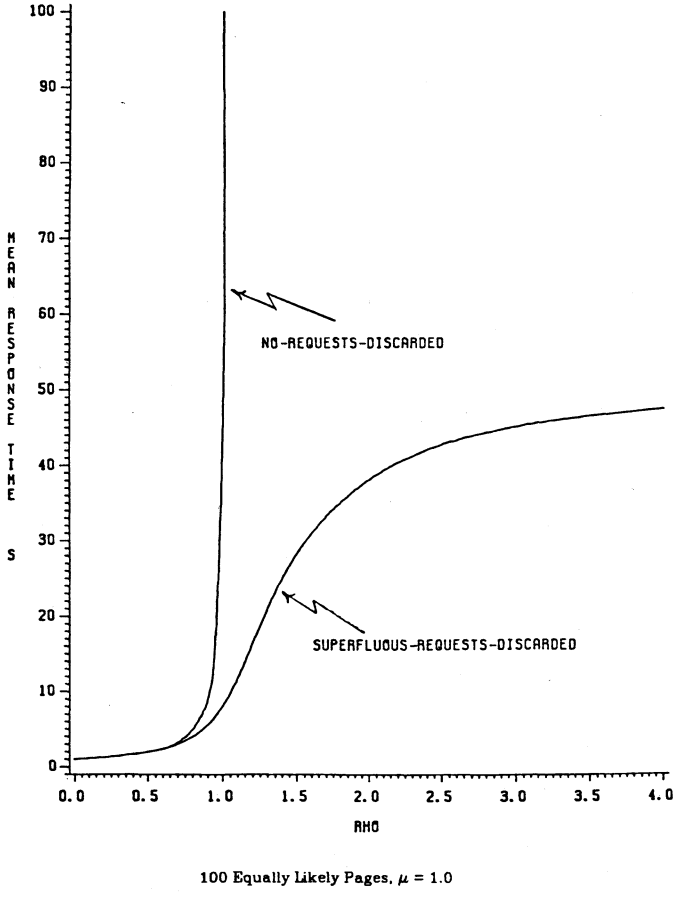

Fig. 5. A comparison of systems with and without superfluous request discarding.

V. Numerical Examples and Discussion

The analytic results in Section III are for a general request probability distribution. We focus here, for simplicity, on the case where each page is equally likely to be requested, i.e., $q_i = 1/N$ for all $i = 1, \cdots, N$. Note that, for this case, $S = S_i$ for any i.

Fig. 4 shows how S varies as a function of N for various values of ρ. We observe the following two properties.

1) For $\rho \geq 1$, the mean response time is a linear function of N as $\rho \to \infty$. (See Section III-C.)

2) For $\rho < 1$, as N increases, the mean response time approaches a constant. For the case $q_i = 1/N$, it can be shown that (8) yields a mean response time given by the $M/M/1$ result, i.e., $1/(\mu(1 - \rho))$ when N is large. This can be explained by the fact that when q_i is small for all i, the probability that an arriving request is superfluous approaches zero. Note that if q_i does not decrease with N for some i, then the mean response time at large N is different from that of the $M/M/1$ model.

Fig. 5 shows the advantage of discarding superfluous requests. In [5] analytic results were obtained for the case where superfluous requests are not discarded. These results can be summarized as follows:

$$S_i = \frac{1}{\mu(1 - \rho(1 - q_i))} \tag{12}$$

and

$$S = \sum_{i=1}^{N} \frac{q_i}{\mu(1 - \rho(1 - q_i))}. \tag{13}$$

We observe that discarding superfluous requests leads to better mean response time. At heavy load, the queue becomes unbounded if superfluous requests are not discarded. The limit of the mean response time in (13) as $\rho \to 1$ (condition of a saturated system when superfluous requests are not discarded) is N/μ. This is approximately twice the heavy load limit in (9) for the case where superfluous requests are discarded.

