SIAM J. COMPUT.
Vol. 11, No. 2, May 1982

0097-5397/82/1102-0015 $1.00/0

THE COMPLEXITY OF FINDING CYCLES IN PERIODIC FUNCTIONS*

ROBERT SEDGEWICK†, THOMAS G. SZYMANSKI‡ AND ANDREW C. YAO§

Abstract. Given a function f over a finite domain D and an arbitrary starting point x, the sequence $f^0(x), f^1(x), f^2(x), \cdots$ is ultimately periodic. Such sequences are typically the output of random number generators. The *cycle problem* is to determine the first repeated element $f^n(x)$ in the sequence. Previous algorithms for this problem have required $3n+O(1)$ operations. In this paper we show that $n(1+\Theta(1/\sqrt{M}))$ steps are both necessary and sufficient, if M memory cells are available to store values of the function. We explicitly consider the performance of the algorithm as a function of the amount of memory available and the relative cost of evaluating f and comparing sequence elements for equality.

Key words. computational complexity, time-space tradeoffs, cycle detection

1. Introduction. Suppose that we are given an arbitrary function f which maps some finite domain D into D. If we take an arbitrary element x from D and generate the infinite sequence $f^0(x), f^1(x), f^2(x), \cdots$, then we are guaranteed by the "pigeonhole" principle and the finiteness of D that the sequence becomes cyclic. That is, for some l and c we have $l+c$ distinct values $f^0(x), f^1(x), \cdots, f^{l+c-1}(x)$ but $f^{l+c}(x) = f^l(x)$. This implies, in turn, that $f^{i+c}(x) = f^i(x)$ for all $i \geq l$. The problem of finding this unique pair (l, c) will be termed the *cycle problem* for f and x. The integer c is the *cycle length* of the sequence, and l is termed the *leader length*. Similarly, the elements $f^l(x), f^{l+1}(x), \cdots, f^{l+c-1}(x)$ are said to form the *cycle* of f on x and $f^0, f^1(x), \cdots, f^{l-1}(x)$ are said to form the *leader* of f on x. For notational convenience, the number $l+c$ of distinct values in the sequence will be denoted by n.

The cycle problem arises when analyzing pseudo-random number generators that produce successive "random" values by applying some function to the previous value in the sequence [1, §3.1]. Solving the cycle problem gives the number of distinct random numbers which can be produced from a given seed. Algorithms for the cycle problem are used in checking the characteristics of random number generators whose internal properties are unknown. Other applications include checking for loops in self-referent lists (see [2]), and studying the performance of certain numerical calculations (see [5]). Beyond these practical motivations, the problem is of some intrinsic combinatorial interest.

A graphic restatement of the problem is provided by imagining a directed graph whose nodes are the elements of D and which contains an arc from y to $f(y)$ for every $y \in D$. For example, Fig. 1a shows the graph corresponding to $f(x) = (2x+1) \bmod 10$, with $D = 0,1, \cdots, 9$. The cycle structure for a function consists of a number of

*Received by the editors November 13, 1980, and in revised form July 24, 1981. This paper was typeset at Bell Laboratories, Murray Hill, New Jersey, using the ***troff*** program running under the UNIX™ operating system. Final copy was produced on December 29, 1981.

†Department of Computer Science, Brown University, Providence, Rhode Island 02912. This work was done in part while this author was visiting the Institute for Defense Analyses, in part under support from the National Science Foundation, grant MCS-75-23738, and in part while this author was visiting the Xerox Palo Alto Research Center.

‡Bell Laboratories, Murray Hill, New Jersey 07974. This work was done in part while this author was visiting the Institute for Defense Analyses.

§Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720. This work was done in part under support from the National Science Foundation, grant MCS-77-05313, and in part while this author was visiting the Xerox Palo Alto Research Center.

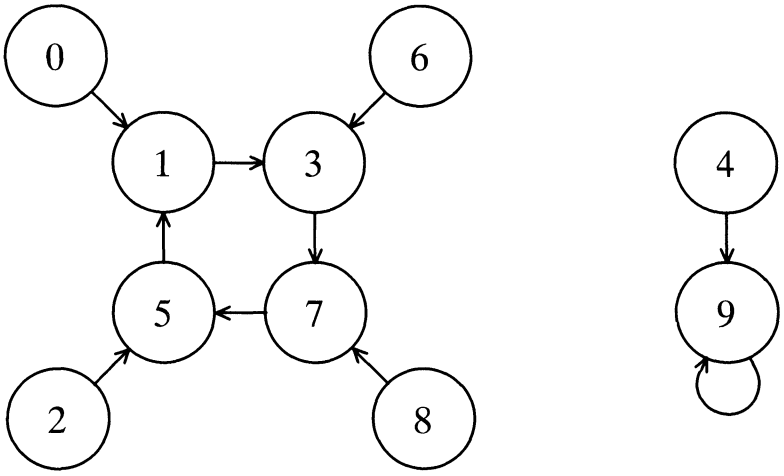

FIG. 1a. *A typical cycle structure.*

disjoint cycles, with disjoint trees feeding points on the cycles. To solve the cycle problem, we need consider only the subgraph consisting of a single cycle and leader, which can be drawn as shown in Fig. 1b.

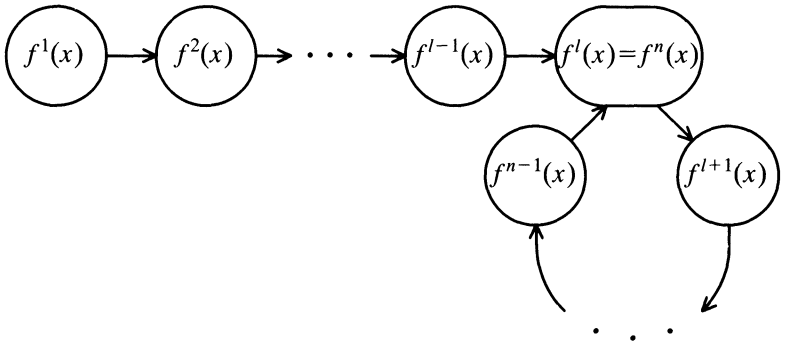

FIG. 1b. *A typical cycle and leader.*

One method for cycle detection, commonly referred to as "the tortoise and hare" algorithm, has been given by Floyd [1, Exercise 3.1-7]. The idea is to have two variables taking on the values in the sequence, one advancing twice as fast as the other. A program implementing this idea is given in Fig. 2.

```
y ← z ← x;
repeat
    y ← f(y);
    z ← f(f(z));
until y=z;
```

FIG. 2. *Floyd's algorithm.*

This algorithm stops with $y=f^i(x)=f^{2i}(x)=z$, where i is the smallest positive multiple of c which is greater than or equal to l. If $l=0$ then $3n$ function evaluations are performed, and if $l=c+1$ or if $c=1$ with $l\neq 0$ then a total of $3(n-1)$ function evaluations are performed. This number may be objectionable when the cost of evaluating f is

high relative to the cost of comparisons.

