some $i \geq 0$ such that either $T^{(i)}(n) = m$ or $T^{(i)}(n) = m/2$. But the first case is a contradiction, since then by Lemma 1.1.1, there would be some $p \geq 0$ such that $H^{(p)} = 2m > m$. So it must be that $T^{(i)}(n) = m/2$. By Lemma 1.2.1, one can choose $i \leq \sigma_\infty(n)$. Thus $2 \cdot alt\_max\_value(n) \geq max\_value(n)$. ∎

Corollary 1.2.3. *An integer $k > 0$ is a peak in* max_value *if and only if k is a peak in* alt_max_value. ∎

Steps and Total Stopping Time

The relationship between peaks in *steps* and peaks in σ_∞ is one of the deep mathematical questions that often appear in the study of the $3x + 1$ problem. However, it is easy to see that the total stopping time cannot be greater than the number of steps, nor as small as half the number of steps.

Lemma 1.2.4. *For all $n > 0$, if $steps(n) \neq \infty$, then $steps(n)/2 < \sigma_\infty(n) \leq steps(n)$.*

Proof: In any sequence of iterates of H, every step of the form $3x + 1$ is followed by a division by 2. Thus the iteration of T removes at most half of these steps. However, if $steps(n)$ is defined, then there must be more division by 2 steps in the standard algorithm than there are steps that multiply by 3 and add one. ∎

A different relationship between total stopping time and *steps* was pointed out by Jeffery Lagarias in a letter (1987). Let $odd(n)$ be the number of odd integers in the iterate sequence of H (excluding 1) and $even(n)$ be the number of even integers that occur until 1 is reached. That is:

$$odd(n) \stackrel{\text{def}}{=} \#\{k \mid k \bmod 2 = 1, H^{(i)}(n) = k, 0 \leq i < steps(n)\} \tag{1.5}$$

$$even(n) \stackrel{\text{def}}{=} \#\{k \mid k \bmod 2 = 0, H^{(i)}(n) = k, 0 \leq i < steps(n)\}. \tag{1.6}$$

Since every step produces an odd or an even number, the sum of odd and even is the number of steps.

Lemma 1.2.5. *For all $n > 0$, $steps(n) = odd(n) + even(n)$.* ∎

It is interesting that the number of even steps is the same as the total stopping time.

Lemma 1.2.6. *For all $n > 0$, $\sigma_\infty(n) = even(n)$.*

Proof: If $\sigma_\infty(n) = \infty$, then $even(n) = \infty$. Furthermore, $\sigma_\infty(1) = 0 = even(1)$ and $\sigma_\infty(2) = 1 = even(2)$.

So suppose $n > 2$ and $\sigma_\infty(n) = m < \infty$. By Lemma 1.1.1, for each $0 \leq i \leq \sigma_\infty(n)$, there is some $j \geq 0$ such that $2T^{(i)}(n) = H^{(j)}(n)$. By Lemma 1.2.1 j can be chosen so that $j \leq steps(n)$. Thus $\sigma_\infty(n) \geq even(n)$. However, if $\sigma_\infty(n) > even(n)$, then there would have to be two iterates of T with the same value (that is, $0 \leq i < l \leq \sigma_\infty(n)$, such that

$T^{(i)}(n) = T^{(l)}(n)$), but if this happened there would be infinitely many such cases, and so $\sigma_\infty(n)$ would be infinite. ∎

Unfortunately, the above analysis does not easily enable one to prove that peaks in *steps* are also peaks in total stopping time, or vice versa. However, the peaks do coincide at least to 12.3 billion (12.3×10^9), as the search program verified. Hence I offer the following conjecture.

Conjecture 1.2.7. *An integer $k > 0$ is a peak in* steps *if and only if k is a peak in total stopping time (σ_∞).*

Jeffrey Lagarias was kind enough to write me (in 1987) to explain why this conjecture would be difficult to prove.

Chapter 2

Large Scale Design Issues

2.1 Partitioning the Search

2.1.1 Characteristics of the Problem

The search for peaks in the derived measures discussed in Chapter 1 has the following characteristics.

1. The values of a derived measure can be computed for any n independently of any other number.

2. To find peaks in a derived measure, it is not necessary to obtain the exact value for the derived measure for each number. It is possible to stop computing if the input can be shown to not be a peak. Some strategies for deciding when to stop computing on a number require knowledge of previous peaks and are most effective if the input number is not more than twice the previous peak.

3. A number is not definitely known to be a peak in a derived measure until all smaller numbers have been checked.

The first item is the source of parallelism in the problem and the second two are the sources of what little synchronization there is in the problem.

2.1.2 Partitioning the Search

Given some number of computers, the basic way to partition the problem is to give each computer a number to check and have them report back on the results. However, it is not efficient to simply give each computer one number to check, because running the algorithm takes very little time and there would be too much time spent in synchronization to obtain the next number. This is especially true in Argus, where the computers are connected over a network and communication among computers takes several orders of magnitude more time than the time for checking an individual number [LS83].

Thus the program is designed to give each computer an interval of numbers instead of a single number. Each computer checks all the numbers in the interval given to it. The size of

the interval can be made large enough so that synchronization takes an insignificant fraction of the total time spent in the search. Because the size of the interval is best determined by performance considerations, it should be easily changed while the system is running.

Although the primary task in searching for peaks is checking individual numbers, there are several bookkeeping tasks such as keeping track of the results and the intervals searched. My design involves two separate guardians. (A guardian is the basic module of an Argus program, and corresponds to an abstract resource.) The first kind of guardian is called a *search guardian*; it is responsible for searching for peaks in the derived measures. The second kind of guardian is called a *coordinator*; it is responsible for the bookkeeping tasks. (The coordinator evolved from an earlier design in which there were only search guardians and I did the bookkeeping tasks manually.) The coordinator records the peaks (and candidate peaks) that have been found, the intervals that are being searched, and the intervals that have already been searched. However, the coordinator does not do any searching for peaks on its own.

When the system is started, each search guardian immediately calls the coordinator to obtain an interval to be searched. The coordinator synchronizes the search guardians, handing out intervals so that there is no duplication of effort. Each search guardian checks each number in the interval it is assigned and reports back to the coordinator.

2.2 Design Issues

2.2.1 Coordinator State

Argus supports two kinds of storage: stable and volatile. Information recorded on stable storage survives (with high probability) the crash of a computer and any guardians running on it. However, stable storage is more expensive than volatile storage, since duplicate copies of information are necessary to ensure that it survives crashes [Lam81]. It is therefore important to decide what information will be kept in stable storage, and what will be kept in volatile storage.

Since the coordinator's purpose is to do bookkeeping, most of its information must be kept in stable storage. This includes the following essential information.

1. The peaks in the derived measures that have been found.

2. Any candidate peaks. These are numbers that have higher values for the derived measures than any known peaks, but which are not known to be true peaks because not every smaller number has been checked.

3. A list of what intervals have been searched already (so that it can be determined what numbers are peaks).

4. A list of what intervals have been handed out to search guardians (so that there will not be any duplication of effort).

One could also imagine having the coordinator track what guardians are involved in the search and what interval each guardian is searching. This information would allow the

coordinator to exert more control over the search. For example, if one guardian was slow in searching a critical interval, the coordinator could reassign it to a different interval and assign a faster guardian. Without knowing these pieces of information it is impossible to reassign intervals in this fashion. On the other hand, this kind of ability does not seem essential and it is difficult to implement; so information about what interval particular guardians are searching is not kept by my implementation.

2.2.2 Search Guardian State

The essential information that a search guardian must keep in stable storage is just the interval it is assigned. If a search guardian were to loose this information, the interval it was searching would never be searched, because the coordinator does not keep any record of which intervals are being searched by which guardians.

However, it is a good idea to save other information on stable storage periodically (i.e., take a checkpoint) so that only a small amount of CPU time will be wasted in the event of a crash. This allows one a greater probability of making progress in a long computation. (DSG note 142 discusses this point in more detail [Lea86].) Thus the next number to be searched and any (candidate) peaks found are kept on stable storage.

In addition, a search guardian also keeps information about earlier peaks in stable storage, both for detecting when it has found a new peak and for algorithmic efficiency (see below). Finally, a search guardian keeps two handler objects that are its connection to the coordinator on stable storage.

In Argus, information that is kept on stable storage must be stored in atomic data containers [LS83] that support synchronization and consistency in the face of parallel processing and machine crashes. However, atomic data containers, such as atomic arrays, are less efficient than non-atomic data containers, as there is some overhead for locking, etc. Thus non-atomic copies of all but the handlers are kept in volatile variables. A checkpoint therefore involves copying information from volatile variables to stable storage in an atomic action (i.e., a transaction). This way of programming the problem also keeps the atomic actions very short, since no searching for peaks happens inside an atomic action.

The search guardian takes a checkpoint after searching an interval of a certain size. The size is set so that checkpoints happen about once every twenty minutes and so that the checkpoint size evenly divides the size of the interval given to the search guardian by the coordinator. Making the checkpoint interval longer reduces the amount of time spent on the overhead of checkpointing as opposed to the "real work" of searching. The amount of time spent on checkpointing in the actual implementation is well below 0.1% of the total CPU time used by the guardian. Since it is important not to duplicate effort by searching past the end of the assigned interval, a search guardian needs to test a count after each number's search in any case; this check is combined with a check to see if it is time to take a checkpoint by making the size of the checkpoint interval evenly divide the size of the interval given by the coordinator.

2.2.3 Synchronization

The search guardians are synchronized by the coordinator, since the coordinator keeps track of the peaks and the intervals reported and taken for searching. It is possible for more than one search guardian to attempt to report (by making a handler call) on the search results in different intervals at the same time. Hence the coordinator's design must synchronize these possibly concurrent requests to ensure that no information is lost due to race conditions and that there is no duplication of work.

