Cycle Detection Using a Stack

Gabriel Nivasch
Weizmann Institute of Science, Rehovot 76100, Israel
gabriel.nivasch@weizmann.ac.il

January 27, 2004

Abstract

We present an algorithm for detecting periodicity in sequences produced by repeated application of a given function. Our algorithm uses logarithmic memory with high probability, runs in linear time, and is guaranteed to stop within the second loop through the cycle. We also present a partitioning technique that offers a time/memory tradeoff. Our algorithm is especially well suited for sequences where the cycle length is typically small compared to the length of the acyclic prefix.

Keywords: Cycle detection; Algorithms; Analysis of algorithms; Stack; Time/memory tradeoff; Random function.

1 Introduction

Given a function $f : D \to D$ and an initial element $x_0 \in D$, define the sequence $\{x_i\}$ by $x_i = f(x_{i-1})$ for $i \geq 1$.

If D is finite, this sequence must eventually become periodic. Then there exist unique μ and λ such that $x_0, \ldots, x_{\mu+\lambda-1}$ are all distinct, but $x_i = x_{i+\lambda}$ for all $i \geq \mu$. The elements $x_0, \ldots, x_{\mu-1}$ form the *prefix* of the sequence, and the elements $x_\mu, \ldots, x_{\mu+\lambda-1}$ constitute its *cycle*.

The *cycle detection problem* asks for finding a pair of elements $x_i = x_j$ for $i \neq j$, and possibly also finding the cycle length λ. There are several existing algorithms for this purpose; see [1, 3], [5, exercise 3.1–6], [9, 10].

Cycle detection arises in a number of situations:

- In studying the behavior of random number generators [5].
- In searching for function collisions. A *collision* is a pair $x \neq y$ such that $f(x) = f(y)$. One way to find a collision is to repeatedly apply f starting from some initial value x_0, and find the cycle length λ of the resulting sequence. Then, using the knowledge of λ, we can reconstruct the collision pair $x_{\mu-1}, x_{\mu+\lambda-1}$. Finding collisions has several cryptanalytic applications [6, 9].
- In order to detect when, say, a cellular automaton configuration has become periodic.
- In Pollard's *rho* methods for factorization and discrete logarithms [7, 8]. (The ideas we are about to present in this paper, however, do not seem applicable to the *rho* factorization method.)

In this paper we present a new cycle detection algorithm. The algorithm requires that a total ordering be defined on D. For simplicity, we assume that D is a finite set of integers.

Throughout this paper, we will refer to the x_i's as *values*, and to the corresponding indices i as *times*.

The rest of this paper is organized as follows: In Sections 2 and 3 we present our algorithm. In Section 4 we analyze its performance under a random function f. In Section 5, we compare our algorithm to some other cycle detection algorithms. We make some final observations in Section 6.

2 The basic stack algorithm

Our basic algorithm is as follows: Keep a stack of pairs (x_i, i), where, at all times, both the i's and the x_i's in the stack form strictly increasing sequences. The stack is initially empty. At each step j, pop from the stack all entries (x_i, i) where $x_i > x_j$. If an $x_i = x_j$ is found in the stack, we are done; then the cycle length is $\lambda = j - i$. Otherwise, push (x_j, j) on top of the stack and continue.

Proposition 1 *The stack algorithm always halts on the smallest value of the sequence's cycle, at some time in* $[\mu + \lambda, \mu + 2\lambda)$.

Proof Consider the cycle's minimal value $x_{\min}$. Once it is added to the stack on the first loop through the cycle, it is never removed. Therefore, the algorithm will halt when it encounters $x_{\min}$ on the second loop through the cycle. On the other hand, any other cycle value is greater than $x_{\min}$, so it will be removed by $x_{\min}$ before it has a chance to appear again. ∎

This algorithm runs in linear time, since the running time of each step is proportional to the number of elements removed from the stack at that step, and each element is removed at most once.