Another method, due to Gosper [2, page 64], was designed to circumvent the overhead of advancing two independently operating "copies" of the generating function as required in Floyd's method. His method is to save certain values of the sequence in a small table (whose size must be at least $\log_2 n$) and to search for each new value to see if it has previously been generated. The table update rule is to save the ith value generated in the jth cell of the table, where j is the number of trailing zeroes in the binary representation of i. This method can require as many as $l+2c$ function evaluations, or $2n$ if $l=0$, and $3n/2$ if $l=c$. Moreover, it requires at least an equal number of table searches, which would be important if the cost of comparisons were high relative to the cost of evaluating f.

These algorithms are suitable for detecting the existence of a cycle. The value of c can be found by proceeding around the cycle one additional time. Of course, this may be undesirable if c is very large. Moreover, neither algorithm has provision for directly finding l except by starting back at the initial value.

In this paper, we develop an algorithm that solves the cycle problem using $n(1+O(1/\sqrt{M}))$ function evaluations in the worst case, where M is the amount of memory available for storing generated function values. The number of memory operations (i.e., stores and searches) used is $O(n/\sqrt{M} + M\log n/M)$. The algorithm is developed in §2 in two parts: one stage which detects the cycle, and a companion stage which recovers the values of l and c. A worst case analysis of the algorithm is given in §3. In §4, we derive a lower bound which shows that no algorithm using the same fundamental operations can have a substantially better worst case performance. Our algorithm for the cycle problem thus demonstrates a tight, non-trivial tradeoff in which time is a continuous function of memory size. A generalization of the problem and some concluding remarks are offered in §5.

2. The algorithm. Any algorithm for the cycle problem must have a running time of at least nt_f where t_f is the (assumed constant) time to perform one evaluation of f. It should be clear that by using a large amount of memory we can produce an algorithm whose running time is $nt_f+O(n\log n)$ by employing, for example, a balanced tree scheme to save all elements generated in the sequence. Such an algorithm is unsatisfactory for at least two reasons. First, it is unrealistic to assume an unlimited supply of memory. Second, it does not take into consideration the relative complexity of evaluating f and comparing two domain elements for equality. Let us therefore construct a framework in which these considerations can be addressed. We shall be particularly interested in the tradeoff between memory size and execution time.

Let *TABLE* be an associative store capable of storing up to M pairs (y, i) of domain elements and integers. Both elements of the pair are *keys* in the sense that it is possible to search *TABLE* for an entry that contains a specified value for its first (or second) component. Let t_u be the time needed to insert or delete a pair from *TABLE*, i.e., to update *TABLE*, and let t_s be the time needed to search *TABLE* for a given key. Depending on the implementation of *TABLE*, t_u and t_s might be constants, logarithmic functions of M or even linear functions of M (see §5). All other operations of the algorithm are assumed to be free.

Within this model, we are ready to develop an algorithm for the cycle problem. The basic idea is to limit the number of operations performed on *TABLE* by only storing and searching for occasional values in the sequence $f^0(x), f^1(x), \cdots$. Thus, most of the time consumed by the algorithm is spent advancing the function f. In order to implement this idea, let us introduce two parameters, b and g. Fig. 3

contains an algorithm which only stores every bth function value in *TABLE* and which performs searches on blocks of b consecutive values spaced gb apart in the sequence.

```
y ← x;
i ← 0;
repeat
    if (i mod b) = 0 then insert(y, i);
    y ← f(y);
    i ← i+1;
    if (i mod gb) < b then search(y);
until found;
output i, j;
```

FIG. 3. *Preliminary version of the algorithm.*

Here the procedure *insert*(y, i) puts the pair (y, i) into *TABLE* without checking to see if there is another entry already there with the same first component. The procedure *search*(y) sets the variable *found* to false if no pair in *TABLE* has y for its first component, otherwise it sets *found* to true and j to the minimum value of j for which (y, j) is in *TABLE*. The modulus computations in this program are used for clarity; an actual implementation would use counters instead.

The program is guaranteed to halt because once the cycle is reached at least one function value out of every block of b consecutive values searched for must be in *TABLE*. Although it is possible to overshoot the point at which the cycle first returns to itself, it is clear that the algorithm will always detect the cycle before the $(n+gb)$th evaluation of f. Since the algorithm performs g updates and b searches for every gb evaluations of the function, the worst case running time of the algorithm is no greater than $(n+gb)(t_f+t_s/g+t_u/b)+bt_s$. The bt_s term is caused by the fact that the searches are not uniformly distributed within the sequence of function evaluations.

It is interesting to note that a dual algorithm can be developed by interchanging the roles of *search* and *insert* in Fig. 3, that is, every bth function value is searched for, and a block of b function values is stored every gb evaluations. Most of the results of this paper then carry through for the dual algorithm. Further development along these lines is left to the reader.

We could arrange to have the algorithm spend virtually all its time doing the (unavoidable) task of stepping f by choosing b and g suitably, were it not for the fact that *TABLE* will soon fill up. Accordingly, we introduce the following memory management mechanism. Whenever *TABLE* gets filled, we invoke a procedure *purge*(b) which removes all entries (z, j) from *TABLE* for which $j \not\equiv 0 \pmod{2b}$. We then double b, and continue. This has the same effect as restarting the program from the beginning (with the larger value of b) and running it to the current value of i. Notice that this effect is achieved at the cost of a few memory operations and, more importantly, no additional function evaluations. The algorithm thus adapts its behavior to the problem at hand. (A similar memory allocation strategy can be devised for the dual version of the algorithm mentioned above.)

