Missouri University of Science and Technology
Scholars' Mine

Computer Science Technical Reports | Computer Science

01 Jul 1985

A Monte Carlo Analysis of the Mincycle Algorithm for Generating Minimal Perfect Hash Functions

Pao-Po Hou

Thomas J. Sager

Follow this and additional works at: https://scholarsmine.mst.edu/comsci_techreports

 Part of the Computer Sciences Commons

Recommended Citation

Hou, Pao-Po and Sager, Thomas J., "A Monte Carlo Analysis of the Mincycle Algorithm for Generating Minimal Perfect Hash Functions" (1985). *Computer Science Technical Reports*. 6.
https://scholarsmine.mst.edu/comsci_techreports/6

This Technical Report is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

A MONTE CARLO ANALYSIS OF THE MINCYCLE ALGORITHM FOR GENERATING MINIMAL PERFECT HASH FUNCTIONS

Pao-Po Hou* and Thomas J. Sager

CSc-85-3

Department of Computer Science

University of Missouri-Rolla

Rolla, Missouri 65401 (314) 341-4491

*This report is substantially the M.S. thesis of the first author, completed July 1985.

ABSTRACT

In this paper, several minimal perfect hashing function generating methods are examined. One of them, the mincycle method by Sager is evaluated by the Monte Carlo method. The results are represented in graphs and tables.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

I. INTRODUCTION

Monte Carlo (sampling) method[1] is one of the most basic techniques used in digital computer simulation. It is used to draw values from a pool of possible ones as input fed into the simulation program. This pool is called the sample space, and values are called samples. Each sample in the sample space is assigned a probability, which determines the frequency or the likelihood that it be drawn. In this application a random number generator is used to obtain these values according to their probabilities. The result of the simulation program then describes the system being simulated under various circumstances.

This technique exploits the nature of the cumulative distribution function F(y) of a random variable (sample) y to generate a value of y. The cumulative distribution function is a function that gives the probability of a value of y less than or equal to a specified value c; ie.

$$P(y<=c) = F(c)$$

Note that the range of the quantity F(c) is $0 <= F(y) <= 1$. In Monte Carlo method a number r in the set of F(c)'s is randomly chosen. The value of y corresponding to this particular value of $F(y) = r$ is the desired value of the random variable (sample) y. For example, suppose we want to simulate the outcome of throwing a die. Now the cumulative distribution function is:

$$F(y) = \begin{cases} y/6 & y \in \{1, 2, 3, 4, 5, 6\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

For example, if $(i-1)/6 <= r < i/6$, where $i \in \{1,2,3,4,5,6\}$, then $y = i$ is the corresponding value to r. As another example, we can use the Monte Carlo method to reach an approximate value of Π. Before we begin we bear in mind that the ratio between the area of a circle of radius r and the area of a square with length of side r is Π. So the ratio between areas enclosed by equations:

$y = \text{square-root-of}(1 - x^2)$
$y = 0$
$x = 0$

and

$y = 0$
$y = 1$
$x = 0$
$x = 1$

is Π/4. Then we use the Monte Carlo method to sample points in region $\{(x, y) \mid 0 <= x < 1, 0 <= y < 1\}$ with uniform probability

density. Now the number of sample points satisfying the inequility:

$$y <= \text{square-root-of}(1 - x^2)$$

to that of total sample points taken is an approximation of $\Pi/4$.

Hashing is a method to store and retrieve a set of items in a table. Each item in the set has a key, w, which uniquely specifies the item. Then the location of the item with key w is given by h(w). Here h is called the hash function. But such is not always the case, because usually we will have $h(w_i) = h(w_j)$ for some i <> j. This situation is called (hash) collision and further work has to be done to get the item desired. This process is called collision resolution.

Collision resolution reduces system performance. A function which makes

$$h(w_i) <> h(w_j) \text{ for all } i <> j$$

then h is called a perfect hash function. Use of such hash functions eliminates the need for collision resolution. In this case the location of the item whose key is w is simply given by h(w) and no provision for hash collision resolution is necessary. This may be good enough, but when the size of the table on which the items are to be stored is taken into

account, perfect hash is still not optimal since the table may contain wasted empty locations. The improvement to perfect hashing is minimal perfect hashing. Now, the size of the table is exactly the number of items to be stored into it. The resultant function and table is determined by the keys of the items. So when these methods are applied, retrieval of items from a static table is facilitated. Such hash functions are dependent on their domain and are not easy to find. Several methods have been developed to compute minimal perfect hash functions. But these methods invariably involve so much computation that they are only suitable to be applied to static sets, ie. where the sets of the keys are not to be changed.

A few years ago it was believed that general computation of minimal perfect hash function for a set of keys was difficult. Knuth [2] gave an example that to compute a perfect hash function which maps a set of 31 keys into a set comprised of 41 integers, may take 10 million computations. Since then, several schemes have been found that do this job in much fewer computations. This paper takes one of them and uses Monte Carlo method to provide the keys and tries to figure out how much time it would require to compute a minimal perfect hash function.

II. MINIMAL PERFECT HASH FUNCTION GENERATION ALGORITHMS

1). Sprugnoli's [3]:

Sprugnoli gave two methods, they are called, respectively, quotient reduction method and remainder reduction method. These two methods gives perfect hash functions but not minimal perfect hash functions, actually sometimes the hash function computed results in rather sparse hash tables.

i. Quotient reduction method.

The basic form of this hash function is:

h(w) = the-integer-part-of ((w+s)/n)

Where w is the key and s and n are parameters of this function. s is called the translation term and can be decomposed into s = q*n + s' for some q and s' (0 <= s'< n). The term q*n is used to set $h(w_0)$ to 0 and s' is to adjust the w's to different intervals [kn,(k+1)n] so that $h(w_i)$ <> $h(w_j)$ for every i <> j.

ii. Remainder reduction method:

The form of this hash function is:

h(w) = the-integer-part-of (((d+wq) mod M)/N)

Where d, q, N, M are parameters to be chosen for this function for it to posses the properties we desire.

2). Cichelli's[4]:

Cichelli gives a minimal perfect hash function in the form: hash value <-- key length + the associated value of the key's first character + the associated value of the key's last character. To apply his method, the keys are first sorted into descending order by the sum of the frequencies of the occurrences of each key's first and last letter. This ordering is then modified such that any word (key) whose hash value is already determined by the previous words (keys) is placed next. After ordering the keys, an exhaustive search is used to find the values associated to each letter.

Cichelli asserts that this method is applicable to sets of keys up to four times as large as those said to be feasible by the method described by Sprugnoli. This method does not guarantee success.

3). Jaeschke's[5]:

This minimal perfect hash function is of the form:

$h(w_i)$ = (the-integer-part-of C/w_i) mod n
where n = |W| and W = $\{w_i \mid 1<=i<=n\}$.

When w_i's are not pairwise prime this C may not exit, in which case a transformation Dw+E is employed so that Dw_1+E, Dw_2+E,..., Dw_n+E are pairwise prime. This method works well for n=|W| up to 15. For larger n, grouping is

used to keep the number of keys in each group less than 15 so that when this method is applied to each group computation time will remain minimal.