For further analysis of the algorithm, we assume that the sequence values have independently random magnitudes, subject to the periodicity constraint defined by μ and λ. This implies, in particular, that the relative order of the values $x_0, \ldots, x_{\mu+\lambda-1}$ corresponds to a random permutation of $\mu + \lambda$ elements. Note that for our purposes, the relative order of the values is all that matters.

(If the sequence is known in advance to have regularities, we can apply a hash function whenever we compare values in the stack.)

Since the cycle minimum $x_{\min}$ appears in a random position in the cycle, the algorithm's average running time for fixed μ and λ is $\mu + \frac{3}{2}\lambda$.

Now we analyze the behavior of the stack up to a given time $n < \mu + \lambda$. We can simplify our conceptual model by letting $\mu = \infty$, and feeding the stack random real numbers in $[0, 1]$. We first show that the maximum size of the stack up to time n is $\Theta(\log n)$ with high probability.

Theorem 2 *Given a positive integer n, let S_n be the stack size at time n, and M_n the maximum stack size up to time n. Then:*

1. *S_n has expectation* $H_{n+1} = \ln n + O(1)$.

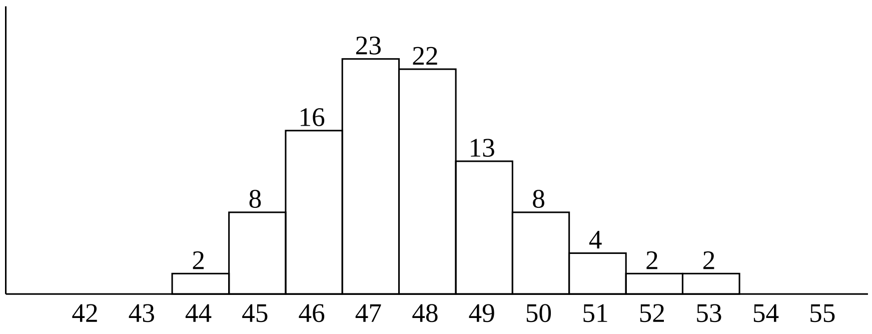

Figure 1: Maximum stack size at time $n = 5 \times 10^8$

2. S_n is almost surely $> \delta \ln n$ for any constant $\delta < 1$.

3. M_n is almost surely $< \delta \ln n$ for any constant $\delta > e$.

Proof (See [4, section 1.2.10].) We examine the sequence backwards. For each i, $1 \le i \le n+1$, let $X_i = 1$ if x_{n+1-i} is present in the stack at time n, and $X_i = 0$ otherwise.

Then, X_1 is always equal to 1, $X_2 = 1$ with probability 1/2, $X_3 = 1$ with probability 1/3 independently of X_1 and X_2, etc. In general, $X_i = 1$ with probability $1/i$ independently of all X_j, $j < i$.

This independence structure among the X_i's implies that they are fully independent.

Now, $S_n = \sum X_i$. Therefore $E \equiv E(S_n) = \sum_{i=1}^{n+1} 1/i = H_{n+1}$, proving our first claim.

For our other two claims, we apply Chernoff's bounds [4, exercise 1.2.10–22], given that the X_i's are fully independent:

$$\Pr(S_n \le rE) \le (e^{r-1}/r^r)^E, \qquad \text{for any } 0 < r \le 1; \tag{1}$$

$$\Pr(S_n \ge rE) \le (e^{r-1}/r^r)^E, \qquad \text{for any } r \ge 1. \tag{2}$$

To prove the third claim, we note that the right-hand side of (2) is $o(1/n)$ for $r > e$. And clearly, this bound on the stack size is also true for times $< n$. Therefore, by the union bound, the probability that the stack reaches size rE at any time t, $0 \le t \le n$, tends to zero as $n \to \infty$.

The second claim follows from (1) and it is even easier. ▮

Experiments seem to indicate that M_n has an upper bound somewhat lower than $e \ln n$. We fed $n = 5 \times 10^8$ pseudorandom values into an initially empty stack, and recorded the maximum stack size. We repeated this experiment 100 times. The results obtained were highly concentrated around an average of about $2.38 \ln n$. See Figure 1.