In Argus, each report request appears at the coordinator in the form of a handler call (a *handler* being the module that services remote procedure calls). If concurrent remote procedure calls occur, then there will be more than one atomic action (i.e., process), active in the coordinator guardian executing the code of the handler. Synchronization is achieved by means of atomic data objects within the coordinator delaying actions as necessary to ensure serialization.

While it may be possible for reports from the search guardians to be processed concurrently in the coordinator, concurrent processing does not seem to be necessary, because each search guardian only reports, on the average, about once a day. This decision to ignore opportunities for exploiting concurrency in the coordinator's processing of reports eases the implementation of the coordinator. In particular, the coordinator uses the Argus atomic arrays in a straight-forward fashion to implement various atomic types with low concurrency. For example, the type `peak_list`, which is used by the coordinator to keep track of the peaks found, is built using atomic arrays. Because it uses atomic arrays, the implementation of `peak_list` is easy. A peak list object can only be used by one action (i.e., by one process) at a time due to the strict locking rules for the atomic array that is used in its representation, but this hardly matters.

It is easy to ensure that an abstract type constructed from a built-in atomic type such as atomic arrays will provide the necessary synchronization and recovery. Thus the only remaining problem for the coordinator is to ensure that there are no deadlocks.

To ensure that no deadlocks occur, the handler that processes a report first gets a write lock on an atomic object. This ensures that reports are processed sequentially, and so avoids deadlocks. Hence there is no real concurrency in the coordinator guardian, but again this hardly matters.

2.2.4 The Reporting Interface

The search guardian needs to call the coordinator

- when the system is initially started to get an initial interval to search, and
- when it finishes checking all the numbers in its assigned interval.

It is convenient to make the initial request for an interval look the same as other search reports. Thus search guardians report that they have searched the interval from 1 to 3 right after they are created. Because peaks are the object of the search, it is also convenient if the search guardian can call the coordinator when it finds a peak, even if it is not finished with its assigned interval.

A search report consists of the lists of candidate peaks, the search guardian's assigned interval, how far it has searched (in case it is not done yet), and the size of the interval it desires. Having the search guardian tell the coordinator how large an interval it wants allows some flexibility in compensating for differences in the speed of various computers. The coordinator must record as "searched" the interval from the beginning of the assigned interval up to how far the guardian says it has searched. The coordinator must also record any peaks. The normal return from a search report is a new search interval and other information necessary to continue the search in that interval. The coordinator can also raise an exception that tells the search guardian to continue searching in the interval it has already been assigned.

There is also a handler call that a search guardian can make to report the results of a search without receiving a new interval to search. This would be useful during reconfiguration.

2.2.5 Reliability

The list of peaks produced by the system must correspond to the mathematical truth: that is the whole point of the program. To verify the correctness of the results, one must verify the correctness of the program. Thus the coding process was concerned with maintaining invariants, etc. I never sacrificed the program's correctness for the sake of efficiency; doing this would be pointless.

The coordinator also double checks the values of any peaks reported by the search guardians, although it cannot check that they are truly peaks. Several bugs were caught by having the coordinator double check the search guardian's reports.

The coordinator checks reports of peaks with a slow but sure algorithm. Furthermore, the coordinator and the search guardians use different implementations of the long integers necessary for the search. (This is a feature of Argus, that each guardian can be independently linked with different implementations.) The coordinator uses an implementation of long integers in which I have more confidence, since it has not been extensively tuned for performance, and since it has not been changed during the course of the search. (Not to say that I do not have confidence in the implementations used by the search guardians, but there are varying degrees of confidence that one can have in pieces of software.)

2.2.6 Security and Reconfiguration

There is also a concern for "security," because one would like to ensure that search reports only come from the search guardians. An easy way to do this is by not putting the handlers for reporting search results in the Argus catalog (a guardian known to all other Argus guardians); instead, these are given to the search guardians when the system is created.

Unfortunately, not putting handlers that the search guardians need in the catalog means that search system is not easily reconfigured. This is not serious, since ease of reconfiguration is not a major requirement.

However, one can do guardian replacement as in Mark Day's thesis [Day87], even though the coordinator's report handlers are not in the catalog. This is done as follows. It is easy to replace the search guardians using Day's scheme. Replacing the coordinator, is done as

in Day's scheme, except that instead of putting the new report handlers in the catalog, they must be sent to the individual search guardians, which have a special handler for this purpose.

In practice, however, it is just as easy to destroy the old search system and completely create a new one. The stable state of the coordinator is stored in text files during the interim. (This also provides some recourse when "stable" storage is corrupted, as happened infrequently in the early Argus implementation.)

2.2.7 Scaling

The partitioning of the search into intervals does not work well when the search is started from scratch at 1. This is because the peaks are very close together near 1 and only spread apart as the numbers increase. However, this is easily overcome by not partitioning the problem at the beginning.

Similar problems arise when considering how many computers can be fruitfully brought to bear on the problem. Certain strategies for stopping computing on a number work best if the search guardians are searching intervals that do not extend past twice the largest known peak. (This is true at least for the search for peaks in *steps*.) Thus, as the search proceeds towards infinity, there are larger and larger intervals available for searching. However, at any given point one can only employ so many search guardians, because they should have intervals large enough so that they do not spend most of their time communicating with the coordinator. Furthermore, the coordinator is not designed to have the fastest algorithms for checking and recording search reports. (The coordinator implementation, as noted above, processes search reports sequentially.)

At the extreme of thousands of processors, perhaps the best way to proceed would be to have each processor check a single number rather than an interval, have each processor that found a peak record that peak, and then have each processor work on the next number as determined by adding the number of processors[1] to the number it worked on before [HS86]. This algorithm would be difficult to manage if the processors could not move in lock step. Moreover, it would be difficult to synchronize the recording of peaks, since it is possible that two processors would find a peak during the same step of the algorithm.

2.2.8 Availability

As the system stands, there is only one coordinator guardian, which is clearly an availability bottleneck. However, the early implementation of Argus stable storage was (at the time I implemented the search program) also a single point of failure. Furthermore, the computer that the coordinator was running on was also fairly reliable. Thus the work involved in replicating the coordinator did not seem justified[2].

The search guardians, on the other hand, have to be able to continue to function when the coordinator cannot be reached. If the coordinator is unavailable when a search guardian finishes its interval, it simply continues searching from where it left off. (This is the only

[1] Actually, since the search only has to check odd numbers, one would add twice the number of processors.

[2] Especially since I was trying to finish my Ph.D. at the time and my thesis advisor would have been extremely upset if I had spent even more time on this program.

thing it can do without remembering more than the current amount of state information.) Each time it takes a checkpoint, if it has reached the end of its assigned interval or surpassed it, it tries to call the coordinator. Thus, when the coordinator cannot be reached, the search guardian tries to call it more frequently, but it keeps on working. Chances are, however, that the work it is doing will duplicate that already done by another search guardian.

2.2.9 Niceness and Politeness

The search guardians are CPU hogs. Originally it was thought that by running them at the lowest possible Unix priority (also known as the "nice" value), they would not get in anyone's way. Unfortunately, the 4.3 BSD Unix process scheduling algorithm handles CPU-intensive low priority processes poorly[3].

Thus, besides being nice, the search guardians are also polite; that is, they check, every 10 minutes, to see if anyone is logged in, and if so, they go to "sleep" by calling a built-in *sleep* primitive of Argus. A sleeping search guardian uses no CPU time and little (real) memory. A sleeping guardian awakens every hour to check if anyone is logged in; if no one is logged in it starts searching again.

The original implementation of this login checking was done by the search whenever it took a checkpoint. However, there was always a compromise between wanting to have a short interval between checking for logins and wanting to have a fairly long interval between taking checkpoints. Furthermore, as the search progressed towards infinity, the search guardians would tend to take longer intervals between checkpoints, so tuning the size of the checkpoint interval was difficult. At Mark Day's suggestion, I implemented the login checking as a separate process within the search guardian. It is thus easy to set the time between checks for logins. This process sets a boolean variable to tell the background process (running the search proper) when to go to sleep. The background process's code checks this variable after each number's search.

The search guardians also have a handler that can be called to set the variable indicating that someone is logged in. Hence calling this handler causes the search guardian to go to sleep immediately (after taking a checkpoint and calling the coordinator). People could thus make the search guardian go to sleep as soon as they logged in my running a program (which I provided). This program is also useful for making a search guardian call the coordinator; for example before restarting the system.

2.2.10 Communication with the Outside World

Output from the search program is obtained by handler calls. The coordinator can be called to return the list of peaks and the intervals reported and taken for searching. The search guardians support a similar handler that allows one to find out the state of a particular

[3]It was observed that having the search guardian at lowest priority slowed down other jobs, especially balanced I/O and CPU jobs such as document formatting. The hypothesis that seems to explain this behavior is that when some high priority Unix process gives up the CPU (e.g., to do some I/O), the scheduling algorithm would begin to bring the search guardian into main memory, causing some of the pages of the high priority process that were in main memory to be written out to disk. This would diminish the working set of the high priority process, cause more disk activity, and in short, slow down the high priority process.

search guardian. The search guardians also support a handler call that makes them go to sleep (see above).

Other handlers are available for changing various parameters that govern the performance of the search. For example, the search guardians can be asked to change the size of the interval that they request from the coordinator and how often they take checkpoints. This kind of handler has been the source of much evolution during the development of the system, as it seems that there is always one more number or variable that should be changeable while the system is running.

Chapter 3

Small Scale Design Issues

This chapter discusses the efficiency of the search guardian, which amounts to a discussion of the algorithms for iterating H and T. See Section 2.2.2 for a discussion of how the overhead of the search is made insignificant.