Jaeschke gave an algorithm to compute the C as following:

1. The w's are sorted into ascending order before computation begins.
2. Compute the smallest common multiple of W: smc. Set L = n * smc.
3. $C_0 = (n-2)w_1 w_n / w_n - w_1$
4. Set $C = C_0$
5. For all i in [1..n] compute C/w_i. If $C/w_i <> C/w_j$ for all j in [1..n] and j <> i then the algorithm terminates successfully.
6. If C > L then the algorithm terminates unsuccessfully.
7. Compute

   j_0 = max{j| Exists i such that $(C/w_i) \bmod n = (C/w_j) \bmod n$}.

   i_0 = max{i| $(C/w_i) \bmod n = (C/w_{j_0}) \bmod n$}

   $\alpha(C,W) = \min\{w_{i_0} - C \bmod w_{i_0}, w_{j_0} - C \bmod w_{j_0}\}$

   $C = C + \alpha(C,W)$

   goto step 5

4). Changs's[6] method:

Chang's hashing function is $h(w_i) = C \bmod p(w_i)$. Where p(w) is a prime number function that transform w into a

prime number and if $w_i <> w_j$ then $p(w_i) <> p(w_j)$. This method is based on the Chinese remainder theorem which states that:

> Let $r_1, r_2, \ldots, r_n$ be integers and $m_1, m_2, \ldots, m_n$ be n pairwise prime integers. Then, there exists an integer C such that $C \equiv r_1 \pmod{m_1}$, $C \equiv r_2 \pmod{m_2}, \ldots, C \equiv r_n \pmod{m_n}$.

Now that we let $r_i = i$ and $m_i = p(w_i)$, then $h(w_i) = C \bmod p(w_i)$ is clearly a minimal perfect hash function. With this, Chang proved the following theorem:

> Let m_i and m_j be relatively prime where $i \neq j$ and $1 <= i, j <= n$. Let $m_1 < m_2 < \ldots < m_n$. $\Sigma_{1<=i<=n} b_i M_i i \bmod m_j = j$ if $M_i = \Pi_{j \neq i} m_j$ and $M_i b_i \equiv 1 \pmod{m_i}$.

Then b_i's are calculate using the famous Euclidean algorithm[7]. For a set of p(w)'s, there are infinite number of C's that satisfy this Chinese remainder theorem, we therefore would like C to be the smallest among them. Since we have $C=\Sigma_{1<=i<=n} b_i M_i i$ which satisfies $C \equiv i \pmod{m_i}$. Let $C' \neq C \pmod{m_i} \equiv i \pmod{m_i}$. Then $C-C' \equiv 0 \pmod{m_i}$ for all i. Then C-C' is a multiple of $\Pi_{1<=i<=n} m_i$, which implies there is at most one solution between 0 and $\Pi_{1<=i<=n} m_i$. Thus $C = \Sigma_{1<=i<=n} b_i M_i i \bmod \Pi_{1<=i<=n} m_i$ is smallest positive integer such that $C \equiv i \pmod{m_i}$. After w's are transformed into prime numbers , m's, the following algorithm is used to find C.

1. [Compute All M_j's]
   Compute $M_j = \Pi_{i \neq j} m_i$ for all $1 <= j <= n$.

2. [Compute All b_i's]
   For all i, $1 <= i <= n$ compute
   $M'_i = M_i \bmod m_i$
   Dend = m_i
   Dsr = M'_i
   j = 1
   Q_j = Dend /Dsr
   Rmd = Dend - Qj * Dsr
   while Rmd ≠ 1 do
      Dend = Dsr
      Dsr= Rmd
      j = j+1
      Q_j= Dend / Dsr
      Rmd = Dend - Q_j * Dsr
   end while
   k = j
   $B_0 = 1$
   $B_1 = -Q_k$
   do j = 1 to k-1
      $B_{j+1} = -B_j * Q_{k-1} + B_{j-1}$
   end do
   $b_i = B_k$

3. [Compute C]
   Compute $C = \Sigma_{1<=i<=n} b_i * M_i * i \bmod \Pi_{1<=j<=n} m_j$

Chang's method is unique in that:

1). It is conceptually very simple.

2). It guarantees a minimal perfect hash function.

3). It does not require back tracking.

But in using this method users are confronted with two problems:

1). Each item must have a unique numerical key.

2). The C's produced by this method are very large[8]. Actually, their magnitude increase exponentially.

5). Sager's Method[8]:

In this method words are transformed into edges in a graph and the overall topology of the graph represents the interdependence of these words. First the algorithm accepts as input set of words, ie. character strings , W = { w_i | 1<=i<=n }. Then each word, w_i, is hashed into 3 independent positive integers: $h_0(w_i)$, $h_1(w_i)$, $h_2(W_i)$. Now, $h_1(w_i)$'s and $h_2(w_2)$'s are the vertices and each word in W defines an edge of the graph G = <V,E>. Where V and E are set of vertices and edges of the graph respectively. Further, to make sure that each word defines an edge, $\{h_1(w) | w \in W\}$ and $\{h_2(w) | w \in W\}$ are disjoint. The algorithm tries to associate a number with each vertices in V so to make

$$H(w) = (h_0(w)+g \circ h_1(w)+g \circ h_2(w)) \bmod n$$

the desired function where the function g associates a number to each vertex.

The method used to find this g is by an exhaustive search which potentially makes this algorithm intractable. But because the words are ingeniously ordered before this search begins, usually only a small

portion of the whole space is searched before the algorithm reaches a solution. Actually, experimental result shows, when V is sufficiently large, there is virtually no backtracking, that is the search succeeds on practically the first hit. The overall algorithm is composed of 3 parts, part 1 hashes the words into positive integers, part 2 orders the words, then in part 3 this sequence of words determines which number in [0...|W|-1] will be assigned to each vertex.

To understand how this ordering is done, consider the sequence of graphs: G_0, G_1,....,G_k where for all $i \in [0..k]$:

$G_i = \langle V_i, E_i \rangle$,

V_i = The partiton of V generated by the smallest equivalence relation containing $\{\langle h_1(w), h_2(w)\rangle | w \in W_i\}$,

E_i = the multiset of edges over V_i whose characteristic function is $\phi(\{p,q\}) = \text{card}(\{w \in W-W_i | \{h_1(w), h_2(w)\} \subset p \cup q\})$.

$W_0 = \phi$

W_i and G_i are computed from G_{i-1} as follows:

Choose p and q such that {p, q} is an edge of the graph G_{i-1} lying on a maximal number of minimal length cycles over G_{i-1}. Then let $X_i = \{w \in W-W_{i-1} | \{h_1(w), h_2(w)\} \subset p \cup q\}$ and let $W_i = W_{i-1} \cup X_i$.

The algorithm employed to accomplish this, which Sager calls the mincycle algorithm, is adapted from the well known $O(n^3)$ Warshall's algorithm[9] For each G_i, using this algorithm as skeleton, mincycle algorithm takes notes of cycles' mid-points so that these cycles can be recovered later. Since $|\{G_i\}|$ is proportional to N, this algorithm is $O(n^4)$. Actually, this algorithm only has to be applied to cycles of lengths greater than 2. Smaller cycles, those of length 2 and edges which do not belong to cycles, can be detected from a graph with algorithms of $O(n^2)$.