We end this section with a result concerning the distribution of the entries in the stack at time n.

Theorem 3 *Suppose an initially empty stack is fed random real numbers in* $[0, 1]$. *Then, at time n, the expected magnitude of the i-th topmost stack entry is 2^{-i}, for $i \ge 1$. The expected time of the i-th bottommost stack entry is $n(1 - 2^{-i})$.*

Proof By induction. Denote the i-th topmost stack entry by s_i. Then s_1 is a uniformly random number in $[0, 1]$, so $E(s_1) = 1/2$. And for any $i > 1$, the entry s_i is a random number subject to the constraint $s_i < s_{i-1}$. Therefore, the distribution of s_i can be generated by $s_i = Xs_{i-1}$, where X is a uniformly random number in $[0, 1)$, independent of s_{i-1}. Therefore, $E(s_i) = E(s_{i-1})/2$.

A similar argument applies to the times of the bottommost stack entries. ∎

3 A partitioning technique

We now present a partitioning technique for reducing the stack algorithm's halting time, while increasing its memory use only by a constant factor, and not affecting the running time per step.

For some integer k, we divide the domain D into k disjoint classes. We could, for example, consider the remainder modulo k of each value x_i. Or, more conveniently, we could let k be a power of 2, and take either the most-significant or the least-significant bits of x_i.

We keep a separate stack for each class. At each step i, we first determine which class x_i belongs to, and then we insert (x_i, i) in the corresponding stack.

This *multi-stack* algorithm works correctly because, whenever a given value appears, it is always sent to the same stack. Further, since at each step, the algorithm operates on a single stack, the run time per step remains practically unchanged.

To derive the algorithm's running time, note that each class j, $0 \le j < k$, contains its own cycle minimum $x_{\min,j}$. The algorithm halts when it encounters the *first* of all these minima for a second time. And under our randomness assumptions, each $x_{\min,j}$ is distributed uniformly and independently at random in the cycle. It follows that, for fixed μ and λ, the average running time decreases to

$$\mu + \lambda(1 + 1/(k + 1)).$$

Any algorithm must obviously call f at least $\mu + \lambda$ times. So if we take, for example, $k = 100$, then our algorithm's running time will be just 1% of the cycle length above the absolute minimum achievable.

It can also be shown that the variance of the running time is close to λ^2/k^2 for large k.

If $k \ll \mu + \lambda$, the memory used increases from $O(\log(\mu + \lambda))$ to

$$O(k \log(\mu + \lambda)),$$

so the memory is multiplied by a factor of k.

4 Performance under a random function

Now we analyze our algorithm's performance under a random function. Let D be a set of m integers, and choose a function $f : D \to D$ uniformly at random from among the m^m possible such functions. Also, choose the initial value x_0 randomly from D.

It has been shown that in this scenario, for large m, the joint probability density function of μ and λ is approximately ([3], [5, exercise 3.1–11], see also [6])

$$w(\mu, \lambda) = \frac{1}{m} e^{-(\mu+\lambda)^2/(2m)}, \qquad \text{for } \mu, \lambda \ge 0.$$

Each of μ and λ has expectation $\sqrt{\pi m/8}$. (The birthday paradox implies that $E(\mu+\lambda) = O(\sqrt{m})$.) For a fixed $\mu+\lambda$, the value of μ is distributed uniformly in $[0, \mu+\lambda)$. This implies the following corollary to Theorem 3:

Corollary 4 *Under a random function f, when the single-stack algorithm halts, the stack contains exactly i entries from the prefix with probability $2^{-(i+1)}$, for $i \geq 0$.*

Proof When the algorithm halts, all the stack entries are from the prefix except for the topmost entry $x_{\min}$.

Consider the stack at time $n = \mu+\lambda-1$. Let $(s_1, t_1), (s_2, t_2), \ldots$ be the entries in the stack at that time. Then s_i is in the prefix if and only if $t_i < \mu$. But, as we said before, once n is fixed, μ is distributed uniformly in $[0, n)$. It follows that $\Pr(t_i \geq \mu) = E(t_i)/n$. But we proved in Theorem 3 that $E(t_i) = n(1 - 2^{-i})$. The claimed result follows. ∎

Now we derive the expectation and the variance of the algorithm's running time under a random f. We start with the single-stack version.