The final version of the algorithm is shown in Fig. 4. The variable m is used to count the number of entries currently in *TABLE*. Notice that b is now a variable of the program, while g is still a parameter. The memory size M must be at least 2 and g can be any integer in the range $1 \le g < M$. If $g=1$ then every generated function value is looked for in *TABLE* and the algorithm will halt very soon after the nth function evaluation. Larger values of g result in fewer searches but delay the point at which a duplicate element is discovered. It will be explained later how to best choose

the value of g.

```
y ← x;
i ← 0;
m ← 0;
b ← 1;
repeat
    if (i mod b) = 0 and m = M then
        begin
            purge(b);
            b ← 2b;
            m ← ⌈m/2⌉;
        end;
    if (i mod b) = 0 then
        begin
            insert(y, i);
            m ← m+1;
        end;
    y ← f(y);
    i ← i+1;
    if (i mod gb) < b then search(y);
until found;
output i, j;
```

FIG. 4. *The cycle detecting algorithm.*

The following lemma provides the key invariant relations necessary for understanding the operation of the algorithm. The corollaries to the lemma provide useful facts needed in the analysis and correctness proof.

LEMMA 1. *The following relationships hold among the variables in the cycle detecting algorithm at the start of each* **if** *statement (and therefore at each call of search, insert, or purge):*

(a) $y = f^i(x)$,

(b) $(f^j(x), j)$ *is in TABLE if and only if* $j \equiv 0 \pmod{b}$ and $0 \leq j < i$,

(c) *the number of entries in TABLE is* $m = \lceil i/b \rceil$.

Proof. Clearly, the relations are all true when the **repeat** loop is first entered. Moreover, it is easy to see that (a) is preserved throughout the program because the variables y and i are only changed in one place in the loop. It remains to show that (b) and (c) are preserved by the loop body.

Consider the first **if** statement in the loop. If its predicate is false, no variables are changed and the relations are preserved. If its predicate is true, then by induction we have $m = \lceil i/b \rceil = M$ with $i \equiv 0 \pmod{b}$. Thus $i = Mb$ and *TABLE* contains $(f^j(x), j)$ for $j \in \{0, b, 2b, \cdots, (M-1)b\}$. After the call on *purge*, *TABLE* contains only those entries with $j \in \{0, 2b, 4b, \cdots, kb\}$ where k is $M-2$ if M is even, and $M-1$ if M is odd. In either case, the number of entries remaining in *TABLE* is $(k/2)+1$, which turns out to be $\lceil M/2 \rceil$. Thus (c) is preserved by the purge and subsequent assignment to m. It should be equally obvious that the purge and subsequent doubling of b preserve (b), so we have thus established that the first **if** statement preserves the relations in the lemma.

The preservation of (b) and (c) by the rest of the loop body is straightforward. □

COROLLARY 1. *For* $k \geq 0$, *the* $k+1$*st call on purge increases b from* 2^k *to* 2^{k+1} *and occurs when* $i = 2^k M$.

COROLLARY 2. *The value of b when search is called is i/M, rounded up to the next integral power of 2, that is,* $2^{\lceil \log_2 i/M \rceil}$.

Our main goal in the study of the performance of Algorithm 4 is to prove that it halts fairly soon after the nth evaluation of f. We would expect termination to occur during the execution of the first block of consecutive searches initiated after the time when $i=n$: this turns out to be true, but the proof is complicated substantially by the possibility that purges can occur at inopportune times, thus delaying the start of the search block. The situation is quite complicated because, though g and M are fixed ahead of time, the algorithm must work correctly for *all* values of l and c. It is not obvious that the algorithm is guaranteed to terminate within a reasonable amount of time: for some choices of g and M there might be values of l and c that the search block is delayed for some time by unfortunate purges. The following lemma shows that this cannot be the case, and gives precise bounds on the time at which the algorithm must terminate.

LEMMA 2. *Let b_n be n/M rounded up to the next power of 2 (i.e., b_n is the value of b when i is assigned the value of n). Then i_f, the value of i at the termination of the cycle detecting algorithm, obeys:*

$$n \le i_f < n+(g+1)b_n \qquad \text{if } \lceil M/g \rceil \text{ is even,}$$
$$n \le i_f < n+(2g+1)b_n \qquad \text{if } \lceil M/g \rceil \text{ is odd.}$$

Moreover, these bounds on i_f are as tight as possible.

Proof. Complicated interactions between the occurrence of purge and search operations, based on arithmetic relationships between M, g, l, and c, make this proof an intricate case analysis, which is relegated to the Appendix. To see the flavor of the proof, consider the simplest case, when no purge operations occur between the first evaluation of $f^n(x)$ and the search that terminates the algorithm. In this case, the algorithm performs searchs for $f^i(x)$, $i_s \le i < i_s + b_n$, where i_s is the (unique) multiple of gb_n for which $n \le i_s < n+gb_n$. During the search, since there are no purges (the precise conditions for this case are given in the Appendix), the value of of b remains fixed at b_n. Since $i_s \ge n$, the values searched for are equal, respectively, to $f^j(x)$, $i_s - c \le j < i_s + b_n - c$, exactly one of which, by Lemma 1(b), must be in *TABLE*. The algorithm therefore finds a match on one of these searches and terminates with *found* true and $i_f < i_s + b_n < n+gb_n+b_n = n+(g+1)b_n$. Notice that if $c<b_n$, the $f^j(x)$ that is found will have $j=i_s$. □

The algorithm of Fig. 4 halts as soon as it discovers a pair $i>j$ of integers for which $f^i(x)=f^j(x)$. This implies that $j \ge l$ and that $i \equiv j \pmod c$, but we need to do some additional processing to find the exact values of l and c. Fig. 5 shows a companion algorithm which recovers the solution (l, c) once the cycle detecting algorithm has terminated.

```
if f^j(x) = f^{j+c}(x) with 1 ≤ c ≤ (g+1)b then
    c ← smallest such c;
else
    c ← i − j;
i' ← max(c, gb⌊i/gb⌋ − gb);
j' ← i' − c;
l ← smallest l > j' such that f^l(x) = f^{l+c}(x);
output l, c;
```

FIG. 5. *The recovery algorithm.*

The second to last statement in this program may require evaluating f starting at a point that is not in *TABLE*. This can be done with little extra overhead. For example, $f^{j'}(x)$ can be found by doing a *search* for a *TABLE* entry whose second component is $b\lfloor j'/b \rfloor$ and then applying f exactly j' (mod b) times to $f^{b\lfloor j'/b \rfloor}(x)$.