For part 3, for all $i \in [1...k]$, let $X_i = W_i - W_{i-1}$ and choose arbitrarily a canonical member x_i of X_i and let $Y_i = \{x_j | j \in [1...i]\}$. Now for all $i \in [0...k]$ and for all $w \in W_i$, let $path(w) = y_0, y_1, \ldots, y_t$ be the unique sequence of edges over Y_i such that the sequence of edges

$$\{h_1(y_0), h_2(y_0\}, \{h_1(y_1), h_2(w_1\}, \ldots, \{h_1(y_t), h_2(y_t)\}$$

form a path from $h_1(w)$ to $h_2(w)$ over the graph G. Then given $H(w) = (h_0(w) + (\Sigma_{0<=j<=t} (-1)^j U(y_j)))$ mod N an injection from W_{i-1} into [0...N-1], to extend it to the domain W_i, search for an $n \in [0...N-1]$ that makes H(w) an injection from W_i into [0...N-1]. Where $path(w) = y_0, y_1, \ldots, y_t$ and $U: Y_i \rightarrow [0...N-1]$ is

$$U(x_j) = \begin{cases} (H(x_j) - h_0(x_j)) \bmod n & \text{if } 0<j<i \\ n & \text{if } j = i. \end{cases}$$

The detail of behavior for this method is discussed in the next chapter.

III. OBSERVATION ON THE PERFORMANCE OF SAGER'S METHOD

To see how this method performs, for each chosen |W| and number of vertices |V|, random numbers of uniform distribution are used as $h_0(w)$'s, $h_1(w)$'s and $h_2(w)$'s, and minimal perfect hash function is generated by an implementation of this method which was was coded by Mr. John Pulley in the summer of 1984. This implementation is coded in Turbo Pascal and is to be run on an IBM PC. Time used to order W (Part 2) and to exhaustively search the function (Part 3) are recorded separately. |W| ranges from 100 to 1200 by 100 and for each |W|, the number of vertices |V| ranges from 75% to 150% of |W|. For smaller |W|, behavior of this algorithm when the number of vertices is small is also investigated. The results are depicted in Fig.1 through Fig.13. The behavior of this method is greatly influenced by the ratio between number of vertices (|V|) and number-of-words (|W|).

i). High |V|/|W|:

The experimental results show that for number of vertices, |V|, of about 75% of |W| and above, little backtracking is encountered in computing the minimal perfect hash function in part 3. In these cases time consumed by this part of the program is insignificant compared to that comsumed by the ordering part. The time consumed by ordering decreases as |V| increases, due to increase of non-cycle edges and long cycles. The appearance of large cycles means that with each application of the mincycle algorithm more vertices

are removed and the mincycle algorithm does not have to be applied as many times. But the main factor that makes ordering time decrease is the increase of non-cycle edges.

When |V| is large, large proportion of all edges, normally more than 30%, are non-cycle edges. Non-cycle edges also reduce the problem of computing minimal perfect hash function to that of computing perfect hash function since vertices that only appear in these edges can be assigned to any number as its U value in part 3.

As the number of vertices decrease, the ordering time increases. But the observed complexity is still less than $O(n^4)$. This is because as non-cycle edges decrease, length 2 cycles increase.

In all, with this |V|/|W|, this method performs marvelously. While other methods consider |W| of 15 as large, this method computes minimal perfect hash function for |W| as large as 1200 in 5 minutes on an IBM PC (|V|/|W| = 1.33).

ii). Low |V|/|W|:

As |V|/|W| decreases times spent in both the ordering part and the search part of the program increase. The increase in the ordering part follows the same pattern as for high |V|/|W|, but time used in the search part increases exponentially and thus tends to cause the method to fail. We tried to find some boundary value of |V|/|W| below which this method is unlikely to succeed, but due to the large variance

involved this value is not easily found. For example, with |V| = 195 and |W| = 300, 62% of tests spent more than 1 hour in part 3, at which point we aborted the test, while with |V| = 194 and same |W| = 300, 80% of the tests finished in 1 hour. So the conclusion is that for low |V|/|W| this method tends to be unreliable. This situation is depicted in table 1 through 4.

IV. CONCLUSION

Sager's minimal perfect hash function is very effective and easy to use. Its calculation is very fast with reasonable |V|/|W|. With small |V|/|W|, it tends to be unreliable but still has some chance to succeed.

We had hoped to find experimentally a function f so that if |V| > f(|W|) then the mincycle algorithm could be expected to succeed and if |V| <= f(|W|) then the mincycle algorithm would behave erratically. The experimental data does not point clearly to any such function. However, by examining the experimental data we can give the very crude guess:

$$|V_{min}| = 0.766 * |W| - 24$$

More data would be necessary to test this hypothesis, however that would be beyond the scope of this thesis. Although this guess turns out to be a linear function, it is not at all clear that a more refined measure of this function would be linear.