Let T_1 be the random variable for the halting time of the single-stack algorithm. For fixed μ and λ, the event $T_1 = t$ occurs with probability $f_1(\mu, \lambda, t) = 1/\lambda$, for $\mu+\lambda \leq t < \mu+2\lambda$.

Therefore, the overall expected running time is

$$\begin{aligned} E(T_1) &= \int_{\mu=0}^{\infty}\int_{\lambda=0}^{\infty}\int_{t=\mu+\lambda}^{\mu+2\lambda} f_1(\mu,\lambda,t)w(\mu,\lambda)\,dt\,d\lambda\,d\mu \\ &= 5\sqrt{\frac{\pi m}{32}} \;\approx\; 1.5666\sqrt{m}. \end{aligned} \tag{3}$$

Similarly,

$$\mathrm{Var}(T_1) = \Big(\frac{29}{9} - \frac{25\pi}{32}\Big)m \approx .7679\,m. \tag{4}$$

In the multi-stack case, for fixed μ and λ, the running time T_k is equal to t with probability

$$f_k(\mu,\lambda,t) = \frac{k}{\lambda}\Big(2 + \frac{\mu - t}{\lambda}\Big)^{k-1}, \qquad \text{for } \mu+\lambda \leq t < \mu+2\lambda.$$

Therefore,

$$E(T_k) = \Big(1 + \frac{1}{2(k+1)}\Big)\sqrt{\pi m/2}, \tag{5}$$

and

$$\mathrm{Var}(T_k) \approx (1+1/k)(2-\pi/2)m, \qquad \text{for large } k. \tag{6}$$

Therefore, as $k \to \infty$, the expectation and the variance converge to their minimum possible values.

5 Comparison to other algorithms

As we have seen, our basic stack algorithm runs in linear time, uses logarithmic space, and is guaranteed to stop within the second repetition of the sequence's cycle, regardless of its size. Our partitioning technique offers a time/memory tradeoff: for memory $O(k\log(\mu+\lambda))$, we halt at time λ/k above the absolute minimum $\mu+\lambda$.

How does our algorithm compare to other known algorithms? We now address this issue.

5.1 Floyd's and Brent's algorithms

We start with a pair of algorithms that need just two memory locations to store sequence values.

Floyd's algorithm [5, exercise 3.1–6] is based on the observation that, for given μ and λ, there is always a unique j, $\mu \leq j < \mu + \lambda$, such that $x_j = x_{2j}$. Thus, to find this j, we initialize $x \leftarrow x_0$, $y \leftarrow x_0$, $j \leftarrow 0$; then we repeatedly perform $x \leftarrow f(x)$, $y \leftarrow f(f(y))$, $j \leftarrow j + 1$, until we find that $x = y$. Once we halt, j will be a multiple of λ (though not necessarily λ itself).

Brent's algorithm [3] is an improvement on Floyd's. In Brent's algorithm we keep a single running copy of the sequence. At each step i, we compare the current value x_i to a previously saved value y; if i is a power of 2, we also update y, by letting $y \leftarrow x_i$, $j \leftarrow i$. Once we find that $x_i = y$, the difference $i - j$ will be the actual value of λ.

Brent shows that this simple approach never performs worse than Floyd's. Brent also analyzes his algorithm's running time T_B under a random function f; he finds that

$$E(T_B) \approx 1.9828\sqrt{m}, \quad \text{and} \quad \mathrm{Var}(T_B) \approx 1.4241\, m. \tag{7}$$

By (3) and (4), our single-stack algorithm is about 20% faster than Brent's on average, and it also has a smaller variance. We pay a price, however, in the amount of memory we use.

5.2 Sedgewick, et al.'s algorithm

Sedgewick, Szymanski, and Yao [10] analyze the problem of optimizing *worst-case* performance, using a bounded amount of memory. They present an algorithm that uses a table of size M, where M is a free parameter.