LEMMA 3. *The recovery algorithm correctly computes c using at most $2(g+1)b_n$ function evaluations.*

Proof. The bound on the number of function evaluations is immediate from the observation that the final value of b is either b_n or $2b_n$.

The correctness of the computation of c has two cases depending on the predicate in the initial **if** statement in the algorithm. If the true branch is taken, then c is correctly computed by definition of cycle size. If the false branch is taken, then we must have $c>(g+1)b$. However, we know from the proof of Lemma 4 that $i<n+(g+1)b$ and hence $i<n+c$. Since $j\geq n-c$, this means that $i-j<2c$. Because $i-j$ must be a multiple of c, this implies that $i-j=c$, which is precisely what the algorithm has computed in this case. □

LEMMA 4. *The recovery algorithm correctly computes l using at most $4(g+1)b_n$ function evaluations and two memory searches.*

Proof. The expression $gb\lfloor i/gb \rfloor$ gives the value of i at the start of the final block of searches that the algorithm performed. Subtracting an additional term of gb from this gives the start of some previous block of searches that was completed unsuccessfully. Thus $n-gb\leq gb\lfloor i/gb \rfloor - gb<n$ and we have $\max(c, l+c-gb)\leq i'<l+c=n$. This implies that $\max(0, l-gb)\leq j'<l$. It should be clear from the definition of leader length that the algorithm correctly computes l.

The time bound follows from a more detailed consideration of the implementation of the statement that assigns l. As mentioned above, $f^{j'}(x)$ and $f^{j'+c}(x)$ can each be found by performing a memory search and $b\leq 2b_n$ function evaluations. Assigning these function values to variables and applying f to both of them until they are equal involves (from the range given above on j') at most $gb\leq 2gb_n$ function evaluations apiece. □

It is possible to design faster recovery procedures for many situations. For example, c could be found by applying "divide and conquer" to the prime decomposition of $i-j$, and l could be found by a binary search procedure. However, the recovery time is heavily dominated by the cycle detection time, so such sophisticated implementation tricks might not be worth the effort.

3. Worst case analysis. The algorithms of the previous section can provide an efficient solution to the cycle problem if the parameter g is chosen intelligently. In this section we shall analyze the running time of the algorithms to find the best choice of g. We shall concentrate on minimizing the worst case running time.

THEOREM 1. *In the worst case, the running time of the cycle detecting algorithm is at most*

$$n\left(1+\frac{4g+2}{M}\right)\left(t_f+\frac{t_s}{g}\right)+t_u M\log_2\frac{8n}{M}.$$

The additional time required to recover the values of l and c is at most

$$n\frac{12(g+1)}{M}t_f+2t_s.$$

Proof. Corollary 2 tells us that $b_n\leq 2n/M$. Lemma 2 then implies that $i_f<n(1+(4g+2)/M)$. The detection algorithm performs i_f evaluations of f for a

contribution of $i_f t_f$ to the running time. Throughout the execution of the algorithm, b searches are performed for every gb function evaluations. Thus the total number of search operations is i_f/g, contributing $i_f t_s/g$ to the running time. This takes care of the first term. (Note that the full result of Lemma 2 implies that coefficient of g in the $O(1/M)$ term could be reduced to 2, by taking $\lceil M/g \rceil$ to be even.)

For the second term, observe that each call on *purge* removes $M/2$ elements from *TABLE*. If we charge each element with t_u time for initially inserting it, and then t_u time for its removal, we get a cost of Mt_u for each purge performed. In addition, *TABLE* can contain up to M elements at the termination of the algorithm, elements which have been inserted but not yet deleted. This contributes at most Mt_u more to the cost. Since the total number of purges performed is no greater than $1+\log_2 b_n$, we get a memory manipulation charge of $Mt_u(2+\log_2 b_n)$ which gives us the second term above.

For the running time of the recovery algorithm, Lemmas 3 and 4 give us an upper bound in terms of b_n, which by Corollary 2 is at most $2n/M$. □

COROLLARY 3. *The total running time of the cycle finding algorithm is*

$$nt_f\left(1+O\left(\left(\frac{t_s}{t_f M}\right)^{1/2}\right)\right)$$

if g is chosen appropriately.

Proof. From Theorem 1, the total running time (of both algorithms) is bounded by

$$n\left(1+\frac{16g+14}{M}\right)\left(t_f+\frac{t_s}{g}\right)+t_u M\log_2\frac{8n}{M}.$$

Choosing g to be the square root of $\dfrac{Mt_s}{16t_f}\left(1+\dfrac{14}{M}\right)$ minimizes this expression, yielding

$$n\left(t_f+8\left(\frac{t_f t_s}{M}\left(1+\frac{14}{M}\right)\right)^{1/2}+\frac{14t_f}{M}+\frac{16t_s}{M}\right)+t_u M\log_2\frac{8n}{M}.$$

Combining terms that are $O(\sqrt{1/M})$ gives the stated result. □

Note, in particular, that a balanced tree implementation will have $t_s = O(\log M)$, and a hashing method could have $t_s = O(1)$. In both cases, the worst case running time will approach nt_f as M gets large. In the next section, we shall see that the algorithms are, in fact, optimal in a much stronger sense than this.