References

[1] W. H. Ninke, "Interactive picture information systems—Where from? Where to?" *IEEE J. Select. Areas Commun.*, vol. SAC-1, Feb. 1983.
[2] J. Martin, *Viewdata and the Information Society*. Englewood Cliffs, NJ: Prentice-Hall, 1982.
[3] J. Gecsei, *The Architecture of Videotex Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
[4] S. S. Lam, "Queueing networks with population size constraints," *IBM J. Res. Develop.*, vol. 21, no. 7, July 1977.
[5] M. H. Ammar and J. W. Wong, "Analysis of broadcast delivery in videotex systems," Comput. Commun. Networks Group, Univ. Waterloo, Waterloo, Ont., Canada, Rep. E-118, Jan. 1984.
[6] J. P. Buzen, "Computational algorithms for closed queueing networks with exponential servers," *Commun. ACM*, vol. 16, no. 9, Sept. 1973.
[7] M. Reiser and H. Kobayashi, "Queueing networks with multiple closed chains: Theory and computational algorithms," *IBM J. Res. Develop.*, vol. 19, no. 3, May 1975.
[8] S. Bruell and G. Balbo, *Computational Algorithms for Closed Queueing Networks*. Amsterdam, The Netherlands: North-Holland, 1980.

Fast Constant Division Routines

SHUO-YEN ROBERT LI

***Abstract*—When there is no division circuit available, the arithmetical function of division is normally performed by a library subroutine. The library subroutine normally allows both the divisor and the dividend to be variables, and requires the execution of hundreds of assembly instructions. This correspondence provides a fast algorithm for performing the integer division of a variable by a predetermined divisor. Based upon this algorithm, an efficient division routine has been constructed for each odd divisor up to 55. These routines may be implemented in assembly languages, in microcodes, and in special-purpose circuits.**

Applications of the algorithm to programs in the assembly language of the *Traffic Service Position Systems No. 1 and No. 1B*, which process 90 percent of all operator-assisted telephone traffic in the United States, have shown a reduction in the usage of processor time by two orders of magnitude. This real-time improvement made a major feature in the two systems feasible and also enhanced the efficiency of other features.

***Index Terms*—Constant division, fast division, Fermat's Little Theorem.**

I. Introduction

When there is no division circuit available, the arithmetical function of division is normally performed by a library subroutine. The precoded subroutine normally assumes that both the dividend and

Manuscript received August 8, 1983; revised May 11, 1984 and August 22, 1984.

The author is with Bell Communications Research, Murray Hill, NJ 07974.

the divisor are variables stored in separate registers, and each invocation of the subroutine normally requires the execution of hundreds of instructions. Often a division subroutine is called for dividing a variable by a predetermined constant. For example, the division can be a change of scale, or it may be the division by a constant prescribed in a mathematical equation. For this kind of division, the processor time usage can be reduced if a division subroutine is provided for each often-used divisor. This correspondence provides an algorithm for writing fast subroutines for this purpose. The algorithm may be implemented in assembly languages, in microcodes, and in the design of special-purpose circuits. The mathematical basis of the algorithm is a well-known theorem of *Euler* which is a generalization of *Fermat's Little Theorem*.

Section III contains a list of fast division routines constructed from the algorithm for small integer divisors. These routines have been translated into a switching assembly programming (SWAP) version and widely applied to the *Traffic Service Position Systems No. 1* [4] *and No. 1B* [5], which process 90 percent of all operator-assisted telephone traffic in the United States. A typical reduction in processor-time usage by two orders of magnitude has been achieved. This real-time improvement made a major feature in the two systems feasible and also enhanced the efficiency of other features.

II. The Basic Algorithm

For any positive integer p, let $n(p)$ be the number of positive integers smaller than p that are relatively prime to p. The function $n()$ is called the *Euler's* function. The following theorem is due to *Euler*. (See, for example, [2, sec. 2.3].)