A fundamental observation is that peaks are extremely rare. For example, in the first 50 billion integers, there are only 49 peaks in *max_value* and only 78 peaks in *steps*. (One reason why the search for peaks is so attractive is because there are minimal storage requirements and the output is not overwhelming.) The peaks become more and more rare as the search progresses; between 1 billion and 50 billion there are only 5 peaks in *max_value* and 12 peaks in *steps*. So a typical number is not a peak, and the main task of the search guardian is to find this out as quickly as possible. There are two basic strategies for doing this.

- Discovering that the input number is not a peak before taking it through all the iterates of H or T down to 1 (or until the values of the iterates fall below the starting value if one is searching for peaks in stopping time). This is called *cutting off the search.*
- Running the steps of the iteration algorithm faster. This is involves both faster algorithms that are equivalent to iterating H or T and hackery to make multiplying, dividing, adding, and comparing numbers faster.

3.1 Cutting off the Search

The best way to cut off the search on a given input number is to prove that the input cannot be a peak and to ignore it without spending any time on it. This is called an *a priori* cutoff. A less effective way to cut off the search on a given input is to prove that the number cannot be a peak after learning something about the path that it takes. This is called an *a posteriori* cutoff.

3.1.1 A Priori Cutoffs

A basic result is that it is possible, *a priori*, to limit the search to odd numbers.

For *max_value*, it suffices to note that the first step of H for an even number is to divide it by two.

Lemma 3.1.1. *For any $k > 0$, $max\_value(2k) = \max\{2k, max\_value(k)\}$.*

Proof: $H(2k) = k$, and thereafter iterating H produces the same sequence of values as iterating H on k. ∎

Corollary 3.1.2. *The number 2 is the only even peak in* max_value.

Proof: Let $k > 1$ be given. By the above lemma, $max\_value(2k)$, is either the maximum of $2k$ or $max\_value(k)$. If $max\_value(2k) = max\_value(k)$, then $2k$ is not a peak. So suppose $max\_value(2k) = 2k$. But then $2k$ is not a peak in max_value either, since $max\_value(2k - 1) \geq 3(2k - 1) > 2k$. The inequality $max\_value(2k - 1) \geq 3(2k - 1)$ holds because $2k - 1$ is odd, hence $H(2k - 1) = 3(2k - 1)$. That $3(2k - 1) > 2k$ holds for $k > 1$ holds is shown by the following:

$$\begin{aligned} k > 1 &\Rightarrow 4k > 4 && (3.1)\\ &\Rightarrow 4k - 3 > 1 && (3.2)\\ &\Rightarrow (6k - 3) - 2k > 1 && (3.3)\\ &\Rightarrow (6k - 3) > 2k + 1 && (3.4)\\ &\Rightarrow 3(2k - 1) > 2k. && (3.5) \end{aligned}$$

∎

Results similar to the above apply to *alt_max_value* as well.

For *steps*, the same observation about the first step of H means that an even number k will take only one more step than $k/2$ to return to 1.

Lemma 3.1.3. *For any $k > 0$, $steps(2k) = 1 + steps(k)$.* ∎

Corollary 3.1.4. *If k is a peak in* steps, *then the least even number greater than k that can be a peak in* steps *is $2k$.*

Proof: Let k be a peak in *steps*. By definition, for all $0 < j < k$, $steps(j) < steps(k)$. Thus by the preceding lemma,

$$steps(2j) = 1 + steps(j) \leq steps(k) = steps(2k) - 1 < steps(2k).$$

∎

A similar result applies to total stopping time.

By these corollaries, it is easy to predict all the even peaks in *steps* and *max_value*. Thus the search for these peaks can ignore all the even numbers and need only check the odd numbers.

For stopping time, the first division by two means that an even number always has a stopping time of 1.

Lemma 3.1.5. *For any $k > 0$, $\sigma(2k) = 1$.* ∎

Corollary 3.1.6. *If $k > 2$ is a peak in stopping time, then k is odd.*

Proof: $\sigma(3) = 4 > 1$. ∎

So the search for peaks in stopping time can also ignore all the even numbers.

The following results allow the search for peaks in stopping time to effectively ignore half of the odd numbers as well, that is, those that are equal to 1 modulo 4. (Mike Vermeulen brought the numbers equal to 1 modulo 4 to my attention in connection with an idea for cutting off the search for peaks in *max_value*.)

Lemma 3.1.7. *For all $k > 0$, if $k \bmod 4 = 1$, then $T(k)$ is even.*

Proof: Suppose $k > 0$ and $k \bmod 4 = 1$. Then $k \bmod 2 = 1$, and hence $T(k) = (3k+1)/2$. But $(3k+1)$ is evenly divisible by 4:

$$k \bmod 4 = 1 \quad \Rightarrow \quad 3k \bmod 4 = 3 \tag{3.6}$$

$$\Rightarrow \quad (3k+1) \bmod 4 = 0, \tag{3.7}$$

and therefore $(3k+1)/2$ must be evenly divisible by 2. ∎

Corollary 3.1.8. *For all $k > 1$, if $k \bmod 4 = 1$, then $\sigma(k) = 2$.*

Proof: Suppose $k > 1$. Then $(3k+1)/4 < k$. By the above lemma, $T(k) = (3k+1)/2$. But $(3k+1)/2 > k$ and $(3k+1)/2$ is even, so $T^{(2)}(k) = (3k+1)/4$. So by definition, the stopping time of k is 2. ∎

Sad to say, I never used the above idea in my search for peaks in stopping time.

In contrast to the above cutoffs, which rely on what happens to a number when it is used as input to iterations of H or T there are other *a priori* cutoffs that rely on how a number can result from (smaller) numbers in the course of iterating H or T. The idea of the following lemmas was brought to my attention by Mike Vermeulen. It is related to the *Collatz graph* discussed in [Lag85].

Lemma 3.1.9. *Let j and k be given so that $0 < j < k$. If there is some $m > 0$ such that $H^{(m)}(j) = k$, then k cannot be a peak in* steps *or* max_value.

Proof: Since the steps taken by k are the same as those taken by j after m initial steps, $steps(j) = m + steps(k)$ and $max\_value(j) \geq max\_value(k)$. ∎

Lemma 3.1.10. *Let j and k be given so that $0 < j < k$. If there is some $m > 0$ such that $T^{(m)}(j) = k$, then k cannot be a peak in stopping time, total stopping time, or* alt_max_value.

Proof: The proof is the same as the proof of the previous lemma. Since the steps taken by k are the same as those taken by j after m initial steps, $\sigma(j) = m + \sigma(k)$, $\sigma_\infty(j) = m + \sigma_\infty(k)$, and $alt\_max\_value(j) \geq alt_m ax\_value(k)$. ∎

The most important practical example of this kind of *a priori* cutoff is that if $k \bmod 6 = 5$, then k cannot be a peak in any of the derived measures mentioned in the lemmas above. This is because if $k \bmod 6 = 5$, then k lies on the sequence of iterates of $(2k-1)/3$, which is smaller than k. Indeed the iterates of H first multiply $(2k-1)/3$ by 3 and add 1, obtaining $2k$, and then divide $2k$ by 2 obtaining k. This result, due to Mike Vermeulen [Ver86a], is proved in the following lemma.

Lemma 3.1.11. *Let $k > 0$. If $k \bmod 6 = 5$, then $T((2k-1/3)) = k$ and $H^{(2)}((2k-1)/3) = k$.*

Proof: Suppose $k > 0$ and $k \bmod 6 = 5$. First we note that $2k-1$ is divisible by 3:

$$\begin{aligned} k \bmod 6 = 5 &\Rightarrow 2k \bmod 6 = 4 && (3.8)\\ &\Rightarrow 2k \bmod 3 = 1 && (3.9)\\ &\Rightarrow (2k-1) \bmod 3 = 0 && (3.10)\end{aligned}$$

Note that $(2k-1)/3$ is odd:

$$\begin{aligned} k \bmod 6 = 5 &\Rightarrow 2k \bmod 6 = 4 && (3.11)\\ &\Rightarrow (2k-1) \bmod 6 = 3 && (3.12)\\ &\Rightarrow (2k-1)/3 \bmod 2 = 1 && (3.13)\end{aligned}$$

The last implication above follows because there is some integer q such that:

$$\begin{aligned} (2k-1) = 6q+3 &\Rightarrow (2k-1) = 3(2q)+3 && (3.14)\\ &\Rightarrow (2k-1)/3 = 2q+1 && (3.15)\end{aligned}$$

Therefore, according to the definitions of T and H,

$$\begin{aligned} T((2k-1)/3) &= k && (3.16)\\ H((2k-1)/3) &= 2k && (3.17)\\ H^{(2)}((2k-1)/3) &= k && (3.18)\end{aligned}$$

∎

Corollary 3.1.12. *Let $k > 0$. If $k \bmod 6 = 5$, then k cannot be a peak in* steps, max_value, *stopping time, total stopping time, or* alt_max_value. ∎

The interested reader might see what happens when $k \bmod 18 = 13$. This idea can be carried as far as one desires. For example, one could keep a table of which numbers modulo 216 cannot be peaks in *max_value* or *steps*, and a counter that gives the value of the current iterate modulo 216. One could then use the table to avoid testing numbers that have no hope of being peaks. In the extreme, one can organize the entire search by constructing the Collatz graph, but the space requirements become prohibitive.

Other *a priori* cutoffs are discussed below, after the introduction of composite polynomials.

3.1.2 A Posteriori Cutoffs in Max_value

When iterating H to search for peaks in *max_value*, one has to check periodically to see if the values produced are greater than the value of the previous iterate (or than the value of the previous peak). However, these comparisons are fairly expensive. One way to reduce the cost of these comparisons is to stop making them after a certain point. It should be obvious

that one does not have to make a comparison after dividing by 2, since it becomes smaller than it was before. Neither does one have to make a comparison after every $3n+1$ step, but only until the iterates have fallen below the initial value (or stopped), due to the following result.