4. Optimality. The cycle detecting algorithm given above can be thought of as a family of algorithms (parameterized by g) that trade off between the two types of basic operations, namely, function evaluations and memory searches. Thus, in the range $1 \le g < M$, if we are willing to use $O(ng/M)$ extra function evaluations in addition to the n function evaluations needed to compute $f^n(x)$, then we need only perform $O(n/g)$ table searches. The corollary to Theorem 1 shows that g can be chosen to allow the cycle problem to be solved in time $nt_f(1+O(\sqrt{t_s/Mt_f}))$. In this section, we shall show these results to be optimal in the sense that no algorithm which does successive function evaluations and has a limited amount of memory to store function values can do substantially better.

We shall consider the problem of *cycle detection,* that is, finding a pair $i \neq j$ such that $f^i(x) = f^j(x)$. The lower bound results will also, of course, apply to the more general cycle finding problem. We need to specify the model of computation to be

considered.

The model. An algorithm A uses an array $T[1], \cdots, T[M]$, each cell of which can store a pair (k, d) with $k \geq 0$ an integer, and d an element of the domain D. At all times, any pair (k, d) stored in a cell will satisfy the relation $d = f^k(x)$; initially all cells contain $(0, x)$. The algorithm can make two types of moves: An *F-move* which picks a pair (i, j) of integers (possibly equal) and sets $T[i] \leftarrow (k, f(d))$ where (k, d) is the contents of $T[j]$ when the move is executed; and an *S-move* which picks an i and tests "is there a $j \neq i$ such that $T[i] = (k, d)$ and $T[j] = (k', d')$ satisfy $k' \neq k$ and $d' = d$". The computation proceeds one move at each time $t = 1, 2, \cdots$, and halts as soon as some S-move results in a "yes" answer. The algorithm is assumed to remember the entire history of the computation (that is, the values of i and j made in all F-moves, and the value of i used in all S-moves) and the choice of the next move can depend on all of this information. Of course, values of D are "remembered" only if they are currently stored in T. The reader will note that algorithms constructed for our model of computation are *oblivious* in that any one algorithm, when run on two different problem instances, will exhibit identical behavior up until the point that one of the computations receives a "yes" response to an S-move and halts.

For any instance (f,x) of the cycle problem, let $F_{(f,x)}(A)$ be the number of F-moves performed by A when run on that instance, and let $S_{(f,x)}(A)$ be the number of S-moves. The running time is thus $F_{(f,x)}(A)t_f + S_{(f,x)}(A)t_s$. We shall use the notation $n_{(f,x)}$ to denote the sum of the leader and cycle lengths for the instance (f,x).

The following theorem gives an explicit lower bound on the tradeoff required between the number of function evaluations and table searches for any algorithm for the cycle problem.

THEOREM 2. *Let k be a positive real and n_0 a positive integer. Suppose that A is an algorithm for the cycle problem, for which $F_{(f,x)}(A) < (1+k)n_{(f,x)}$ whenever $n_{(f,x)} \geq n_0$. Then*

$$S_{(f,x)}(A) > \frac{n_{(f,x)}}{8kM(1+4k)^2}$$

for all (f,x) with $n_{(f,x)}$ sufficiently large.

Proof. Consider the algorithm A working on an input (f,x) with $n_{(f,x)} = \infty$. Let t_m be the time when $f^m(x)$ is first computed by an F-move and stored into the array. Let $s(m, m')$ denote the number of S-moves performed in the time interval $[t_m, t_{m'})$. The method of proof will be to bound $s(m, m')$ for appropriate m, m', and then sum these results over a large range of intervals to prove the theorem.

First we shall show that if $m \geq n_0$ and $m' \geq (1+k)m$ then we must have

$$s(m, m') \geq \frac{m^2}{4(M-1)m'}.$$

To prove this, suppose that A has not yet halted at time $t_{m'}$. Then the S-moves made so far must have given enough information to establish that $f^m(x) \neq f^{m-c}(x)$ for $1 \leq c \leq m$, otherwise there would exist an instance (f,x) of the cycle problem on which A would perform more than $m' \geq (1+k)m = (1+k)n_{(f,x)}$ F-moves.

We shall proceed by determining how many inequalities must be found in order to discount those cycle sizes in the range $\alpha m \leq c \leq m$ for some α to be determined later. For each such c, let $f^{i_c}(x) \neq f^{j_c}(x)$ with $j_c < i_c < m'$ be the "witness" that $f^m(x) \neq f^{m-c}(x)$. Then $i_c - j_c = h_c c$ for some integer $h_c \geq 1$. Since $i_c < m'$ and $c \geq \alpha m$, we must have $h_c \leq m'/\alpha m$. Thus each inequality $f^i(x) \neq f^j(x)$ can eliminate at most $m'/\alpha m$ cycle sizes in the range $\alpha m \leq c \leq m$. Because each S-move can supply up to $M-1$ inequalities, we have

$$s(m, m')(M-1)\frac{m'}{\alpha m} \geq (1-\alpha)m.$$

Taking $\alpha = \frac{1}{2}$ yields the bound claimed in the statement of the theorem.

Next we shall bound the total number of S-moves made before time $t_{n_{(f,x)}}$. Since we already know that algorithm A cannot halt before time $t_{n_{(f,x)}}$, this will suffice to prove the theorem. Define $m_i = \lceil n_0(1+2k)^i \rceil$, $0 \leq i < \infty$. We may assume, without loss of generality, that $n_0 \geq 1 + 1/k$, and so, $1+k \leq m_{i+1}/m_i \leq 1+3k$. Thus we have

$$s(m_i, m_{i+1}) \geq \frac{m_i}{4M(1+3k)}.$$

For any $n_{(f,x)} > n_0$, define t to be the largest integer such that $m_t < n_{(f,x)}$. We thus have

$$S_{(f,x)}(A) \geq \sum_{0 \leq i < t} s(m_i, m_{i+1}) \geq \frac{1}{4M(1+3k)} \sum_{0 \leq i < t} n_0(1+2k)^i = \frac{1}{8Mk} n_0 \frac{(1+2k)^t - 1}{1+3k}.$$

If $n_{(f,x)}$ is sufficiently large, then t is large enough to guarantee that

$$\frac{(1+2k)^t - 1}{1+3k} > \frac{(1+2k)^t}{1+4k} > \frac{(1+2k)^{t+1}}{(1+4k)^2}.$$

Thus

$$S_{(f,x)}(A) > \frac{1}{8Mk} n_0 \frac{(1+2k)^{t+1}}{(1+4k)^2} \geq \frac{1}{8Mk} \frac{n_{(f,x)}}{(1+4k)^2},$$

which completes the proof. □

The reader should note that if our model of computation is altered to allow an unbounded supply of memory, and the basic operations changed to allow F-moves and simple comparisons, that is, in one step, test whether some specified pair of memory locations contain equal elements of D, then our proof techniques imply that any algorithm satisfying the hypotheses of Theorem 2 must perform at least $\frac{n_{(f,x)}}{8k(1+4k)^2}$ comparisons. It should also be noted that Theorem 2 has an alternate proof which is only valid when $k < \frac{1}{2}$ but which improves the constant in the bound on $S_{(f,x)}$. The alternate bound is $S_{(f,x)}(A) > \left(\frac{1-k}{k(1+k)M}\right) n_{(f,x)}$.