Theorem 2.1: Let the integer c be relatively prime to p. Then

$$c^{n(p)} = 1 \bmod p . \tag{1}$$

The special case of the theorem when p is prime is called *Fermat's Little Theorem*. The next theorem can be verified by straightforward algebra.

Theorem 2.2: Let n be an integer and p be an odd integer that is greater than 1 and divides $2^n - 1$. Write $(2^n - 1)/p$ as $Binary(b_1b_2 \cdots b_{n-1})$. Then

$$\frac{1}{p} = Binary(0.0b_1b_2 \cdots b_{n-1}) \prod_{i=0}^{\infty} (1 + 2^{-n\cdot 2^i}) . \tag{2}$$

In particular, Theorem 2.2 holds when n is the smallest integer such that p divides $2^n - 1$. Take $c = 2$ in (1). Theorem 2.1 guarantees, for every odd integer p, the existence of such a number $n < p$. Now suppose that one wants to construct a subroutine for the division of an integer variable x by p from assembly or microinstructions. (These typically are shifts, addition, subtraction, and binary logical function, etc., when multiplication and division circuits are not available.) Equation (2) shows that division by p is equivalent to multiplication by $Binary(0.0b_1b_2 \cdots b_{n-1})$ followed by multiplications by factors of the form $1 + 2^{-m}$.

The multiplication by $1 + 2^{-m}$ means a shift-and-add. In certain assembly languages, a shift-and-add takes only one machine cycle, which is as little processor time as the no-operation instruction takes. This fact is exemplified by the SWAP language, in which most of the electronic switching machines deployed in the United States are programmed.

To illustrate how to compute x/p from this approach, let $p = 5$ and $n = 4$. Then $b_1b_2b_3 = 011$. Equation (2) in this special case yields the following infinite product:

$$\frac{x}{5} = \left(\frac{x}{2^3} + \frac{x}{2^4}\right)\left(1 + \frac{1}{2^4}\right)\left(1 + \frac{1}{2^8}\right)\left(1 + \frac{1}{2^{16}}\right)\cdots \tag{3}$$

The conventional way of computing $x/5$ is by the approach of infinite series:

$$\frac{x}{5} = x \cdot Binary(0.001100110011\cdots)$$

$$= \frac{x}{2^3} + \frac{x}{2^4} + \frac{x}{2^7} + \frac{x}{2^8} + \frac{x}{2^{11}} + \frac{x}{2^{12}} + \cdots \tag{4}$$

Computation from the infinite series expression up to the term of $x/2^{32}$ in the straightforward manner requires 16 shifts and 15 additions. As a contrast, computation of the same approximation from the infinite product expression requires only 5 shifts and 4 additions.

III. Implementation

Equation (2) offers a basic algorithm to convert the division by p into a series of shift-and-adds. The actual implementation of the algorithm allows a wide range of variations because of the following reasons. First, multiplication by $Binary(0.0b_1b_2 \cdots b_{n-1})$ can be translated into shift-and-adds in more than one way. Second, each shift-and-add can be adjusted by a (positive and negative) power of 2 in order to avoid register overflow or to minimize the truncation loss. Finally, there is no definite rule on how to most efficiently compensate for truncation. Given the value of p and a range of the input value (the dividend), one can always search for the most efficient implementation through trial and error that avoids register overflow and properly compensates for truncation.