Their algorithm is as follows. Let T be a table of size M. Initially we set $d \leftarrow 1$. At each step i, if $i \bmod gd < d$, we search for x_i in T (here g is a free parameter, just like M). And if i is a multiple of d, we store (x_i, i) in T. Whenever the table becomes overfull, we double d and erase from the table all entries (x_j, j) where j is no longer a multiple of d.

The table T should be implemented so as to asymptotically reduce the worst-case search time, using, for example, a balanced tree approach or some hashing method.

Let t_s be the time needed to perform one search in T, and let t_f be the time needed to evaluate f once. Then, the authors show, g can be chosen as a function of t_s, t_f, and M, such that the algorithm's worst-case running time is $t_\mathrm{f}(\mu+\lambda)(1+\Theta(\sqrt{t_\mathrm{s}/Mt_\mathrm{f}}))$. (They also show that this worst-case performance is asymptotically optimal.)

Thus, our multi-stack algorithm's *average-case* performance, as a function of the memory used, is asymptotically better than the *worst-case* performance of Sedgewick et al.'s algorithm. (Our algorithm's worst-case performance is, of course, much worse.)

5.3 Algorithms with a better time/memory tradeoff

Some approaches achieve a better time/memory tradeoff than ours.

The *distinguished point* method [6, 9] consists of keeping all values that satisfy a certain easily-testable *distinguished* property (e.g., ending in a certain number of zeros in binary). At each step i, if the current value x_i is distinguished, we store it and compare it to all previously stored values. We use hashing to perform all these comparisons simultaneously. If we choose the distinguished property in advance so that about k

points are memorized, then the expected running time is about $(\mu + \lambda)(1 + 1/k)$. In contrast, to achieve the same running time, our multi-stack algorithm needs $\log(\mu + \lambda)$ times as much memory.

Our algorithm, however, offers certain advantages. First, the distinguished point algorithm is not guaranteed to succeed; it will not halt if the cycle contains no distinguished points, which is likely if λ happens to be small. Furthermore, our algorithm is simpler because it does not require the use of a hash table.

Another approach, based on a suggestion by Woodruff [11], is to keep a *random sample* of sequence values. We proceed as follows. As in Section 3, we divide the domain D into $k \geq 2$ disjoint classes of equal size. We keep an array S of size k. At each step i, we first find to which class j, $0 \leq j < k$, the current value x_i belongs to; then we check whether $x_i = S[j]$, and if so, we halt. Otherwise, we decide at random whether to replace $S[j]$ by x_i or not: we let $S[j] \leftarrow x_i$ with probability k/i.

It is easy to check that, at every time $n \gg k$, each entry $S[j]$ contains a uniformly random representative of the values x_i, $i \leq n$, that belong to class j. One can further show that the algorithm's expected running time, for fixed μ and λ, is $(\mu + \lambda)(1 + 1/(k-1))$. Thus, we achieve a time/memory tradeoff similar to the distinguished point algorithm.

This sampling method, however, has the disadvantage of requiring an extensive amount of randomness. Since $\mu + \lambda$ might be very large, we need to generate a large amount of random numbers, so we need a random number generator of high quality. Then the random number generation could take up a significant portion of the running time.

5.4 The case $\mu \gg \lambda$

Not in all applications does f behave like a random function. There are situations where it is common for μ to be much larger than λ—for example, when running configurations in certain cellular automata until they oscillate, such as in the Game of Life [2]. Besides the stack algorithm, all the algorithms mentioned so far have a running time proportional to μ as well as to λ. In contrast, the stack algorithm's running time above μ depends only on λ; therefore, it is ideally suited for cases where $\mu \gg \lambda$.

Gosper's algorithm [1] shares this property; like ours, it is guaranteed to stop before time $\mu + 2\lambda$, and it also uses logarithmic memory. Gosper's algorithm works as follows. For $n \geq 1$, denote by $z(n)$ the number of trailing zeros of n written in binary. Keep an array T of saved values. Initially, let $T[0] = x_0$. At each step i, compare x_i to all the values stored in T; if a match is found, halt. Otherwise, let $T[z(i+1)] = x_i$.