COROLLARY 4. *If $t_s/t_f = O(M)$, then the running time for any algorithm for the cycle detecting problem is*

$$nt_f\left(1+\Omega\left(\left(\frac{t_s}{t_f M}\right)^{1/2}\right)\right)$$

Proof. Let k be the square root of $t_s/t_f M$. For any cycle detecting algorithm, there are two cases:

Case 1. The algorithm performs more than $n(1+k)$ F-moves for infinitely many n. Thus the algorithm has a running time of at least

$$n(1+k)t_f = nt_f\left(1+\left(\frac{t_s}{t_f M}\right)^{1/2}\right)$$

for infinitely many n.

Case 2. The algorithm uses no more than $n(1+k)$ F-moves for all n greater than some n_0. In this case, Theorem 2 applies, and the running time of the algorithm is at

least

$$nt_f + \frac{nt_s}{8kM(1+4k)^2} = nt_f\left(1+\frac{1}{8(1+4k)^2}\left(\frac{t_s}{t_fM}\right)^{1/2}\right)$$

for all instances of the problem with $n_{(f,x)}$ sufficiently large. Since $t_s/t_f = O(M)$, k is bounded from above as M varies, and thus $1/(1+4k)$ is bounded from below. This gives us the claimed result. □

The above results are asymptotic statements about the performance of algorithms for the cycle problem for instances (f,x) with $n_{(f,x)}$ large. Of course, it is implicit in these statements that the size of the domain D must also be large since we have the constraint that $n_{(f,x)} \leq |D|$. If $|D|$ is known to be small, an algorithm might be able to make use of that information.

The lower bounds derived in this section shed some light on the possibility of extending further the algorithms in §2. The cycle finding algorithm has a running time of

$$\left(1+c_1\frac{g}{M}\right)nt_f + c_2\frac{nt_s}{g}$$

for small positive constants c_1, c_2, but only under the constraint that $g<M$. It is interesting to inquire whether algorithms can be found which extend this range.

For example, if one is willing to use many more, say Mn extra function evaluations, can the number of searches be lowered to $O(n/M^2)$? Theorem 2 provides part of the answer, since it says that with this many extra function evaluations, one still has to perform $\Omega(n/M^2)$ table searches. We do not know of any algorithm that achieves this lower bound.

Another direction of research involves finding a non-trivial lower bound on the number of function evaluations independent of the number of searches performed. It is easy to show, for example, that any algorithm which uses memory M must perform at least

$$\left(1+\frac{1}{2M}\right)n_{(f,x)} - 1$$

function evaluations in the worst case for sufficiently large $n_{(f,x)}$. This can be proved by considering the contents of memory at time t_{n-1} and choosing the cycle size so that $f^l(x)$ is at least $n/2M$ "away" from any stored element. Details are left to the reader.

5. Concluding remarks. We have dealt exclusively with algorithms with good worst case performance for solving a particular instance of the cycle problem on an unknown function. The problem is also interesting under other variations of the model.

One variation is to take the function to be *random* (in some sense) and to talk about an average case measure of complexity. R. W. Floyd has pointed out that studying the probabilistic structure of random functions over D can lead to savings on the average. For example, l is known to be relatively large in the case of a random function, so it may not be worthwhile to save or search for values at the beginning.

Anotner variation is to let the cost of computing $f^j(x)$ be independent of j. A famous factoring algorithm due to Shanks [4] is based on this problem. Shanks' solution, which uses the additional knowledge that $l=0$ and n is bounded by some constant N, finds n in time proportional to $\sqrt{n}$ using $\sqrt{n}$ memory. The method is similar to the dual algorithm mentioned in §2: one saves the first $\sqrt{n}$ values generated,

then does table searches at subsequent intervals of $\sqrt{n}$. Our cycle detecting algorithm also works in this case, but it runs slightly longer due to its need to discover that $l=0$ and its need to adapt to the value of c. A rough analysis for this case follows. From the program, we see that the function evaluation cost will be about the same as the table update cost, so the expression for the total running time in Theorem 1 tells us that the best thing to do is to pick g as large as possible (about M), for a running time of

$$O\left(\frac{nt_s}{M}+(t_u+t_f)M\log_2\frac{n}{M}\right).$$

This expression is minimized to $O(\sqrt{n\log_2 N})$ by choosing M to be $O(\sqrt{n/\log_2 N})$ if enough memory is available and we know that n is bounded by N.

A generalization of the cycle problem arises when all the points of the domain D are to be studied. In general, D is partitioned by f into disjoint sets with the property that all points in each set lead to the same cycle. Properties of the *cycle structure* (e.g., the number of sets, their sizes, the sizes of their cycles) can be found by solving the cycle problem on all points of D. The algorithms of this paper can be adapted to avoid retraversing long cycles by maintaining versions of *TABLE* for each cycle.

Another generalization of the cycle problem can be formulated in the following way. As before we are given a unary function f, only now we allow the domain of f to be infinite. We suppose that we are also given a binary predicate P on $D\times D$. The problem is to find the smallest n for which there exists an $l<n$ such that $P(f^n(x), f^l(x))$. In the absence of any further information, it is easy to show that this problem requires $\binom{n}{2}$ evaluations of P. However, if P is *preserved* by f, that is, $P(a, b)$ implies $P(f(a), f(b))$, then the algorithms of this paper can be made to run in time $n(t_f+O(t_p))$ where t_p is the time needed to evaluate P. It is interesting to note that the algorithms of [1] and [2] simply do not work for this problem.