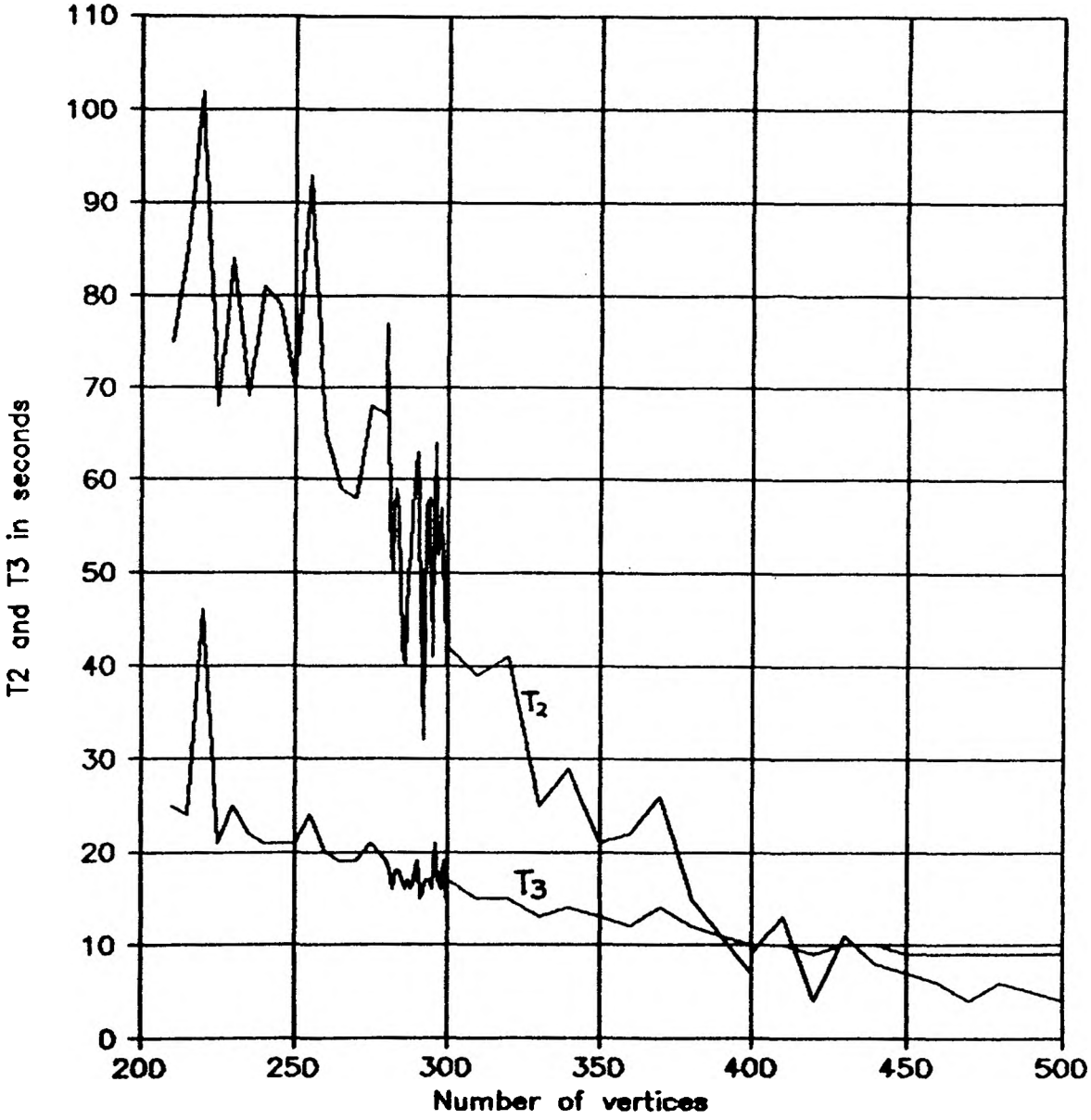

Fig.1
Time Used by Part 2 and Part 3 of Mincycle Program, |W| = 300 , large |V|/|W|