Take the example when $p = 19$. The smallest integer n such that 19 divides $2^n - 1$ is $n = 18$. Equation (2) yields the following approximation:

$$\frac{1}{p} \approx Binary(0.0b_1b_2 \cdots b_{17})(1 + 2^{-18}) \tag{5}$$

where $b_1b_2 \cdots b_{17} = 00011010111100101$. Often, when an odd integer divides $2^{2k} - 1$, it also divides $2^k + 1$. Indeed, 19 does divide $2^9 + 1$. Therefore, the binary number $b_1b_2 \cdots b_{17}$ contains the factor $2^9 - 1$. Similar facts hold when the divisor p is taken to be 5, 9, 11, 13, or 17 instead. The approximation formula (5) can now be written as

$$\frac{x}{p} \approx x\, Binary(11011)(2^9 - 1)(1 + 2^{-18})2^{-18}. \tag{6}$$

Multiplication by $Binary(11011)$ can be converted into three shift-and-adds, either in the straightforward manner or by *Booth's* algorithm. But one can actually do better by recognizing the factorization.

$$Binary(11011) = Binary(11)Binary(1001) . \tag{7}$$

In general, if the binary pattern of the multiplier or part of the pattern contains a factor of the form $2^k \pm 1$, the multiplication very likely can be translated into fewer shift-and-adds than *Booth's* algorithm would require. Substitution of (7) into (6) yields the following routine:

$$x \leftarrow x \cdot 2 + x$$

$$x \leftarrow x \cdot 2^3 + x$$

$$x \leftarrow x \cdot 2^9 - x$$

$$x \leftarrow [x \cdot 2^{-18}] + x$$

$$x \leftarrow [x \cdot 2^{-18}] .$$

Here the symbol [ ] is the usual notation for *integer part*. The result of the above procedure does not always turn out to be the desired value [(input x)/19] because of truncation errors. The content of x register after the first four instructions is approximately 13 824 times the input value. The concern of overflow then places a strict ceiling on the input value. Through trial and error, the following modified routine has been constructed:

$$x \leftarrow x \cdot 2 + x + 2$$
$$x \leftarrow [x \cdot 2^{-3}] + x$$
$$x \leftarrow x - [x \cdot 2^{-9}]$$
$$x \leftarrow [x \cdot 2^{-18}] + x$$
$$x \leftarrow [x \cdot 2^{-6}].$$

Note that the directions of shifts in the second and third instructions have been reversed. A compensation for truncation loss is appended to the first instruction. The accuracy of this routine has been verified for all input values of x between 0 and 500 000 000 on a processor with 32 bit hardware registers. Theoretically, this routine will eventually fail for some large value of x on any processor either by overflowing a register or by missing the precise value by 1. The problem of overflow starts at the input values 310 689, 4 971 027, and 1 272 582 902 on 20, 24, and 32 bit machines, respectively. In general, the threshold of the input value to the above routine is approximately three-fifths of the largest positive content of a register, or approximately three-fifths of 2^{k-1} on a k bit machine.

Table I is a list of efficient routines for division by all odd integers up to 55. The dividend is the input contained in X register. The quotient, which is the output, is also contained in X register. The remainder is not computed. Content of Y register is scratched if the register is used. "L" and "R" stand for left shift and right shift, respectively. They are immediately followed by the distance of shifts. For each routine, there is a critical ceiling on the input value beyond which a register overflow will occur amid the computation. The lowest ceiling among all routines in Table I is approximately two-fifths of the largest possible positive content of a hardware register. This translates into about 200 000 with 20 bit registers or 3 300 000 with 24 bit registers. The output in X register gives the precise quotient for any input value up to at least a few million. Beyond this range any computational error will be minimal unless overflow occurs.