An earlier version of this paper [3] left open the question of whether an algorithm exists for the cycle problem which used a bounded amount of memory and an optimal number of function evaluations. We have resolved this question in §4, with the somewhat surprising result that the algorithm of §2 may be viewed as optimal for the range of problem parameters for which it is applicable.

Appendix. In this appendix, we give a detailed proof of the complete result about the termination time of the cycle finding algorithm, Lemma 2 from §2 of the paper. The main purpose of including this proof in detail is that it precisely illustrates why the result given is the most general available for the problem: in fact, each of the cases below was essentially discovered as a counterexample during the search for a simpler or better version of Lemma 2.

A key fact needed to establish the correctness of the algorithm is that at least one complete block of searches on b consecutive values of the function is performed between any two consecutive calls on *purge*. Let the consecutive purges take place at $i=2^{k-1}M$ and 2^kM. During this time interval $b=2^k$. The following fact implies that at least one block of searches will be started early enough in this interval to be completed before the latter purge. More specifically, the search block will start no later than 2^kM-2^k.

FACT 1. *Let M and g be integers with $1\le g<M$ and $M\ge 2$. For any integer $k>0$ there exists an integer i, $2^{k-1}M\le i\le 2^kM-2^k$, such that $i\equiv 0 \pmod{2^kg}$.*

Proof. If $\lceil M/2\rceil\le g<M$, then 2^kg clearly has the required properties. If $1\le g\le \lfloor M/2\rfloor$, simply count the number of multiples of 2^k in the specified interval. If

M is even, there are precisely $M/2$ such multiples, whereas if M is odd, there are $(M-1)/2$ of them. In either case, the interval contains $\lfloor M/2 \rfloor$ consecutive multiples of 2^k so one of them must be congruent to $2^k g$. □

The next fact is a technical result which will be useful in determining the amount by which the algorithm can overshoot n.

FACT 2. *Let M, g, and x be positive integers. Then $\lceil M/g \rceil$ is even if and only if the smallest multiple of gx that is at least Mx is an even multiple of gx.*

Proof. (If) By hypothesis, there exists an even integer z such that $(z-1)gx < Mx \leq zgx$. But then, $z-1 < M/g \leq z$ and so $\lceil M/g \rceil \leq z$. Thus $\lceil M/g \rceil$ is even.

(Only if) By hypothesis, there exists an even integer z such that $z = \lceil M/g \rceil$. Then $z-1 < M/g \leq z$ and so $(z-1)gx < Mx \leq zgx$. Since $(z-1)gx$ and zgx are consecutive multiples of gx, zgx must be the smallest multiple of gx that is at least Mx. Since z is even, zgx is an even multiple of gx. □

At this point we are ready to prove the result of Lemma 2 from the text, which states precisely when the algorithm halts. The key idea is that termination is guaranteed to occur during the execution of the first block of consecutive searches initiated after the time when $i=n$. As mentioned in the text, the proof is complicated substantially by the necessity to account for the effects of purges which could delay the start of the last search block. Fact 1 will be used to show that the delay cannot be indefinite and Fact 2 will be used to establish a bound on the amount of the delay.

LEMMA 2. *Let b_n be n/M rounded up to the next power of 2 (i.e., b_n is the value of b when i is assigned the value of n). Then i_f, the value of i at the termination of the cycle detecting algorithm, obeys:*

$$n \leq i_f < n+(g+1)b_n \qquad \text{if } \lceil M/g \rceil \text{ is even,}$$
$$n \leq i_f < n+(2g+1)b_n \qquad \text{if } \lceil M/g \rceil \text{ is odd.}$$

Moreover, these bounds on i_f are as tight as possible.

Proof. Let $i_p = b_n M$, and let i_s be the (unique) multiple of gb_n for which $n \leq i_s < n+gb_n$. By Corollary 1, $i_p \geq n$. Thus i_p is the first moment after $i=n$ at which a purge operation can occur, and i_s is the first moment after $i=n$ at which a new block of search operations can commence. A number of cases now arise.

Case 1. $i_s < i_p$. Since i_s and i_p are both multiples of b_n, $i_s + b_n \leq i_p$ and the algorithm performs searchs for $f^i(x)$, $i_s \leq i < i_s + b_n$, during which time the value of b remains fixed at b_n. Since $i_s \geq n$, the values searched for are equal, respectively, to $f^j(x)$, $i_s - c \leq j < i_s + b_n - c$, exactly one of which, by Lemma 1(b), must be in *TABLE*. The algorithm therefore immediately terminates after one of these searches with $i_f < i_s + b_n < n + gb_n + b_n = n + (g+1)b_n$.

Case 2. $i_p \leq i_s$ and $\lceil M/g \rceil$ is even. By definition, i_s is the smallest multiple of gb_n that is at least as great as n. Since the condition of this case requires that $n \leq i_p \leq i_s$, i_s is the smallest multiple of gb_n that is at least $i_p = Mb_n$. Fact 2 thus guarantees that i_s is an even multiple of gb_n and hence $i_s \equiv 0 \pmod{2gb_n}$. Fact 1 guarantees that no additional purges take place between the times when $i=i_p$ and $i=i_s$. Thus the algorithm performs search operations for $f^i(x)$, $i_s \leq i < i_s + 2b_n$, during which time $b = 2b_n$. Since $i_s \geq n$, at least one of these function values is in *TABLE* and the algorithm halts with $i_f < i_s + 2b_n$. Thus $n \leq i_f < n + (g+2)b_n$.