Lemma 3.1.13. *Let $k > 0$ be a peak in* max_value. *If for some $m > 0$, $H^{(m)}(k) < k$, then $max\_value(k) = \max\{H^{(i)}(k) \mid 0 \leq i \leq m\}$.*

Proof: Let $m > 0$, be such that $H^{(m)}(k) = j < k$. Since $j < k$, $max\_value(j) < max\_value(k)$, because k is a peak. Since after this point the sequence of iterates of k is the same as that taken by j, it cannot be the case that more iterations will reach or exceed the maximum value obtained up to this point. ∎

The way this lemma is used in an *a posteriori* cutoff is to stop making comparisons for purposes of finding a peak in *max_value* after the sequence of iterates falls below the initial input number, k. Note that the lemma depends on k being a peak in *max_value*. If this is not the case, the maximum value may be obtained after the value of an iterate falls below its starting value. An example is the number 55, which reaches a value of 376 before it first falls below 55 (to 47). It then goes on to reach a maximum value of 9,232. However, this is only a problem if the search guardian thinks that such a number really *is* a peak in *max_value*. It will be a problem because the guardian will not obtain the correct maximum value for the number, yet it will report to the coordinator that the number is a peak in *max_value*. The coordinator will then check the report and find it to be in error (which causes an unhandled exception in the search guardian). However, this problem never causes trouble if each search guardian knows about a peak in *max_value* that is reasonably close to the interval that it is searching. In practice this is only a problem if the search guardian does not know any peaks in *max_value* other than 1, which is one reason why the search cannot easily be partitioned near 1.

3.1.3 A Posteriori Cutoffs in Steps

After the value of the an iterate of H has fallen below the input value, the search is only concerned with whether or not the input number is a peak in *steps* (or total stopping time). In such cases the cost is running the steps of the hailstone algorithm and checking the value of the iterates to see if the algorithm should terminate. The following lemma and its practical importance in *a posteriori* cutoffs was brought to my attention by Mike Vermeulen.

Lemma 3.1.14. *For all $k > 0$, if $H^{(p)} = n \leq j$, where j is a peak in* steps, *then $steps(k) \leq p + steps(j)$*

Proof: If $n = j$, then $steps(k) = p + steps(j)$. If $n < j$, then, since j is a peak in *steps*, $steps(n) < steps(j)$, by the definition of a peak. ∎

In practice, the above lemma is used as follows. Let the initial value be k. After a step where H divides the current iterate value by two, one finds (if possible) the largest peak, j, in *steps* such that the current iterate's value is no greater than j, and uses the lemma above to bound $steps(k)$. If this bound on $steps(k)$ indicates that k is not a peak in steps, then k can be dismissed as far as *steps* is concerned.

The importance of this *a posteriori* cutoff is the empirical observation that the cutoff allows the search for peaks in *steps* to be cut off, on the average, after a constant number of steps. This seems to be true in any sufficiently large interval, provided that all but one or two peaks in *steps* less than the interval are known. That is, if one knows all the peaks in steps up to j, and if j is sufficiently large, then over the interval from $j+1$ to $2j$ one should always be able to cut off the search after an average of so many steps, independent of the value of j. I have instrumented a program and shown that one can cut off the search in steps after about 15 steps, on the average.[1] Considering that the number of steps taken by a number goes up logarithmically with the input number, this cutoff makes sense as soon as the average (odd) number starts taking more than 15 steps worth of time plus the time required to see if the search can be cut off.

3.2 Faster Iteration Algorithms

Even with the results above, there are still infinitely many numbers that have to be run through at least part of the hailstone algorithm of Figure 3.1, which computes the iterates of H, or the similar algorithm that computes the iterates of T. Thus the problem of running the hailstone algorithm efficiently turns out to be important in extending the search for peaks very far. The problem considered in this section is finding an equivalent algorithm that can be executed in less time. The focus in this section is on the hailstone algorithm, and the search for peaks in *steps* and *max_value*.

Figure 3.1: The hailstone algorithm, which computes iterates of H.

```
% input: an integer n > 0
% output: number of steps and max_value reached
steps: int := 0
max_value: bigint := n
while n ≠ 1 do
        if (n mod 2) = 0
        then n := n / 2
        else n := 3×n + 1
             max_value := max(max_value, n)
        end
        steps := steps + 1
end
```

[1] I recorded data for the 100,000 odd numbers in the interval from 17,828,259,369 to 17,828,459,369. Note that 17,828,259,369 is a peak in *steps*. In this interval the average number of steps is 276.

Figure 3.2: Hailstone algorithm using *make_odd*.

```
% input: an odd integer n > 0
while n ≠ 1 do
        % n is odd
        n := 3×n + 1
        % n is even
        % check for max_value
        n, p := make_odd(n)
        % the number of steps taken this time around the loop is p+1
        % check for a posteriori cutoffs
end
```

3.2.1 Make_odd

Division by 2 is best implemented by shifting in a binary representation. Also, shifting a number by several bit positions is roughly as fast as shifting a number by one bit position. Thus, one idea for making a faster algorithm is to replace the division by 2 step in the hailstone algorithm by a step that shifts the input as many bits as necessary in order to make it odd. How effective will this be? If the value of the hailstone algorithm's variable n were uniformly distributed among the even integers by the $3n+1$ step, then half of the time n would not be divisible by 4, and one fourth of the time n would not be divisible by 8, and so on. Thus the expected number of bit positions that an even number would be shifted is

$$1/2 + 2(1/4) + 3(1/8) + 4(1/16) + \cdots = \sum_{i=1}^{\infty} \frac{i}{2^i} \tag{3.19}$$

$$= 2 \tag{3.20}$$

Thus on the average, shifting n by as many bits as necessary to make it odd does the work of two divisions by 2 and should take the same amount of time.

Furthermore, a nice property of shifting n so that it is odd is that one no longer has to check to see whether n is odd or even, because one can write the hailstone algorithm (for odd inputs) as in Figure 3.2. In the figure, the procedure *make_odd* returns the new value of n and the number of bit positions that the old value of n had to be shifted to make it odd. Another pleasing property of this hailstone algorithm is that one can check for cutoffs when n is as low as it can be before going up again, this means that one spends less time checking for cutoffs, on the average.

3.2.2 Composite Polynomials

A more efficient hailstone algorithm is the result of Mike Vermeulen's efforts. The standard hailstone algorithm looks at the last bit of the value of the variable n to decide what step to take. By looking at the last m bits of the binary representation of n, one can decide what the next several steps that will be taken are, combine all these steps into a polynomial,

Figure 3.3: Hailstone algorithm using composite polynomials.

```
% input: an integer n > 0
while n ≠ 1 do
        p, s := mBitPoly(n mod 2^m)
        n := polyEval(p, n)
        % the number of steps taken this time around the loop is s
        % check for a posteriori cutoffs
end
```

and then do the work of all those steps by evaluating the polynomial at the value of n. An algorithm that uses this idea for computing the iterates of H is shown in simplified form in Figure 3.3. In the figure, *mBitPoly* returns both a polynomial and the number of steps that the polynomial represents. Checking for *max_value* is described below.

There are two strategies for expressing the polynomial.

One strategy is to obtain a polynomial of the form:

$$\frac{3^k x + z}{2^m}$$

which is equivalent to the sequence of $k+m$ steps taken. This *standard polynomial*, represents k steps of the form $3n+1$ and m divisions by 2. (It will be shown below why the number of divisions by 2 is equal to the number of bits considered.) This polynomial can be evaluated by multiplying by the appropriate power of 3, adding in the appropriate integer z and then shifting by the appropriate power of 2. Since the shifting is done last, there is an opportunity to use the *make_odd* trick, ending up with an odd number.

For each n, the standard m bit polynomial for n will be written $Spoly_m(n)$.

The strategy that Mike Vermeulen uses is to obtain a polynomial of the form:

$$\left\lfloor \frac{x}{2^m} \right\rfloor 3^k + y$$

This *Vermeulen polynomial* also represents k steps of the form $3n+1$ and m divisions by 2. Vermeulen polynomials will yield an answer equivalent to the normal sequence of steps when evaluated in the following manner:

1. divide by 2^m and truncate, that is, shift the binary representation right by m bits,
2. multiply the result (the most significant part of n) by the appropriate power of 3, and
3. add y.

The idea is that a Vermeulen polynomial represents the effect of the steps on the most significant part of n only. The number y represents the overflow from the least significant m bits. This strategy has the advantage of dealing with smaller numbers during the evaluation of the polynomial. By contrast, the evaluation of a standard polynomial is more likely to

incur the cost of creating larger representations intermediate results. Furthermore, when evaluating a Vermeulen polynomial in the case where $3^k < 2^m$, one can be sure that there will be no overflow in computing the intermediate result, because the shifting (division) is done first.

For each n, the m bit Vermeulen polynomial for n will be written $Vpoly_m(n)$.

To explain how these polynomials are generated, consider the following examples, adapted from a mail message [Ver86b]. Items to the right of the dot (.) in last bits do not affect the other bits, and items to the left of the dot do. The following is the generation of an 8 bit Vermeulen polynomial, $Vpoly_8(3)$.

```
Max Last bits          N      Partial poly   Next step
   .00000011           3        X              *3
   .00001010          10        3X+1           /2
   .0000101            5        3(X/2)+2       *3
 * .0010000           16        9(X/2)+7       /16
   .001                1        9(X/32)+1      *3
   .100                4        27(X/32)+4     /4
   .1                  1        27(X/128)+1    *3
 10.0                  4        81(X/128)+2    /2
 10.                   2        81(X/256)+2
```

Notice that division by 2 removes one bit from the right, this corresponds to moving in bits from the left that are unknown. The process of computing partial polynomials stops when all the known bits are shifted out, that is, when m divisions by 2 have been performed. The row marked with the asterisk indicates the partial polynomial which produces the largest intermediate result. If one is checking for peaks in *max_value*, then the evaluation must be broken into three steps: evaluating the partial polynomial at this point, checking for a peak in *max_value*, and then evaluating the rest of the polynomial. For comparison, $Spoly_8(3)$ is $(81x + 269)/256$.