Again, we should use hashing to check the presence of x_i in table T in constant time. The size of T at time i is $\lceil \log_2(i+1) \rceil$. Thus, Gosper's algorithm is comparable to the single-stack algorithm in both memory use and halting time. As before, however, our algorithm is simpler because it does not require the use of a hash table.

6 Conclusion

Of all the cycle detection algorithms available, which is the best? It depends on the circumstances. If f behaves like a random function, our multi-stack algorithm is not a bad choice, although the distinguished point algorithm requires less memory. To prevent the latter from occasionally failing to halt, we could simultaneously run a stack as a backup.

If f is known *not* to behave like a random function (i.e., it often produces short cycles), our stack algorithm is most appropriate.

On the other hand, when our objective is to minimize *worst-case* performance, using a predetermined, fixed amount of memory, we should use the algorithm of Sedgewick et al. [10].

We end with some observations:

1. In principle, the partitioning technique can be applied to any cycle-finding algorithm that keeps only one running copy of the sequence. In fact, we have already done so in the sampling algorithm above. (The distinguished point method also uses partitioning: it simply ignores all classes except for one, to which it applies the "brute-force" approach of keeping all the values in the sequence.)

   For illustration, let us apply the partitioning technique to Brent's algorithm [3]. Instead of a single saved value y, we now have a saved value y_j for each class j, $0 \leq j < k$. Each class also has its own time counter t_j, besides a "global" time counter t. In order to space out the save times as much as possible, we could update y_j whenever $t_j = \lfloor 2^{i+j/k} \rfloor$, for $i = 0, 1, 2, \ldots$. The analysis, however, lies beyond the scope of this article.

2. As mentioned in the Introduction, our stack algorithm does not seem applicable to Pollard's *rho* factorization method [3, 7]. Given two numbers modulo n, we do not know how to determine which of the two is larger modulo an unknown factor p of n. Similarly, the partitioning technique does not seem applicable either.

Acknowledgements

I would especially like to thank Adi Shamir for all his assistance throughout the development of the material in this paper. Thanks also to Ron Rivest for pointing out reference [4], and to David Woodruff for some help in proving Theorem 2.

References

[1] M. Beeler, R. W. Gosper, and R. Schroeppel, *HAKMEM*, M.I.T. Artificial Intelligence Lab Memo 239 (1972), Item 132.

[2] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways for your Mathematical Plays, II: Games in Particular*, Academic Press, 1982, pp. 817–850.

[3] R. P. Brent, An improved Monte Carlo factorization algorithm, *BIT*, vol. 20 (1980), pp. 176–184.

[4] D. E. Knuth, *The Art of Computer Programming* vol. 1, 3rd edition, Addison-Wesley, Reading, MA, 1997.

[5] D. E. Knuth, *The Art of Computer Programming* vol. 2, 3rd edition, Addison-Wesley, Reading, MA, 1997.

[6] P. C. van Oorschot and M. J. Wiener, Parallel collision search with cryptanalytic applications, *J. Cryptology*, vol. 12 (1999), pp. 1–28.

[7] J. M. Pollard, A Monte Carlo method for factorization, *BIT*, vol. 15 (1975), pp. 331–334.

[8] J. M. Pollard, Monte Carlo methods for index computation (mod p), *Math. Comp.*, vol. 32, no. 143 (1978), pp. 918–924.

[9] J.-J. Quisquater and J.-P. Delescaille, How easy is collision search? Application to DES, *Advances in Cryptology—Eurocrypt '89 Proceedings*, Lecture Notes in Computer Science, vol. 434, Springer-Verlag, Berlin, pp. 429–434.

[10] R. Sedgewick, T. G. Szymanski, and A. C. Yao, The complexity of finding cycles in periodic functions, *SIAM J. Comput.*, vol. 11, no. 2 (1982), pp. 376–390.

[11] David P. Woodruff, personal communication.