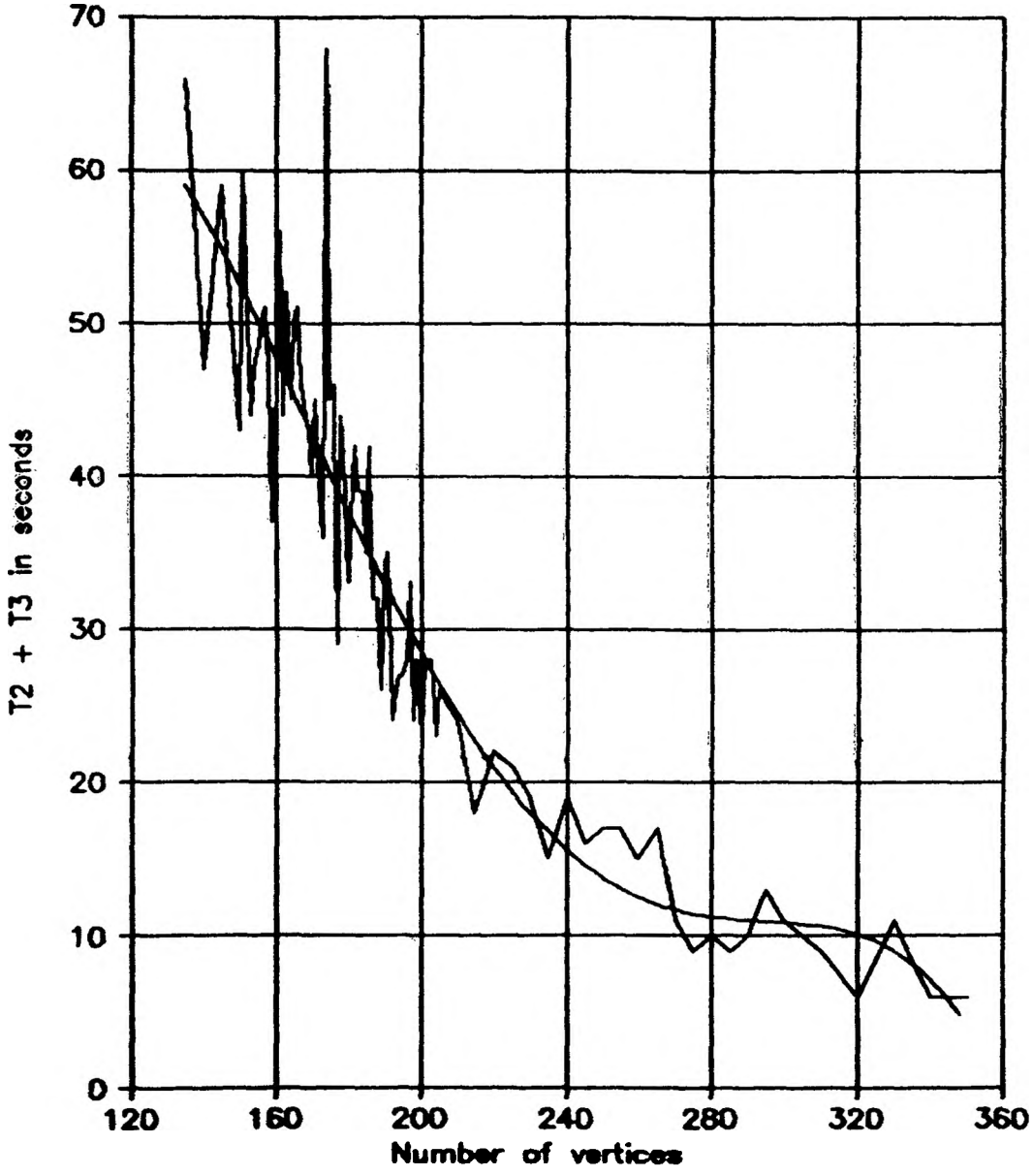

Fig.2
Time Used by Mincycle Program, |W| = 200, large |V|/|W|

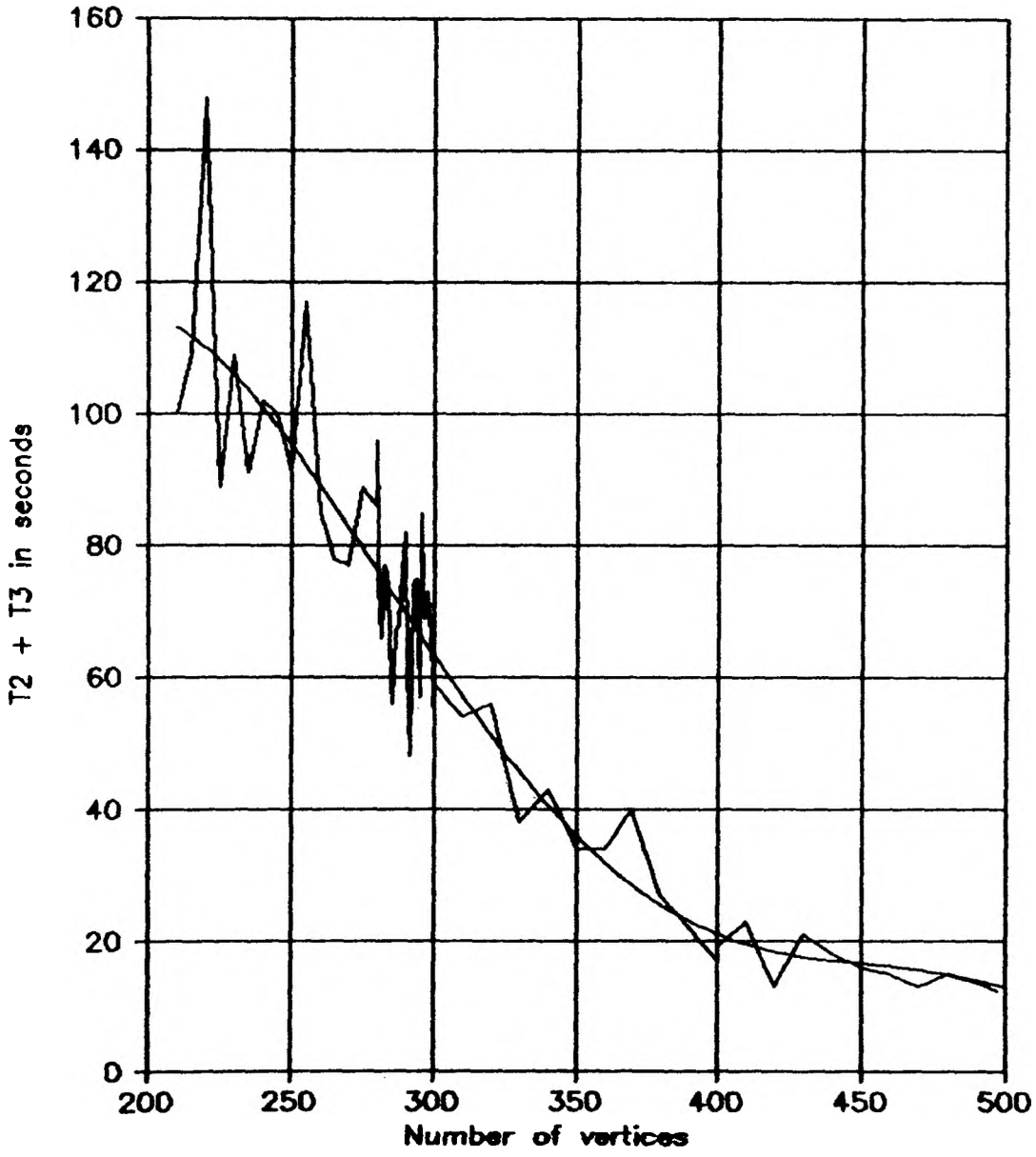

Fig.3
Time Used by Mincycle Program, |W| = 300, large |V|/|W|

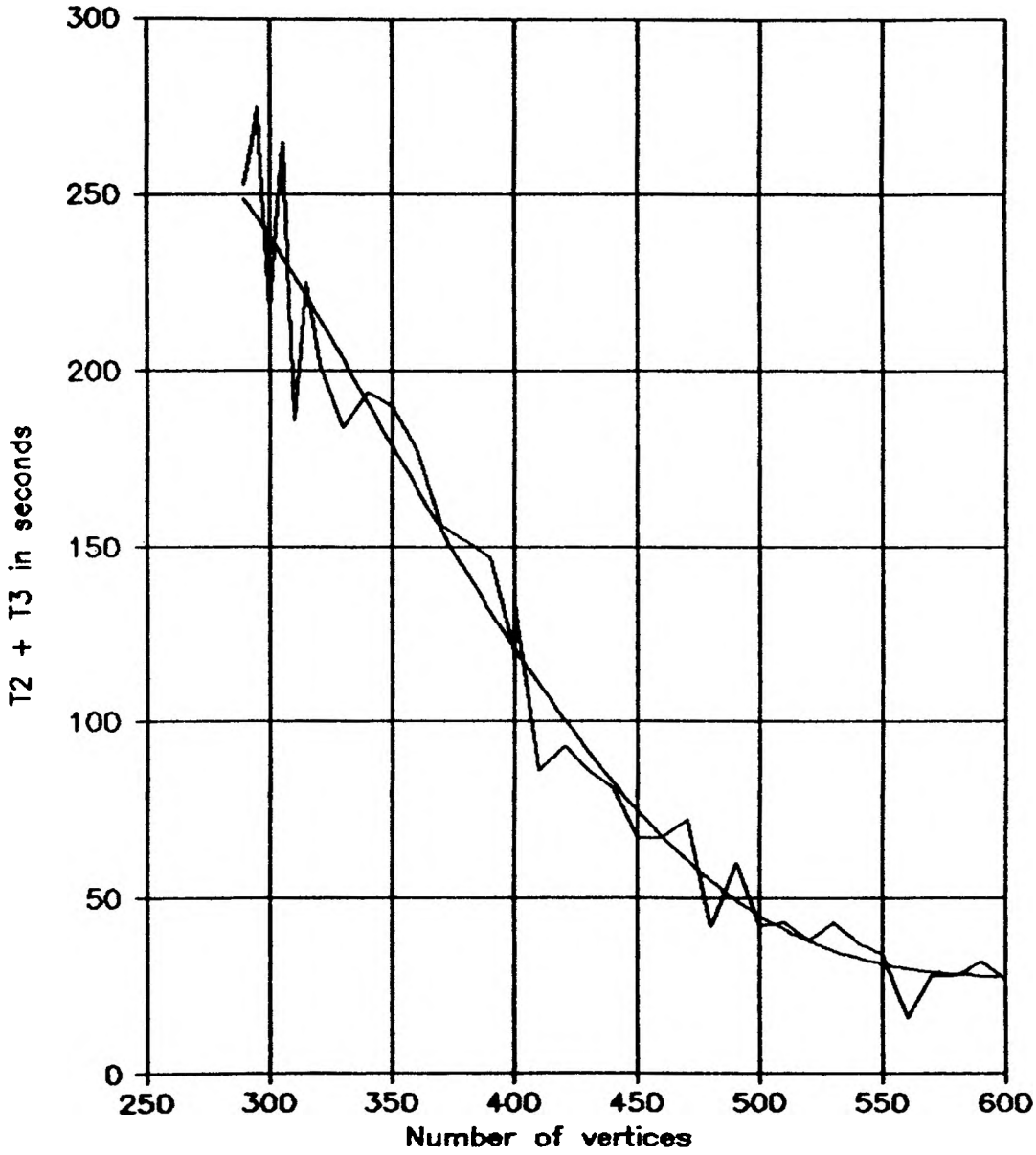