As another example, $Vpoly_8(27)$ is $\lfloor x/256 \rfloor 2187 + 242$; for comparison $Spoly_8(27)$ is $(2187x + 2903)/256$.

Although the composite polynomial idea leads to a hailstone algorithm that is perhaps an order of magnitude faster than the algorithm in Figure 3.2, I have no proof that this approach yields an optimal algorithm.

The practical drawback to such an algorithm is its complexity of implementation. One needs automated tools for generating the polynominals. Furthermore, it is difficult writing the code to check for peaks in *max_value* or stopping time. This implementation difficulty makes the results of the algorithm less reliable (as the implementation is more difficult to verify).

However, even if composite polynomials are not used in the iteration algorithm, they can be used to generate *a priori* cutoffs.

3.2.3 A Priori Cutoffs based on Composite Polynomials

Composite polynomials lead to new strategies for cutoffs [Ver86b].

Steps

Mike Vermeulen cuts off the search for peaks in *steps a priori* as follows.

Lemma 3.2.1. *Let $m > 0$ be given. Let $0 < r_1 < r_2 < 2^m$, $Vpoly_m(r_1) = Vpoly(r_2)$. Then there is no $k \geq 2^m + r_1$ such that $k \bmod 2^m = r_2$ and k is a peak in* steps.

Proof: Let $k \geq 2^m + r_1$ be such that $k \bmod 2^m = r_2$. Let b be the largest number such that $b < k$ and $b \bmod 2^m = r_1$. Since the polynomials $Vpoly_m(r_1)$ and $Vpoly_m(r_2)$ are identical they represent the same number of steps. After these steps, b and k take the same steps since $Vpoly_m(r_1)(b) = Vpoly_m(r_2)(k)$. So k cannot be a peak, because $b < k$ and $steps(b) = steps(k)$. ∎

Max_Value

Mike Vermeulen cuts off the search for peaks in *max_value a priori* if the m bit Vermeulen polynomial has as the coefficient of x a term that is less than unity. For example, using 8 bit polynomials, he does not look for peaks in values when the last 8 bits are ".00000011," because $Vpoly_8(3) = \lfloor x/256 \rfloor 81 + 2$ and $81/256 < 1$. Another example: he *does* look for peaks in values when the last 8 bits are ".00011011" because $Vpoly_8(27) = \lfloor x/256 \rfloor 2187 + 242$.

This kind of cutoff is formalized using the notion of the "largest" partial polynomial. A *partial polynomial* is a polynomial incorporating the first $s \geq 0$ steps of the hailstone algorithm, as predicted (see above) from the least significant m bits. We say that $p \geq q$ for m bit partial polynomials if there is some $N > 0$ such that, for all $n > N$, $p(n) \geq q(n)$.

Let $LVpoly_m(r)$ be the largest m bit partial Vermeulen polynomial predicted for r. Similarly, let $LSpoly_m(r)$ be the largest m bit partial standard polynomial predicted for r.

Lemma 3.2.2. *Let $m > 0$ be given. Let $poly_m(r)$ and $Lpoly_m(r)$ denote either the standard or Vermeulen m bit polynomial and largest m bit partial polynomial predicted for r.*

There is some $N > 0$ such that for all $k > N$, the following hold. If $k \bmod 2^m = r$ and

1. *$poly_m(r)(k) < k$, and*
2. *there is some* max_value *peak $j < k$ such that $Lpoly_m(r)(k) \leq max\_value(j)$,*

then k is not a peak in max_value.

Proof: Choose N such that for all $k > N$ and for all predicted m bit partial polynomials, p and q, $p \geq q$ implies $p(k) \geq q(k)$. Now, given $k > N$, the highest value the hailstone algorithm's variable n reaches before falling below its initial value, k, is $Lpoly(r)(k)$, where $r = k \bmod 2^m$. This is because $poly(r)(k) < k$, and so n falls below k (after the number of steps represented by the polynomial) and because by construction $Lpoly(r)(k)$ is the highest value that n attains (in these first steps). Thus by Lemma 3.1.13, if k is a peak in *max_value*, then

$$Lpoly(r)(k) = max\_value(k) < max\_value(j)$$

which is a contradiction. So k cannot be a peak in *max_value*. ∎

Note that the above lemma is independent of the kind of polynomial used. In practice, the choice of N is not a problem, because the input numbers soon dwarf the coefficients of the composite polynomials.

The above lemma allows one to set up a table that tells which numbers modulo 2^m need to be checked for peaks in *max_value*. For each $0 < r < 2^m$, one must check the two conditions of the above lemma. For sufficiently large numbers, one can check the first condition of the lemma above *a priori*.

One can also satisfy the second condition of the lemma *a priori* over a certain interval. Suppose N is chosen to be the least number to satisfy the lemma and such that the first condition of the lemma can be checked *a priori*. Suppose the largest known peak in *max_value*, call it j, is such that $j > N$. Finally, suppose that there is a rational number $R > 0$ such that, for all $k > N$ and for all $0 \leq r \leq 2^m$ such that $poly_m(r)(k) < k$:

$$LVpoly_m(r)(k) < R \cdot k.$$

Then in the interval from N to $max\_value(j)/R$ the second condition of the lemma can be checked *a priori*. In fact, in such an interval, the second condition of the lemma is satisfied whenever the first condition of the lemma is satisfied.

Thus the practical justification for setting up a table of cutoffs as described above is that, for sufficiently large input numbers, the maximum value of the largest peak is many orders of magnitude greater than the inputs numbers and the coefficients of the Vermeulen polynomial cannot be very large. In fact the following result holds.

Lemma 3.2.3. *Let $m > 1$ be given. For all $0 < r < 2^m$, if there is some M_r such that for all $k > M_r$, $k \bmod 2^m = r$ implies that $Vpoly_m(r)(k) < k$, then the coefficient of x in $LVpoly_m(r)$ is at most $3^j/2^{(j-1)}$, where j is the largest integer such that $3^j < 2^m$.*

Proof: Every step in the hailstone algorithm of the form $3n+1$ is followed by a step that divides by 2. Thus the coefficient of x in the partial Vermeulen polynomial is made as large as possible when it is of the form $3^j/2^{(j-1)}$. However, because the final polynomial is of the form $3^{j+p}/2^m + y$, for some $p \geq 0$, 3^j must be less than 2^m if the Vermeulen polynomial is to be such that $Vpoly_m(r)(k) < k$, for all sufficiently large $k \bmod 2^m = r$. ∎

As an example, for 8 bit Vermeulen polynomials, the coefficient can be at most

$$3^5/2^4 = 243/16 = 15.1875.$$

Thus, as long as the is the largest peak j in *max_value* is at least 16 times the input numbers being checked, then for all sufficiently large k:

$$Vpoly_8(r)(k) < k \Rightarrow LVpoly_8(r)(k) \leq max\_value(j).$$

This condition seems to be met for all numbers past 7.

3.2.4 Hackery

The search must deal with large precision integers. This is obvious as the input numbers themselves become larger than the single precision integers implemented in Argus. However,

the peaks in *max_value* exceed the limitations of the built-in **int** type more quickly. For example

$$max\_value(77{,}671) \quad = \quad 1{,}570{,}824{,}736. \tag{3.21}$$

The implementation of multiple precision integers is thus the foundation upon which the algorithms discussed above run. Thus the efficiency of the routines for multiplication, and so on have a great effect on the efficiency of the search as a whole.

The basic hack is using a binary representation for multiple precision integers and writing routines to return $3n + 1$, etc. in assembly language to take advantage of the machine arithmetic. Argus's built-in type **int** is quite slow for two reasons:

1. **int**'s arithmetic operations are defined to check for overflow, and

2. the representation chosen for **int** on the vax architecture makes checking for overflow inefficient.

Essentially, after adding two **int**s, Argus must add them again and then check for overflow, taking three instructions overall for an addition. In assembly language, with a suitable representation invariant, one can add numbers much faster. Similar remarks apply to multiplication and division.

Chapter 4

Conclusions

4.1 Suitability of Argus

Argus proved to be a good implementation language because it provides good support for making progress in long running computations. More specifically:

- Argus stable storage provides a convenient way to checkpoint intermediate states.
- Actions insure that checkpoints only record consistent states. This allows invariants to be preserved across crashes.
- Handlers provide a convenient way to coordinate and monitor the state of the computation. In particular, the ability to separate the code that monitors the computation from the code performing the actual computation makes program design easier, and allows performance tuning of just the code that performs the computation.
- Guardians recover from machine crashes automatically.

Using a simple guardian with background code that periodically saves state in stable storage is a general paradigm for long running computations. I was able to write a simple program the perform a long-running search on one machine in a matter of hours.

Adding coordination of multiple guardians, however, took several days.

4.2 Lessons Learned about Argus

This section discusses observations about using Argus that I learned while writing the hailstone system. Observations relating to the efficiency of the resulting program, however, are discussed above.

A guardian's state variables are accessible throughout the guardian definition. There is thus a tendency to not pass the objects they refer to routines that are internal to the guardian, causing modularity problems. The problem is that it can be difficult to extract a routine from a guardian, if you want to compile and link it separately.

I never needed recover code. Initializations of the volatile variables always sufficed for the hailstone system.