TABLE I
FAST DIVISION ROUTINES BY SMALL ODD DIVISORS

Divisor	Division Procedure		
3	X ← XL2+X+5 X ← XR16+X	X ← XR4+X X ← XR4	X ← XR8+X
5	X ← XL1+X+3 X ← XR16+X	X ← XR4+X X ← XR4	X ← XR8+X
7	X ← X+1 X ← XR24+X	X ← XL2+XR1 X ← XR5	X ← XR6+X
9	X ← X+1 X ← XR12+X	X ← XL1+X+XR1 X ← XR24+X	X ← XR6+X X ← XR5
11	X ← X+1 X ← XR10+X	Y ← XL2+X X ← XR20+X	X ← Y+YR4+XR1 X ← XR6
13	X ← X+1 X ← XR12+X	X ← XL2+XR1 X ← XR24+X	X ← X+(XR1+X)R4 X ← XR6
15	X ← X+1 X ← XR16+X	X ← XL2+XR2 X ← XR6	X ← XR8+X
17	X ← XL2+X+5 X ← XR16+X	X ← XR1+X X ← XR7	X ← XR8+X
19	X ← XL1+X+2 X ← XR18+X	X ← XR3+X X ← XR6	X ← X-XR9
21	X ← XL1+X+3 X ← XR24+X	X ← XR6+X X ← XR6	X ← XR12+X
23	X ← X+1 X ← XR22+X	X ← (XR3+X)R1+X+XL2 X ← XR7	X ← XR11+X
25	Y ← X+1 X ← XR20+X	X ← YL2+X+YR3 X ← XR7	X ← X-XR10
27	Y ← XL1+X+15 X ← XR18+X	Y ← YR2+XL2 X ← XR7	X ← X-XR9
29	Y ← XR2+XL2+3 X ← X-XR14	Y ← YR5+Y X ← XR28+X	X ← XR5+Y X ← XR7
31	X ← X+1 X ← XR20+X	X ← XL2+XR3 X ← XR7	X ← XR10+X
33	X ← X+1 X ← XR3+Y X ← XR7	Y ← XL1+X X ← XR10+X	Y ← YR2+Y X ← XR20+X
35	X ← X+1 X ← XR3+X X ← XR7	Y ← XL1+X X ← XR12+X	X ← XR2+Y X ← XR24+X
37	X ← XL2+X+4 X ← XR2+Y	Y ← XR3+X X ← X-XR18	Y ← YR7+Y X ← XR8
39	X ← X+1 X ← XR5+X X ← XR7	Y ← XL1+X X ← XR12+X	Y ← XR2+Y X ← XR24+X
41	Y ← XL1+X+2 X ← XR20+X	X ← XR3+Y X ← XR7	X ← X-XR10
43	X ← XL1+X+2 X ← XR28+X	X ← X-XR7 X ← XR7	X ← XR14+X
45	X ← X+1 X ← XR4+Y X ← XR8	Y ← XL2+X X ← XR12+X	Y ← YR3+Y X ← XR24+X
47	X ← (X+9)R1+X+XL2-XR4 + (((XR4+X)R1+X)R3+X)R7	X ← XR23+X	X ← XR8
49	X ← XL2+X+5-XR7 + (XR4+X)R2	X ← XR21+X	X ← XR8
51	X ← XL2+X+5 X ← XR8	X ← XR8+X	X ← XR16+X
53	X ← X+1 Y ← Y+YR6 Y ← Y+XR16 X ← XR8	Y ← X+XR9 Y ← XR2-Y X ← Y+XR18	Y ← XL2+Y Y ← XR14-Y X ← X-XR26
55	Y ← X-XR3 Y ← X+YR2 X ← XR6	Y ← X-YR4+148 X ← X+YR3	Y ← X+YR2 X ← XR20+X

IV. DISCUSSIONS

Algorithms for division by integer constants have previously been considered in [3] and [1]. This section briefly describes these previous algorithms and compares them with the algorithm in this correspondence.

Assume the input register is in m bits containing the dividend x. Let k be smallest such that $2^k - 1$ is divisible by the fixed divisor p. Let n be a multiple of k that exceeds m. One can then write

$$2^n - 1 = ps \tag{8}$$

where s corresponds to $Binary(b_1 b_2 \cdots b_{n-1})$ in Theorem 2.2. Suppose the result of dividing x by p would yield the quotient q and the remainder r. Then

$$-xs = q - rs - 2^n q. \tag{9}$$

Since $0 \le x \le 2^m - 1 \le 2^n - 1$, x is divisible by p if and only if $-xs$ modulo 2^n equals $0, 1, \cdots$, or $s - 1$. When this is the case, $-xs$ modulo 2^n is the quotient q. This provides a fast technique to determine whether x is divisible by p and, if it is, to compute the quotient. The algorithm in [1] is to apply this procedure iteratively to $x, x - 1, x - 2, \cdots$, until an even division is determined. The number of iterations minus 1 is the remainder r. This algorithm computes both q and r with an average computational complexity of $O(p)$.