Fig.4
Time Used by Mincycle Program, |W| = 400, large |V|/|W|

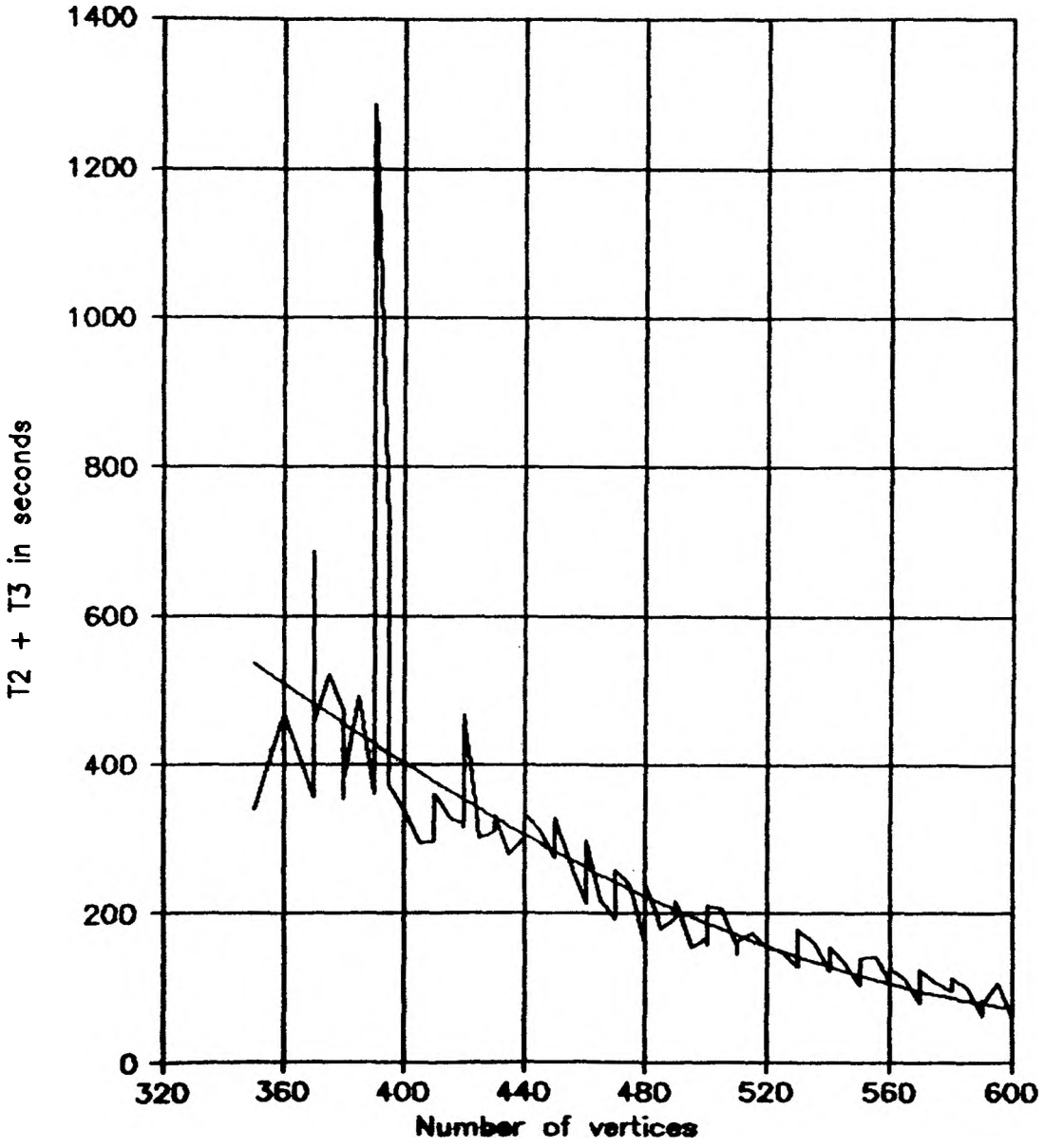

Fig.5
Time Used by Mincycle Program, |W| = 500, large |V|/|W|

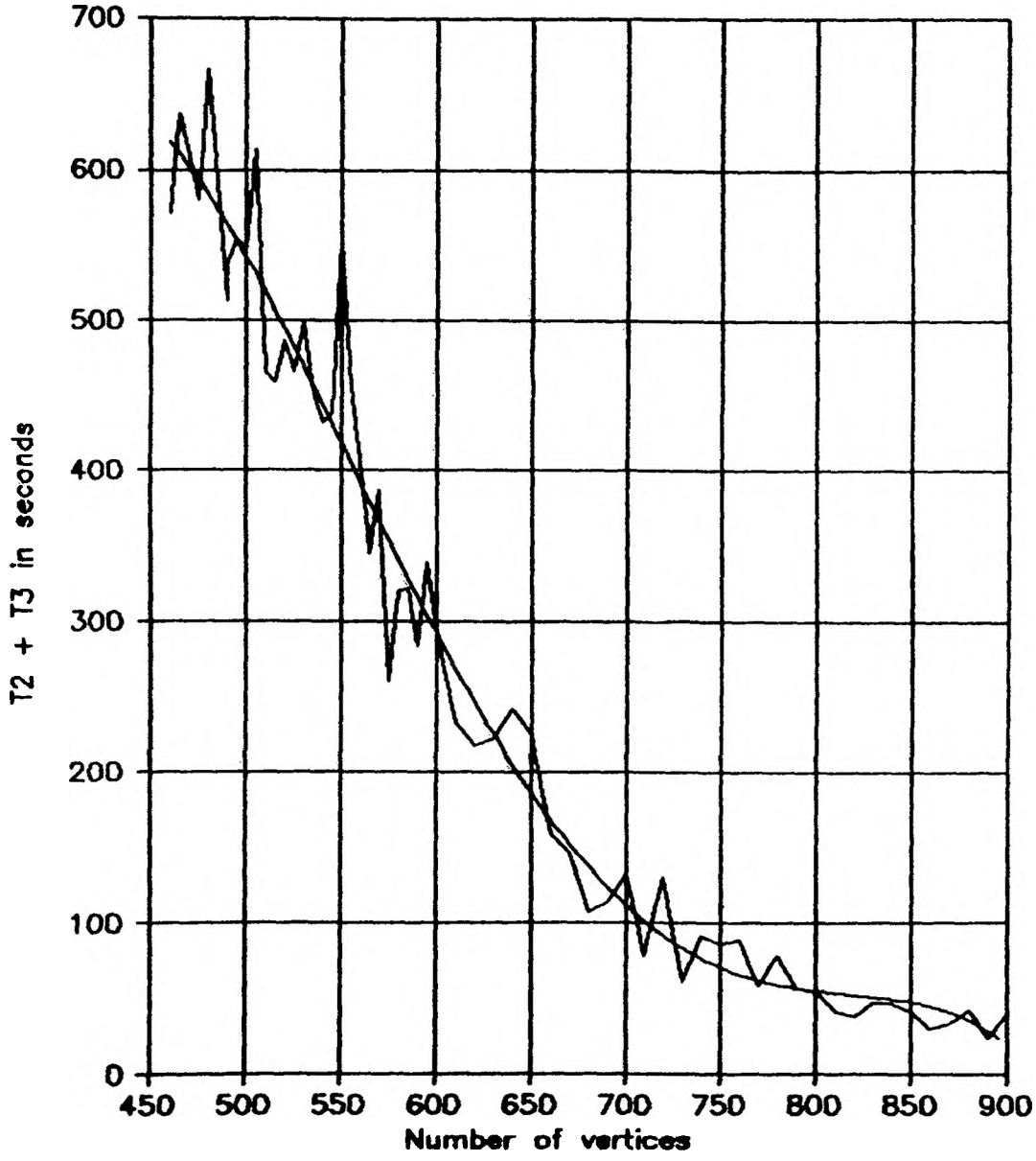

Fig.6