I always created guardians by passing them an initial state, even though there is a well defined distinguished initial state. This is partly because I usually wanted to continue from where the search left off, but also because it is easier to have just one way of creating the hailstone system.

Defining stable variables holding integers where the value can change is tricky. I used an atomic record containing an integer. The problem is that one easily thinks of the variable as an integer variable, whereas it is actually an atomic record variable.

Guardians have invariants that describe their state, especially the relationship of the volatile variables to the stable variables. These invariants hold for action-consistent snapshots. I comment each guardian with an invariant. I also noticed that the invariants for guardians tended to be much longer and more involved than those for clusters. But using abstract types instead of using just atomic arrays and so on helps simplify the invariants.

I have an implementation of immutable and atomic integers whose rep is mutable and nonatomic (an array). This works because I never modify the array after passing out a pointer to it, although I never understood (or thought about) why it was resilient until I talked to Elliot Kolodner.

4.3 Acknowledgements

The work described in this report was done while the author was a graduate student at MIT, and hence was supported indirectly by MIT, the Laboratory for Computer Science, and in particular by Barbara Liskov.

I am grateful to Mike Vermeulen for many optimizations and countless discussions about the hailstone search. While the Argus group has benefited from having a running application, Paul Johnson has been helpful beyond the call of duty in fixing problems with the Argus system (and my programs!) in a timely manner. Paul has also helped me with assembly language programming. I also thank all the members of the Argus group for putting up with the hailstone system with such good grace and I thank them for logging out at the end of the day.

Thanks also to Kelvin Nilsen who helped correct the final draft of this report.

Appendix A

Tables of Peaks

This appendix contains tables of results from the various search programs.

A.1 Peaks in more than one Derived Measure

A few numbers have been found to be peaks in several derived measures. In a sense, these are the most interesting numbers I found.

Tables A.5, A.6, and A.7 list the peaks in both *steps* and total stopping time; these are known to be identical up to 12.3 billion (12.3×10^9).

Table A.1 lists numbers that are peaks in other combinations of derived measures; each entry is filled in if the number is a peak in the corresponding function and left empty otherwise.

Table A.1: Peaks in more than one derived measure.

n	steps(n)	$\sigma(n)$	max_value(n)
1	0	0	1
2	1	1	2
3	7	4	16
7	16	7	52
27	111	59	9,232
703	170	81	250,504
26,623	307		106,358,020
270,271		164	24,648,077,896
626,331	508	176	
63,728,127	949	376	
12,235,060,455	1,184	547	

A.2 Maximum Values

The peaks in *max_value* are listed in Tables A.2 and A.3. The peaks up to 100 billion (100×10^9) have been verified by two programs: mine and Mike Vermeulen's. Mike Vermeulen is responsible for all the peaks above 100 billion. There is some question whether the peaks in *max_value* over 100 billion listed below can be trusted, due to a bug in Mike's program, but they seem genuine. The list of the peaks he has found is complete up to 1.711 trillion (1.711×10^{12}).

Of particular interest here are the peaks at n equal to 27, 6,631,675, and 319,804,831. Also listed in the tables is the ratio of the peak's maximum value reached to the previous maximum value reached (labeled "ratio") and the expansion factor (labeled $s(n)$); the expansion factor, $s(n)$, defined by max_value$(n)/(2n)$, is rounded to two significant digits.

Table A.2: Peaks in max_value up to n = 5,000,000.

n	$max\_value(n)$	ratio	$s(n)$
1	1		0.5
2	2	2.0	0.5
3	16	8.0	2.7
7	52	3.3	3.7
15	160	3.1	5.3
27	9,232	57.7	1.7×10^2
255	13,120	1.4	2.5×10^1
447	39,364	3.0	4.4×10^1
639	41,524	1.1	3.2×10^1
703	250,504	6.0	1.8×10^2
1,819	1,276,936	5.1	3.5×10^2
4,255	6,810,136	5.3	8.0×10^2
4,591	8,153,620	1.2	8.9×10^2
9,663	27,114,424	3.3	1.4×10^3
20,895	50,143,264	1.8	1.2×10^3
26,623	106,358,020	2.1	2.0×10^3
31,911	121,012,864	1.1	1.9×10^3
60,975	593,279,152	4.9	4.9×10^3
77,671	1,570,824,736	2.6	1.0×10^4
113,383	2,482,111,348	1.6	1.1×10^4
138,367	2,798,323,360	1.1	1.0×10^4
159,487	17,202,377,752	6.1	5.4×10^4
270,271	24,648,077,896	1.4	4.6×10^4
665,215	52,483,285,312	2.1	3.9×10^4
704,511	56,991,483,520	1.1	4.0×10^4
1,042,431	90,239,155,648	1.6	4.3×10^4
1,212,415	139,646,736,808	1.5	5.8×10^4
1,441,407	151,629,574,372	1.1	5.3×10^4
1,875,711	155,904,349,696	1.0	4.2×10^4
1,988,859	156,914,378,224	1.0	3.9×10^4
2,643,183	190,459,818,484	1.2	3.6×10^4
2,684,647	352,617,812,944	1.9	6.6×10^4
3,041,127	622,717,901,620	1.8	1.0×10^5
3,873,535	858,555,169,576	1.4	1.1×10^5
4,637,979	1,318,802,294,932	1.5	1.4×10^5
5,656,191	2,412,493,616,608	1.8	2.1×10^5

Table A.3: Peaks in *max_value* from n = 5,000,000 to 1,711,000,000,000.

n	$max\_value(n)$	ratio	$s(n)$
5,656,191	2,412,493,616,608	1.8	2.1×10^5
6,416,623	4,799,996,945,368	2.0	3.7×10^5
6,631,675	60,342,610,919,632	12.6	4.5×10^6
19,638,399	306,296,925,203,752	5.1	7.7×10^6
38,595,583	474,637,698,851,092	1.5	6.1×10^6
80,049,391	2,185,143,829,170,100	4.6	1.4×10^7
120,080,895	3,277,901,576,118,580	1.5	1.4×10^7
210,964,383	6,404,797,161,121,264	2.0	1.5×10^7
319,804,831	1,414,236,446,719,942,480	220.8	2.2×10^9
1,410,123,943	7,125,885,122,794,452,160	5.0	2.5×10^9
8,528,817,511	18,144,594,937,356,598,024	2.5	1.1×10^9
12,327,829,503	20,722,398,914,405,051,728	1.1	8.4×10^8
23,035,537,407	68,838,156,641,548,227,040	3.3	1.5×10^9
45,871,962,271	82,341,648,902,022,834,004	1.2	9.0×10^8
51,739,336,447	114,639,617,141,613,998,440	1.4	1.1×10^9
59,152,641,055	151,499,365,062,390,201,544	1.3	1.3×10^9
59,436,135,663	205,736,389,371,841,852,168	1.4	1.7×10^9
70,141,259,775	420,967,113,788,389,829,704	2.0	3.0×10^9
77,566,362,559	916,613,029,076,867,799,856	2.2	5.9×10^9
110,243,094,271	1,372,453,649,566,268,380,360	1.5	6.2×10^9
204,430,613,247	1,415,260,793,009,654,991,088	1.0	3.4×10^9
231,913,730,799	2,190,343,823,882,874,513,556	1.5	4.7×10^9
272,025,660,543	21,948,483,635,670,417,963,748	10.0	4.0×10^{10}
446,559,217,279	39,533,276,910,778,060,381,072	1.8	4.4×10^{10}
567,839,862,631	100,540,173,225,585,986,235,988	2.5	8.8×10^{10}
871,673,828,443	400,558,740,821,250,122,033,728	4.0	2.3×10^{11}

A.3 Steps and Total Stopping Time

Peaks in *steps* are listed in Tables A.5, A.6, and A.7. The peaks up to 100 billion (100×10^9) have been verified by two programs: mine and Mike Vermeulen's. Mike Vermeulen is responsible for all the peaks above 100 billion, and his list is believed to be complete up to 1.711×10^{12}. It seems that every peak in *steps* is also a peak in total stopping time, σ_∞, although I have only verified this conjecture up to 12.3×10^9. That is the tables A.5, A.6, and A.7 also contain peaks in total stopping time, up to 12.3 billion.

Also listed in these tables are the difference between each peak's number of steps and the previous peak's number of steps, the value of the stopping time $\sigma(n)$, the value of $\sigma_\infty(n)$, and the maximum value reached.

Of particular interest here are the peaks at n equal to 27 and 63,728,127 which have large increments in the number of steps and the peaks that are simply twice the previous peak in steps, found in Table A.4.