The algorithm in [3] is based upon the equation

$$\frac{1}{p} = s2^{-n} \sum_{j=0}^{\infty} 2^{-jn}. \tag{10}$$

This differs from (2) by being an infinite series rather than an infinite product. The asymptotic computational complexity in computing an approximate value of $1/p$ based upon (10) is proportional to the length of precision, while the approach from (2) has a logarithmic complexity. The advantage of using an infinite product over an infinite series has also been pointed out in [1].

V. CONCLUSION

The present algorithm turns the technique of approximation via an infinite product into a method of computing the precise value of the quotient (without computing the remainder). Variations in the im-

plementation of the algorithm do not change the theoretic asymptotic complexity. However, in real applications, such as the case of TSPS No. 1 and TSPS No. 1B, optimizations in the implementation greatly enhance the usefulness of the algorithm. This correspondence demonstrates some of the optimization techniques and compiles a table of practical division routines that are fine-tuned toward a seven-digit range of input values.

REFERENCES

[1] E. Artzy, J. A. Hinds, and H. J. Saal, "A fast division technique for constant divisors," *Commun. ACM*, vol. 19, no. 2, pp. 98–101, 1976.
[2] P. M. Cohn, *Algebra, Vol. 1*. New York: Wiley, 1974.
[3] D. H. Jacobsohn, "A combinatoric division algorithm for fixed-integer divisors," *IEEE Trans. Comput.*, vol. C-22, pp. 608–610, June 1973.
[4] R. J. Jaeger, Jr. and A. E. Joel, Jr., "TSPS no. 1: System organization and objectives," *Bell Syst. Tech. J.*, vol. 49, no. 10, pp. 2417–2444, 1970.
[5] R. E. Staehler and J. I. Cochrane, "Traffic service position system no. 1B: Overview and objectives," *Bell Syst. Tech. J.*, vol. 62, no. 3, pp. 755–764, 1983.

An Approximation Algorithm for Diagnostic Test Scheduling in Multicomputer Systems

HENRYK KRAWCZYK AND MAREK KUBALE

Abstract **— The problem of diagnostic test scheduling (DTS) is to assign to each edge e of a diagnostic graph G a time interval of length $l(e)$ so that intervals corresponding to edges at any given vertex do not overlap and the overall finishing time is minimum. In this correspondence we show that the DTS problem is NP-complete. Then we present a longest-first sequential scheduling algorithm which runs in worst case time $O(dm \log n)$ and uses $O(m)$ space to produce a solution of length less than four times optimal. Then we show that the general performance bound can be strengthened to $3 \cdot \mathrm{OPT}(G)$ for low-degree graphs and to $2 \cdot \mathrm{OPT}(G)$ in some special cases of binomial diagnostic graphs.**

Index Terms **— Approximation algorithm, computational complexity, diagnostic graph, diagnostic test scheduling, multicomputer system, NP-completeness.**

I. INTRODUCTION

The recent development of VLSI technology has made multicomputer systems economically attractive for practical applications. However, there remain still some open problems that must be solved in forthcoming years. One of them is scheduling of diagnostic testing of one unit by another during the maintenance period of the systems.

It is well-known that multicomputer systems may react on a failure in different ways. Some of them, for example, are able to mask errors and stop their propagation inside the system to operate continuously. Others can recognize their state and reconfigure themselves, especially in case of error localization. Apart from this a periodical self-diagnosis is done to prevent uncovered errors from causing graduated or total failure of the system. For general computation the diagnostic tests should be executed as fast as possible. The frequency of periodical testing depends on requirements of practical application, on the system failure rate, etc.