We shall conclude this case by demonstrating by contradiction that we must actually have $i_f < n + (g+1)b_n$. To do this, suppose that $i_f \geq n + (g+1)b_n$. Consider $i_{ps} = i_s - gb_n$ and $i_{pf} = i_f - gb_n - b_n$. i_{ps} is the point where the previous search block began. We shall show that a previous find would have occurred at i_{pf}, terminating the

algorithm and giving us the desired contradiction. Observe that $i_{ps} < n$ because $i_s < n + gb_n$. Moreover, $n \le i_{pf}$ because $n + (g+1)b_n \le i_f$. Notice that $i_{pf} < i_{ps} + b_n$ because $i_f < i_s + 2b_n$. Finally, observe that $i_{ps} + b_n \le i_p$ because both i_{ps} and i_p are multiples of b_n with $i_{ps} < n \le i_p$. Putting these together we have $i_{ps} < n \le i_{pf} < i_{ps} + b_n \le i_p$. Since $i_{ps} \equiv 0 \pmod{gb_n}$, this implies that the algorithm performs searches for $f^i(x)$, $i_{ps} \le i < i_{ps} + b_n$ during which time b remains constant at b_n. By definition of i_f, $i_f - c \equiv 0 \pmod{2b_n}$, and so $i_{pf} - c \equiv 0 \pmod{b_n}$. Thus the search for $f^{i_{pf}}(x)$ should have caused the algorithm to terminate with $i = i_{pf}$.

Case 3. $i_p \le i_s$ and $\lceil M/g \rceil$ is odd. As in case 2 above, i_s is the smallest multiple of gb_n that is at least i_p. This time, however, Fact 2 tells us that i_s is an odd multiple of gb_n and hence $i_s \not\equiv 0 \pmod{2gb_n}$. The first block of searches starting after $i = n$ must therefore begin at $i_s + gb_n$. Since b at this time is $2b_n$, we see that $i_f < i_s + gb_n + 2b_n < n + (2g+2)b_n$.

As in case 2, we can next argue that the last b_n values in this range are not really possible. To do this, suppose that $i_f \ge n + (2g+1)b_n$. Consider $i_{ps} = i_s - gb_n$ and $i_{pf} = i_f - 2gb_n - b_n$. Once again, algebra reveals that $i_{ps} < n \le i_{pf} < i_{ps} + b_n \le i_p$ and we can argue that the algorithm would have terminated earlier.

To see that the stated bounds are the tightest possible, let us suppose that M and g are given, with $1 \le g < M$. We shall show how to construct an infinite set of instances of the cycle finding problem in which i_f is at the extreme high end of the ranges given in the statement of the lemma.

For any integer $k \ge 0$, let i_p be $2^k M$, and let i_{s_1} be the largest multiple of $2^k g$ that is less than i_p. Let n be $i_{s_1} + 2^k$. Since both n and i_p are multiples of 2^k, $n \le i_p$ and $b_n = 2^k$.

If $\lceil M/g \rceil$ is even, let i_{s_2} be $i_{s_1} + 2^k g = n + gb_n - b_n$ and let l be $1 + gb_n - b_n$. If $\lceil M/g \rceil$ is odd, let i_{s_2} be $i_{s_1} + 2^{k+1} g = n + 2gb_n - b_n$ and let l be $1 + b_n$. These choices for l are possible because n is at least $gb_n + b_n$. It can be shown through the use of Fact 2 that, in either case, i_{s_2} is the smallest multiple of $2^{k+1} g$ that is at least i_p. Now consider the operation of the algorithm on an instance of the cycle finding problem whose solution is given by the n and l defined above. The algorithm will perform searches for $f^i(x)$, $i_{s_1} \le i < i_{s_1} + b_n = n$. These searches all fail. A purge then occurs at i_p increasing b to $2b_n = 2^{k+1}$. The next block of searches is performed for $f^i(x)$, $i_{s_2} \le i < i_{s_2} + 2b_n$. Since the cycle size $c = n - l$ is sufficiently large, the algorithm will terminate for the first i with $i_{s_2} \le i < i_{s_2} + 2b_n$ and for which $i - c \equiv 0 \pmod{2b_n}$. Let us write this i_f as $i_{s_2} + j$ with $0 \le j < 2b_n$.

If $\lceil M/g \rceil$ is even, then $i_f - c = i_{s_2} + j - c = n + gb_n - b_n + j - c = l + gb_n - b_n + j = 1 + 2gb_n - 2b_n + j$. Thus $i_f - c \equiv 0 \pmod{2b_n}$ when $j = 2b_n - 1$ and the search halts at $i_f = i_{s_2} + 2b_n - 1 = n + (g+1)b_n - 1$. If $\lceil M/g \rceil$ is odd, then $i_f - c = i_{s_2} + j - c = n + 2gb_n - b_n + j - c = l + 2gb_n - b_n + j = 1 + 2gb_n + j$. Thus $i_f - c \equiv 0 \pmod{2b_n}$ when $j = 2b_n - 1$ and the search halts at $i_f = i_{s_2} + 2b_n - 1 = n + (2g+1)b_n - 1$. In either case, i_f is the largest value permitted in the ranges given in the statement of the lemma. □

Acknowledgments. The authors take pleasure in thanking A. V. Aho, W. Beckman, and M. D. McIlroy for their helpful comments on various drafts of this paper.

Postscript. F. E. Fich has recently shown [5] that the cycle problem is also interesting to study under a complexity measure that counts only the number of function evaluations. Memory references are not counted explicitly in the cost, but

algorithms must respect the limitation of using only a fixed amount of memory. She proves a lower bound of $n(1+1/(M-1))$ function evaluations for algorithms restricted to M memory cells, while the algorithm in this paper uses $n(1+2/(M-1))$ function evaluations. Also, she gives upper and lower bounds for $M=2$ and various other restrictions.

REFERENCES

[1] D. E. KNUTH, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms,* Addison-Wesley, New York, 1969.

[2] R. GOSPER ET AL., *HACKMEM,* M.I.T. Artificial Intelligence Lab Report No. 239, 1971.

[3] R. SEDGEWICK AND T. G. SZYMANSKI, *The complexity of finding periods,* Proc. 11th Annual ACM Symp. on the Theory of Computing, (April 1979), pp. 74-80.

[4] D. SHANKS, *Class number, a theory of factorization, and genera,* Proceedings of Symposia in Pure Mathematics, American Mathematical Society, Providence, Rhode Island, 1970.

[5] F. E. FICH, *Lower bounds for the cycle detection problem,* Proc. 13th Annual ACM Symposium on the Theory of Computing, (May 1981), pp. 96-105.