Table A.4: Even Peaks in *steps* up to 1,711,000,000,000.

peak (n)	steps(n)
2	1
6	8
18	20
54	112
31,466,382	705
127,456,254	950
537,099,606	965
1,341,234,558	987
9,780,657,630	1,132
63,389,366,646	1,220
404,970,804,222	1,308

Table A.5: Peaks in *steps* up to n = 100,000.

n	$steps(n)$	diff.	$\sigma(n)$	$\sigma_\infty(n)$	$max\_value(n)$
1	0		0	0	1
2	1	1	1	1	2
3	7	6	4	5	16
6	8	1	1	6	16
7	16	8	7	11	52
9	19	3	2	13	52
18	20	1	1	14	52
25	23	3	2	16	88
27	111	88	59	70	9,232
54	112	1	1	71	9,232
73	115	3	2	73	9,232
97	118	3	2	75	9,232
129	121	3	2	77	9,232
171	124	3	5	79	9,232
231	127	3	12	81	9,232
313	130	3	2	83	9,232
327	143	7	21	91	9,232
649	144	1	2	92	9,232
703	170	26	81	108	250,504
871	178	8	35	113	190,996
1,161	181	3	2	115	190,996
2,223	182	1	8	116	250,504
2,463	208	26	21	132	250,504
2,919	216	8	26	137	250,504
3,711	237	21	37	150	481,624
6,171	261	24	58	165	975,400
10,971	267	6	8	169	975,400
13,255	275	8	8	174	497,176
17,647	278	3	73	176	11,003,416
23,529	281	3	2	178	11,003,416
26,623	307	26	65	194	106,358,020
34,239	310	3	92	196	18,976,192
35,655	323	13	135	204	41,163,712
52,527	339	16	18	214	106,358,020
77,031	350	11	89	221	21,933,016
106,239	353	3	97	223	104,674,192

Table A.6: Peaks in *steps* from n = 100,000 to n = 5,000,000,000.

n	$steps(n)$	diff.	$\sigma(n)$	$\sigma_\infty(n)$	$max\_value(n)$
106,239	353	3	97	223	104,674,192
142,587	374	21	24	236	593,279,152
156,159	382	8	37	241	41,163,712
216,367	385	3	83	243	11,843,332
230,631	442	57	73	278	76,778,008
410,011	448	6	75	282	76,778,008
511,935	469	21	16	295	76,778,008
626,331	508	39	176	319	7,222,283,188
837,799	524	16	105	329	2,974,984,576
1,117,065	527	3	2	331	2,974,984,576
1,501,353	530	3	2	333	90,239,155,648
1,723,519	556	26	176	349	46,571,871,940
2,298,025	559	3	2	351	46,571,871,940
3,064,033	562	3	2	353	46,571,871,940
3,542,887	583	21	180	366	294,475,592,320
3,732,423	596	13	8	374	294,475,592,320
5,649,499	612	16	116	384	1,017,886,660
6,649,279	664	52	146	416	15,208,728,208
8,400,511	685	21	214	429	159,424,614,880
11,200,681	688	3	2	431	159,424,614,880
14,934,241	691	3	2	433	159,424,614,880
15,733,191	704	13	8	441	159,424,614,880
31,466,382	705	1	1	442	159,424,614,880
36,791,535	744	39	34	466	159,424,614,880
63,728,127	949	205	376	592	966,616,035,460
127,456,254	950	1	1	593	966,616,035,460
169,941,673	953	3	2	595	966,616,035,460
226,588,897	956	3	2	597	966,616,035,460
268,549,803	964	8	5	602	966,616,035,460
537,099,606	965	1	1	603	966,616,035,460
670,617,279	986	21	18	616	966,616,035,460
1,341,234,558	987	1	1	617	966,616,035,460
1,412,987,847	1,000	13	8	625	966,616,035,460
1,674,652,263	1,008	8	13	630	966,616,035,460
2,610,744,987	1,050	42	46	656	966,616,035,460
4,578,853,915	1,087	37	81	679	966,616,035,460
4,890,328,815	1,131	44	135	706	319,497,287,463,520

Table A.7: Peaks in *steps* from n = 5,000,000,000 to 1,711,000,000,000.

n	$steps(n)$	diff.	$\sigma(n)$	$\sigma_\infty(n)$	$max\_value(n)$
4,890,328,815	1,131	44	135	706	319,497,287,463,520
9,780,657,630	1,132	2	1	707	319,497,287,463,520
12,212,032,815	1,153	21	15	720	319,497,287,463,520
12,235,060,455	1,184	31	547	739	1,037,298,361,093,936
13,371,194,527	1,210	26	62	755	319,497,287,463,520
17,828,259,369	1,213	3	2	757	319,497,287,463,520
31,694,683,323	1,219	6	7	761	319,497,287,463,520
63,389,366,646	1,220	1	1	762	319,497,287,463,520
75,128,138,247	1,228	8	7	767	319,497,287,463,520
133,561,134,663	1,234	6	10	771	319,497,287,463,520
158,294,678,119	1,242	8	15	776	319,497,287,463,520
202,485,402,111	1,307	65	270	816	2,662,567,439,048,656
404,970,804,222	1,308	1	1	817	2,662,567,439,048,656
426,635,908,975	1,321	13	40	825	2,662,567,439,048,656
568,847,878,633	1,324	3	2	827	2,662,567,439,048,656
674,190,078,379	1,332	8	5	832	2,662,567,439,048,656
881,715,740,415	1,335	3	329	834	5,234,135,688,127,384
989,345,275,647	1,348	13	165	842	1,219,624,271,099,764
1,122,382,791,663	1,356	8	16	847	2,662,567,439,048,656
1,444,338,092,271	1,408	52	202	879	1,219,624,271,099,764

A.4 Stopping Time

Peaks in stopping time, σ, are not necessarily peaks in *steps*, and conversely peaks in *steps* are not necessarily peaks in stopping time. The same remark applies to total stopping time.

Table A.8 lists the peaks in stopping time as found by my program. (These results have not been verified by another program.) This list is complete up to 1.043 trillion (1.043×10^{12}). The most interesting of these peaks is 12,235,060,455.

Also listed are the difference between each peak's stopping time and the stopping time of the previous peak (labeled "diff."), the value of *steps* for that peak, the total stopping time σ_∞, and the maximum value reached. Unlike the peaks in *steps*, the maximum values reached by these peaks rarely repeat.

Table A.8: Peaks in stopping time, σ, up to 1,043,000,000,000.

n	$\sigma(n)$	diff.	$steps(n)$	$\sigma_\infty(n)$	$max\_value(n)$
1	0		0	0	1
2	1	1	1	1	2
3	4	3	7	5	16
7	7	4	16	11	52
27	59	52	111	70	9,232
703	81	22	170	108	250,504
10,087	105	24	223	142	2,484,916
35,655	135	30	323	204	41,163,712
270,271	164	29	406	256	24,648,077,896
362,343	165	1	360	228	565,335,124
381,727	173	8	373	236	565,335,124
626,331	176	3	508	319	7,222,283,188
1,027,431	183	7	377	239	17,808,240,724
1,126,015	224	41	527	331	90,239,155,648
8,088,063	246	22	566	356	16,155,154,672
13,421,671	287	41	608	382	1,591,706,254,336
20,638,335	292	5	694	435	89,243,211,616
26,716,671	298	6	658	413	3,696,858,621,088
56,924,955	308	10	742	465	7,209,046,267,252
63,728,127	376	68	949	592	966,616,035,460
217,740,015	395	19	793	395	2,516,021,527,120
1,200,991,791	398	3	873	547	35,681,506,677,556
1,827,397,567	433	25	928	581	118,736,698,851,769,012
2,788,008,987	447	14	944	591	81,887,769,175,732
12,235,060,455	547	100	1,184	739	1,037,298,361,093,936
898,696,369,947	550	3	1,136	712	791,612,079,014,220,715,456

Appendix B

History of the Search Programs

This appendix contains a history of the search programs that were run at MIT during my time as a graduate student. The entries in this appendix were taken from a file that I maintained while the search was running, except that the last two entries were made as this report was being written (since at the time I was in the midst of finishing my dissertation).

There were three searches programs in all. The first was a search for peaks in *steps* and *max_value*. The second was a search for peaks in total stopping time (σ_∞). The third was a search for peaks in stopping time σ.

The entries all follow the same format. The date, followed by the number reached in the search is reported first. This is followed by a comment.

In the comments, the name "Paul" refers to Paul Johnson, and "Mike" means Mike Vermeulen.

The names "rinso," "prj," "duz," and so on are the names of particular machines at MIT. Their characteristics are listed in Table B.1. The idle time on six micro-vax II's (VAX stations) and two VAX 3200s (the configuration used for the third search for a long time) was generally enough to search an interval of size 95 billion for peaks in stopping time (σ) in a month. The amount searched would be more or less, depending on the time of year (e.g., during holidays no one would be logged in), the number and type of the machines involved, and the efficiency of the algorithms used. At the end, each VAX station could search an interval with a size of approximately 264,000 numbers for peaks in σ in a minute (of real time). Other comments on the speed of the search are found below.

Mike Vermeulen has searched the interval from 1 to at least 1.711 trillion (1.711×10^{12}) for peaks in *steps* and *max_value*. His search is now also defunct. Until sometime in 1989 he was using a HP 9000/350 computer[1] and a program written in C and assembly language, which was running at HP Labs in Fort Collins, Colorado since December 1986.

B.1 The First Search: Peaks in Steps and Max_Value

The first search began in May 1986 and searched for peaks in *steps* and *max_value*. It was ended when it reached 100,085,608,537, over a year later. The initial version of the program is described in an internal note at MIT [Lea86]. The more-or-less final version of the system

[1] Until about August 1988, Mike used 7 machines.

Table B.1: Machines used in the search.

name	machine type
rinso	DEC VAX 750
duz	DEC VAX 750
prj,pm-prj	DEC micro-vax II
electron	DEC micro-vax II
graviton	DEC micro-vax II
proton	DEC micro-vax II
neutron	DEC micro-vax II
positron	DEC micro-vax II
photon	DEC micro-vax II
muon	DEC micro-vax II
kaon	DEC VAX 3200
meson	DEC VAX 3200

was described in a similar internal note [Lea87], which has been updated in the main body of this report.

8 May 1986 77,671 First guardian written. The search started using only rinso.

15 May 1986 3,272,371 Started using 2 machines. Rinso and prj.

20 May 1986 8,210,229 Rinso catches up to prj by reaching 5,004,427. The search slows down prj, however, so it's back to just rinso.

6 June 1986 14,024,621 Duz is found to be faster than rinso (probably because its users are on vacation). The search is moved to duz.

20 June 1986 19,686,965 Complaints surface that even at lowest priority, the search is slowing down duz. The guardians are recoded so that they go to sleep when someone logs in. The search is put up on all 5 machines (rinso at 19,686,965, bold at 22,000,001, fab at 25,000,001, duz at 30,000,001, and prj at 40,000,001).