Time Used by Mincycle Program, |W| = 600, large |V|/|W|

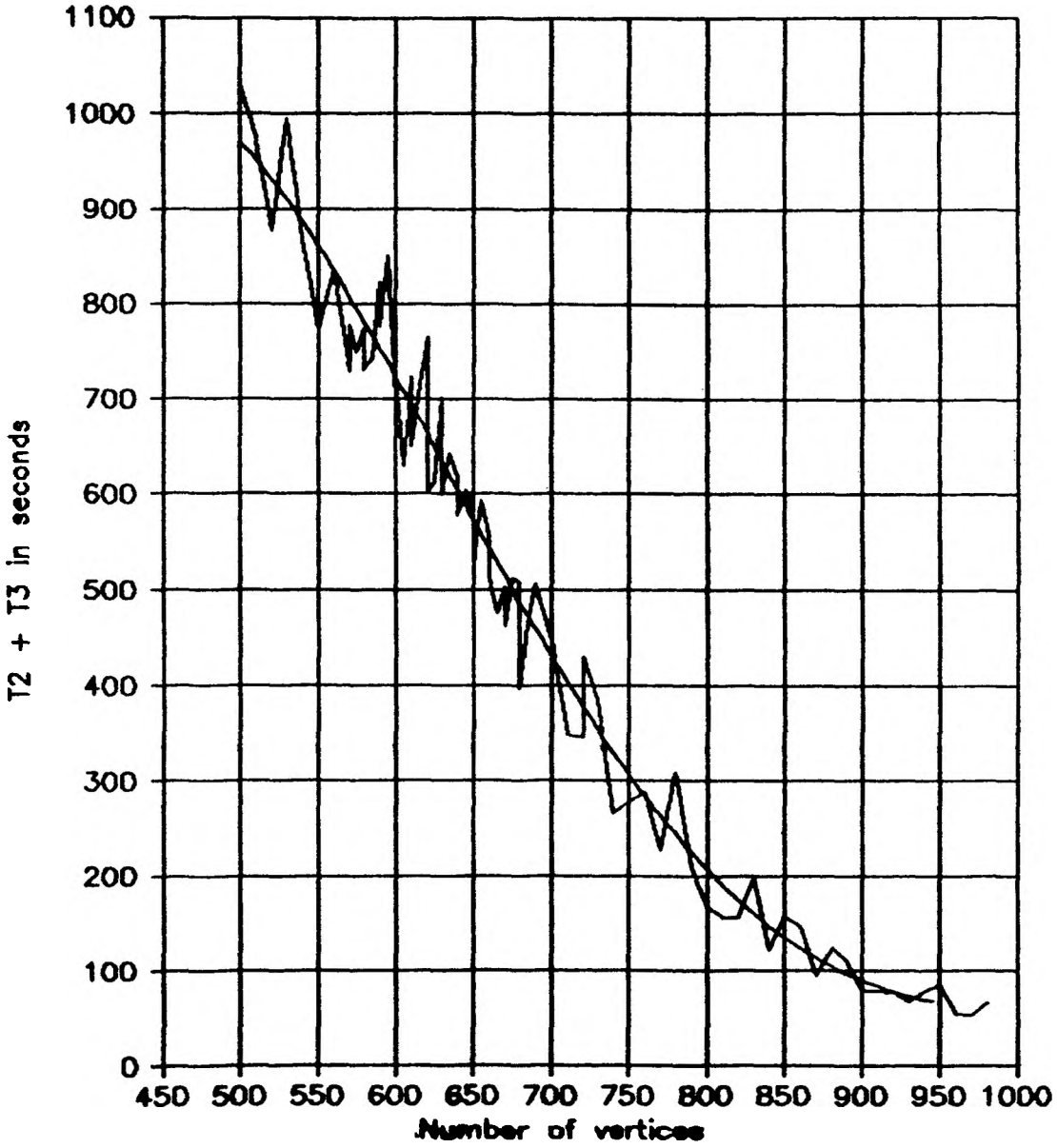

Fig.7
Time Used by Mincycle Program, |W| = 700, large |V|/|W|

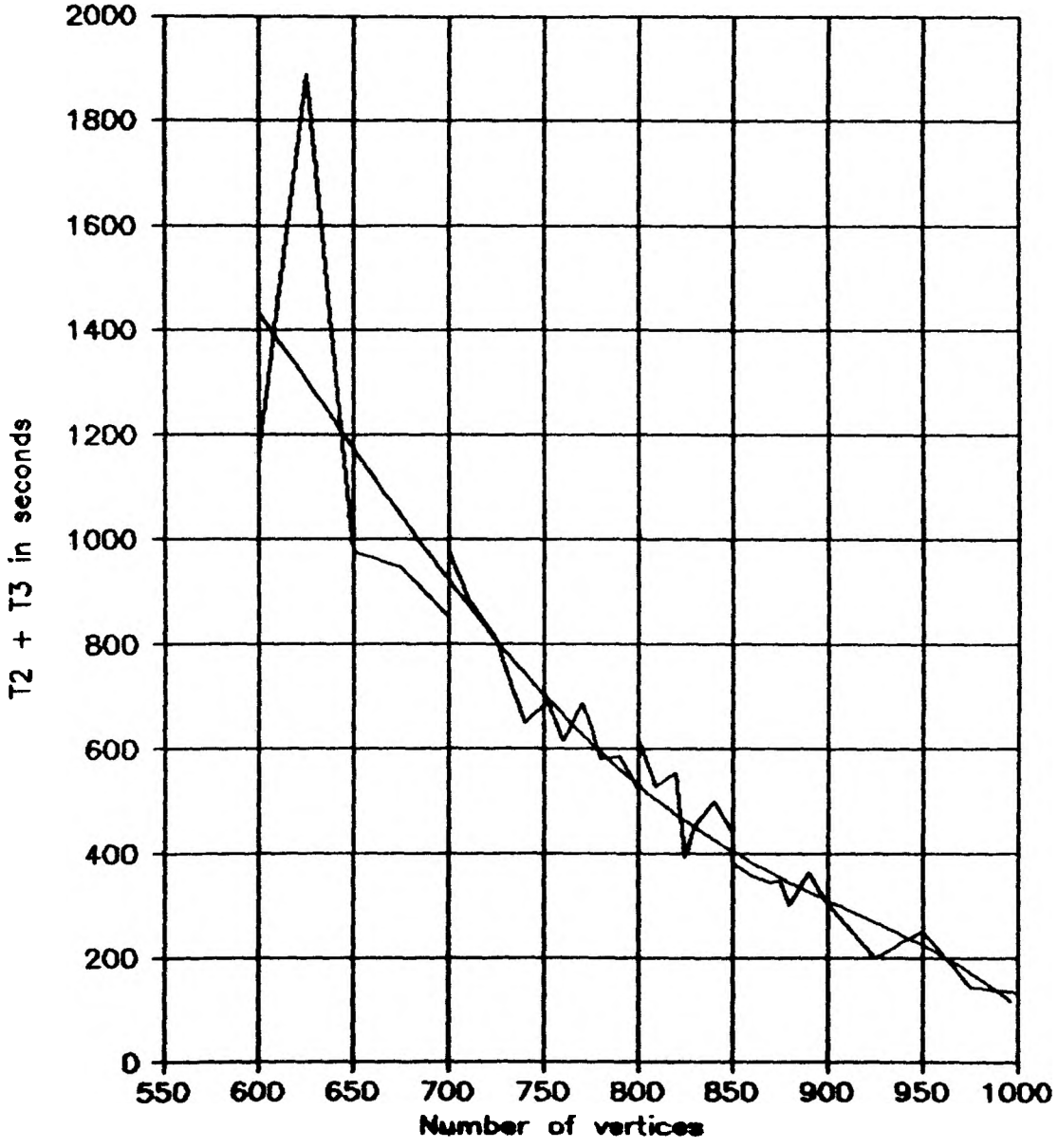

Fig.8
Time Used by Mincycle Program, |W| = 800, large |V|/|W|