In this correspondence we consider nonhomogenous systems and assume that the periodical self-diagnosis is scheduled by a central controller and performed off-line. It means that during this process user tasks are suspended (because only "nonbusy" units can take part in it). Saheban and Friedman noticed that not all units of a real time system must be assigned to the diagnosis simultaneously (cf. [4], [10]). According to their assignment strategy some units are "busy," i.e., are assigned to computation, while the others are "nonbusy," i.e., are assigned to self-diagnosis, and therefore, computation and diagnosis are performed concurrently in real time. The assignment is dynamic in the sense that each unit must turn into nonbusy status to provide a complete system diagnosis. The general scheduling for such a system consists of unit, task, and test schedulers (see Fig. 1). Based on environment circumstances, the unit scheduler divides the system into busy and nonbusy units. The task scheduler assigns units to modules of a given user task. The test scheduler controls nonbusy units during self-diagnosis.

To perform self-diagnosis, nonbusy units are usually assumed to diagnose in pairs. It means that in each diagnostic pair one unit is tested by the other. Clearly, the duration of a particular testing depends on units involved in the diagnostic link. Hence, all tests that result from a current assignment of nonbusy units can be modeled by a *diagnostic (testing) graph* $G = (V, E; l)$ where the vertex set $V = \{v_1, \cdots, v_n\}$ corresponds to the set of nonbusy units, the edge set $E = \{e_1, \cdots, e_m\}$ corresponds to the set of tests, and the weighting function l: $E \rightarrow R^+$ assigns to each edge $e = \{u, v\}$, $u, v \in V$ a positive real number $l(e)$ representing the length of time taken for unit u to test v or for both units to test mutually (e.g., if they are matched for comparison testing [1], [7]). Since between any pair of vertices there are at most two edges, the diagnostic graph can be viewed as a weighted 2-graph.

For technical reasons self-diagnosis must meet the following time constraints.

- Any unit tests (or is tested by) at most one of the other units simultaneously.
- Any test is nonpreemptive (i.e., a test is not interrupted once its execution has begun).

The above constraints, together with the fact that a diagnostic testing is realized by a pair of units rather than by a single unit, constitute the main difference in scheduling between the module of user tasks (job shop scheduling) and the module of diagnostic tasks.

Let $G = (V, E; l)$ be a diagnostic graph induced by nonbusy units of a system. Then the problem of *diagnostic test scheduling* (DTS) is to assign to each edge $e \in E$ a time interval of length $l(e)$ so that intervals corresponding to edges at any given vertex do not overlap except, possibly, in common endpoints. The schedule resulting in the shortest finishing time for the whole graph is desired. In Section II we give a simple proof that the DTS problem is NP-complete. In the succeeding section we present an approximation algorithm which sequentially schedules a longest nonscheduled edge during the earliest possible time interval and shows that our longest first (LF) algorithm runs in worst case time $O(dm \log n)$, expected time $O(dm)$, and uses $O(m)$ space. In Section IV we analyze performance guarantees for LF and prove that the algorithm produces a solution of length less than $4(d-1)/d$ times optimal for any graph G with $d > 1$. Here and above d is the maximum degree of any vertex, m is the number of edges, and n is the number of vertices in the diagnostic graph. We also strengthen the overall performance bound to $3 \cdot \mathrm{OPT}(G)$ for graphs having no vertex of degree exceeding 5 and to $2 \cdot \mathrm{OPT}(G)$ for binomial graphs with

Manuscript received July 5, 1983; revised February 6, 1984.

The authors are with the Institute of Informatics, Technical University of Gdańsk, 80-952 Gdańsk, Poland.

0018-9340/85/0900-0869$01.00