30 June 1986 31,466,382 Found (even) peak in steps at 31,466,382 almost in time for grads meeting.

8 July 1986 39,000,001 Mike Vermeulen comes up with an optimization of the algorithm to cut off the search if the starting number cannot produce a new peak in steps. This is coded, tested, and the guardians are started up again, with provisions for updating a list of peaks in steps (the cutoffs list) as they run. Managing the guardians is becoming more complex.

23 July 1986 175,881,671 Guardians are given slices 200,000,000 apart so that there will be no overlap during Gary's vacation.

21 August 1986 259,618,047 Hailstone coordinator guardian tested and put into use. Search continues on 5 machines.

26 August 1986 321,324,733 Spurred on by Mike Vermeulen's C version of the hailstone search, various low-level optimizations are made. The search guardians on a micro vax are able to search an interval of about 6000 in a minute. Mike's version reaches 1 billion.

3 Sept 1986 537,099,606 Finding another double peak in steps prompts rewriting of search guardians to only search the odd numbers, while the coordinator predicts the double peaks in steps.

12 Sept 1986 718,823,481 It becomes obvious that the slice from 718,823,481 to 719,823,481 has been forgotten. Some guardian was assigned it but never reported it to the coordinator. It will have to be re-searched.

15 Sept 1986 718,823,481 Bold, duz, and fab are taken away. The unreported slice is still unreported. Restarting the search on rinso and prj uncovers a harmless bug in the coordinator's predicting of double peaks in steps, which is fixed.

16 Sept 1986 841,830,793 Rinso and prj have filled in the gaps in the search, but the Argus system starts to have problems with stable storage and bringing up the new microvaxes. Missing slice problems appear because of some Argus problems and so the search is soon halted.

23 Sept 1986 849,310,791 Argus is running on all the microvaxes, and the search is started on the 4 microvaxes (electron, neutron, proton, pm-prj) and rinso.

27 Sept 1986 891,105,783 Designed, coded, tested and installed new version of unsignedints specially designed for the hailstone search problem. Seems to have speeded up the search by about a factor of 3. Prj and rinso continually have problems with sendmails running to the exclusion of the search guardians.

28 Sept 1986 1,033,537,851 Search passes 1 billion. Parameters adjusted so that coordinator hands out slices of size 5,000,000 and search guardians checkpoint after every 100,000 searches.

30 Sept 1986 1,101,537,851 Some fine tuning of the new representation. Measurements indicate that the search guardians on a microvax can search an interval of about 20000 in a minute, a speedup of about a factor of 3.

6 Oct. 1986 1,694,000,001 More fine tuning of the new representation. Measurements indicate the rate on a microvax is an interval of about 25,000 in a minute.

28 Oct. 1986 3,886,291,371 Another round of representations and tuning by rewriting part of it in assembly language. Rate is an interval of about 42500/minute on a microvax (neutron) in the interval from 3.89 to 3.93 billion. This is a speedup of a factor of 2 or more from the previous version.

28 Nov. 1986 11,378,406,329 Three new microvaxes are installed. A slight improvement is made in the code for 3n+1 in assembly language. It is installed after some testing but not before Paul has looked at it, because a bug is discovered in the spec of the check_for_logins handler (poke_search). The speed is holding pretty steady at an interval of 50000/minute.

15 Dec. 1986 20,295,987,881 A assembly language routine is written for adding 2 to an unsigned integer, which results in a speed up. The search speed is now slightly faster than an interval of 80000/minute in the range around 20 billion. The previous week saw the first stable storage problem in a hailstone search guardian.

17 Dec. 1986 20,486,895,159 An improved version of make_odd in assembly language causes some trauma when a bug is discovered. The bug is fixed, but a day's work had to be abandoned because the guardians were running with the faulty code. Preliminary measurements indicate a speed of about an interval of 98000/minute in the range around 20 billion.

9 Jan. 1987 37,145,592,705 Received a message from Mike Vermeulen that his search is up to 115 billion and going strong.

12 Feb. 1987 50,830,226,819 Search passes 50 billion. Reorganized the search guardian's code to make optimization easier, added in a priori cutoff for n mod 6 = 5. Preliminary measurements indicate rate is an interval of 94000/minute. Mike's search has passed 300 billion.

13 Feb. 1987 Added a priori cutoffs based on Vermeulen polynomials. Preliminary measurements show a rate of an interval of 127000/minute.

16 Mar. 1987 75,716,745,243 Added an eighth micro-vax (muon) to the machines doing the search. Search passes 75 billion. Mike's search is past 500 billion.

7 Apr. 1987 100,085,608,537 Stopped the search after reaching 100 billion.

B.2 The Second Search: Peaks in Total Stopping Time

The second search looked for peaks in total stopping time (σ_∞). It was started in April 1987, and continued for less than a month, until May 1987, when it reached 12.3 billion (12.3×10^9). At this point it seemed that all the peaks in σ_∞ would also be peaks in *steps*, although that is only a conjecture. Thus the search was not very interesting after the easy peaks were found.

29 Apr. 1987 2,432,000,001 Searching for steps in total stopping time (sigma infinity). Conjecture is verified up to 2.4 billion.

16 May 1987 12.3 billion Conjecture that all steps in total stopping time are also peaks in steps verified up to 12.3 billion. Shut down of this search.

B.3 The Third Search: Peaks in Stopping Time

The third search ran for the longest time, and looked for peaks in stopping time (σ). It was amazing and frustrating, because after discovering several peaks including the one at 12,235,060,455 during the first month of the search, no new peaks were discovered for over a year and a half!

16 May 1987 787,141,743 Looking for peaks in stopping time (sigma) now. Found peaks different from that in steps (or total stopping time).

19 May 1987 2.5 billion

26 May 1987 6,271,600,001 Improvements made to the algorithm and coding speed up the search by a factor of about 2.5.

27 May 1987 7,360,383,283 Speed of the sigma search measured at an interval of 264000/minute (approximately).

17 June 1987 30,552,706,183 Search reaches 30 billion. No new peaks since 12,235,060,455.

9 Nov. 1987 204,867,151,963 Using 6 machines. No new peaks since 12,235,060,455.

8 Jan. 1988 301,507,887,715 No changes, and still no new peaks since 12,235,060,455.

10 Feb. 1988 352,723,633,185 No changes. Remeasured speed at about an interval of 266000/minute, which as close to the previous measurement as quantization allows.

29 Feb. 1988 379,386,592,257 Still no new peaks.

14 Mar. 1988 399,958,386,665 No changes, still no new peaks.

21 Apr. 1988 450,613,332,285 Now running under Unix 4.3 on neutron. Still no new peaks.

4 June 1988 501,514,648,555 Now running on graviton (a VAX station) as well as kaon (a VAX 3200). Still no new peaks.

8 June 1988 512,778,240,151 Now running on meson (a VAX 3200). This makes 9 machines total, of which 6 are VAX stations. Still no new peaks.

21 June 1988 554,041,256,259 Still no new peaks. Also running on kaon.

10 July 1988 615,182,955,057 Still no new peaks.

22 July 1988 650,340,880,339 Still no new peaks.

9 Aug. 1988 700,521,112,239 Still no new peaks.

26 Aug. 1988 749,652,024,883 No longer running on kaon. Not enough disk swap space for Argus stable storage.

29 Aug. 1988 754,962,262,075 Still no new peaks.

30 Aug. 1988 755,180,044,185 No longer running on meson. Back to 7 microvaxes.

23 Sept. 1988 800,794,838,815 Still no new peaks.

2 Nov. 1988 855,258,557,331 Running again on kaon and meson. No longer running on proton (guardian won't start for some reason). Still no new peaks.

17 Nov. 1988 900,243,884,863 Found new peak in stopping time: 898,696,369,947 takes 550 steps!

1 Dec. 1988 944,761,925,329 Still no new peaks.

3 Jan. 1989 1,043,596,025,277 Reached a trillion. No new peaks. Terminating the search, since no longer at MIT.

References

[Day87] Mark S. Day. Replication and Reconfiguration in a Distributed Mail Repository. Technical Report MIT/LCS/TR-376, Massachusetts Institute of Technology, Laboratory for Computer Science, April 1987.

[Hay84] B. Hayes. Computer Recreations: On the Ups and Downs of Hailstone Numbers. *Scientific American*, 250(1):10–16, January 1984.

[HS86] W. Daniel Hillis and Guy L. Steele Jr. Data Parallel Algorithms. *Communications of the ACM*, 29(12):1170–1183, December 1986.

[Lag85] J. C. Lagarias. The 3x+1 Problem and its Generalizations. *The American Mathematical Monthly*, 92(1):3–23, January 1985.

[Lam81] B. W. Lampson. Atomic Transactions. In *Distributed Systems–Architecture and Implementation*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer-Verlag, New York, N.Y., 1981. This is a revised version of Lampson and Sturgis's unpublished *Crash Recovery in a Distributed Data Storage System.*

[LDH+87] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl. Argus Reference Manual. Technical Report 400, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1987. An earlier version appeared as Programming Methodology Group Memo 54 in March 1987.

[Lea86] Gary T. Leavens. Using a Guardian to Make Progress in a Long Computation: The Search for Hailstone Peaks. DSG Note 142, Massachusetts Institute of Technology, Laboratory for Computer Science, May 1986.

[Lea87] Gary T. Leavens. The Hailstone System. DSG Note 148, Massachusetts Institute of Technology, Laboratory for Computer Science, March 1987.

[LS83] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[Ver86a] Mike Vermeulen, December 1986. Private communication. In this mail message Mike states and proves the theorem that if n mod 6 = 5, then n cannot be a peak in steps or values.

[Ver86b] Mike Vermeulen, January 1986. Private communication. In this mail message Mike explains his composite polynomials, including a C program that generates them.

Tech Report: TR 89-22
Submission Date: November, 1989