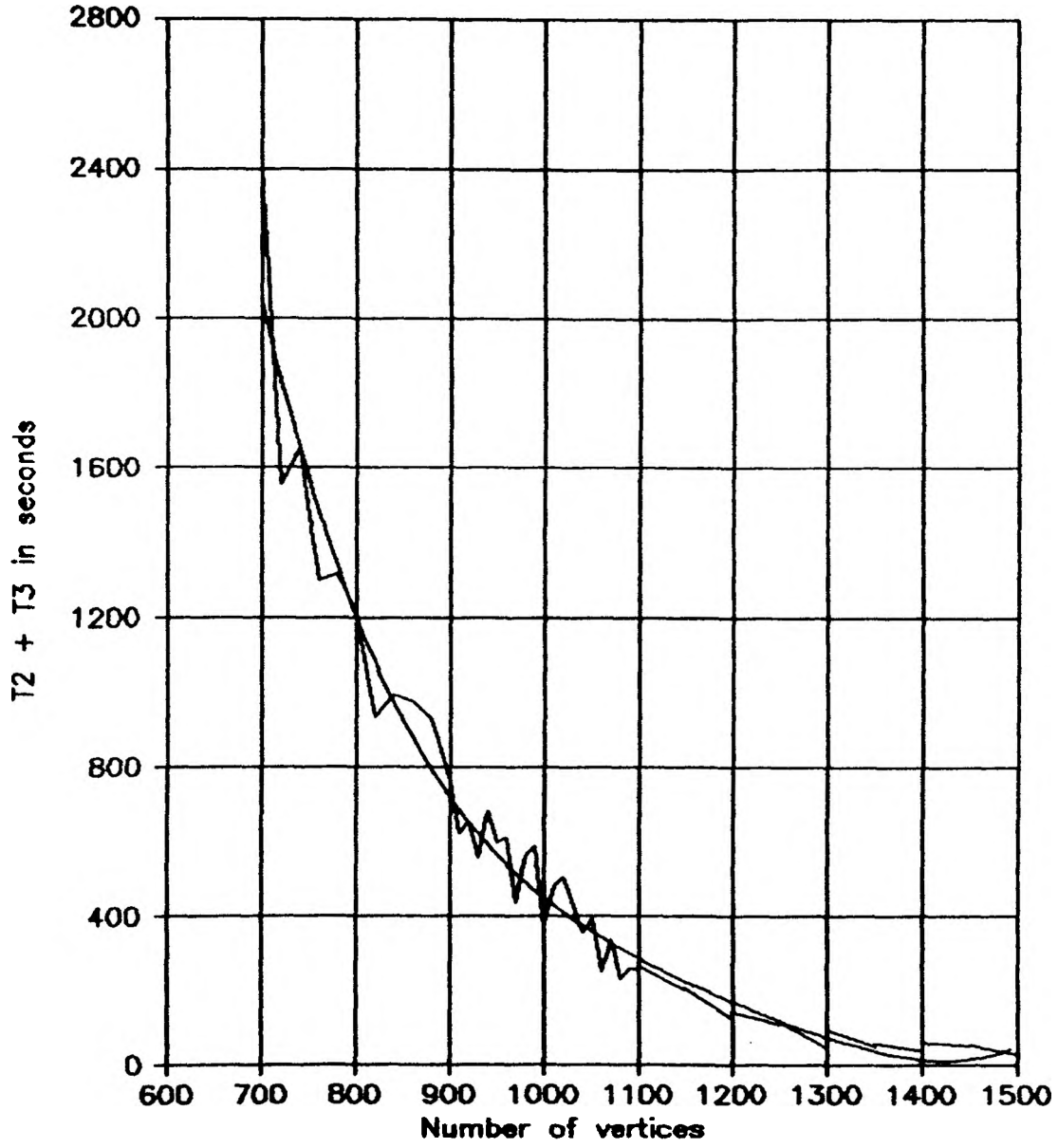

Fig.9
Time Used by Mincycle Program, |W| = 900, large |V|/|W|

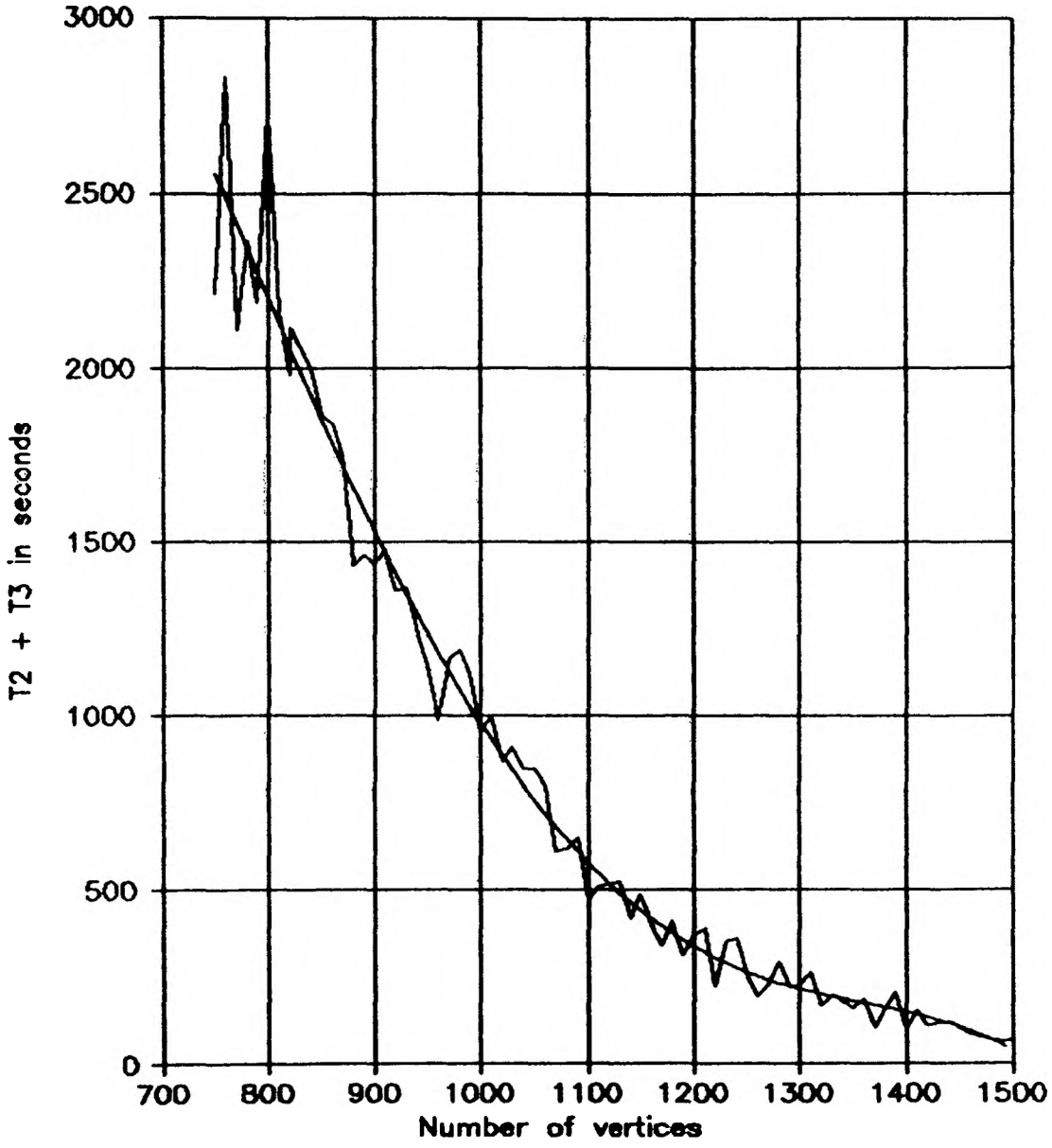

Fig.10
Time Used by Mincycle Program, |W| = 1000, large |V|/|W|

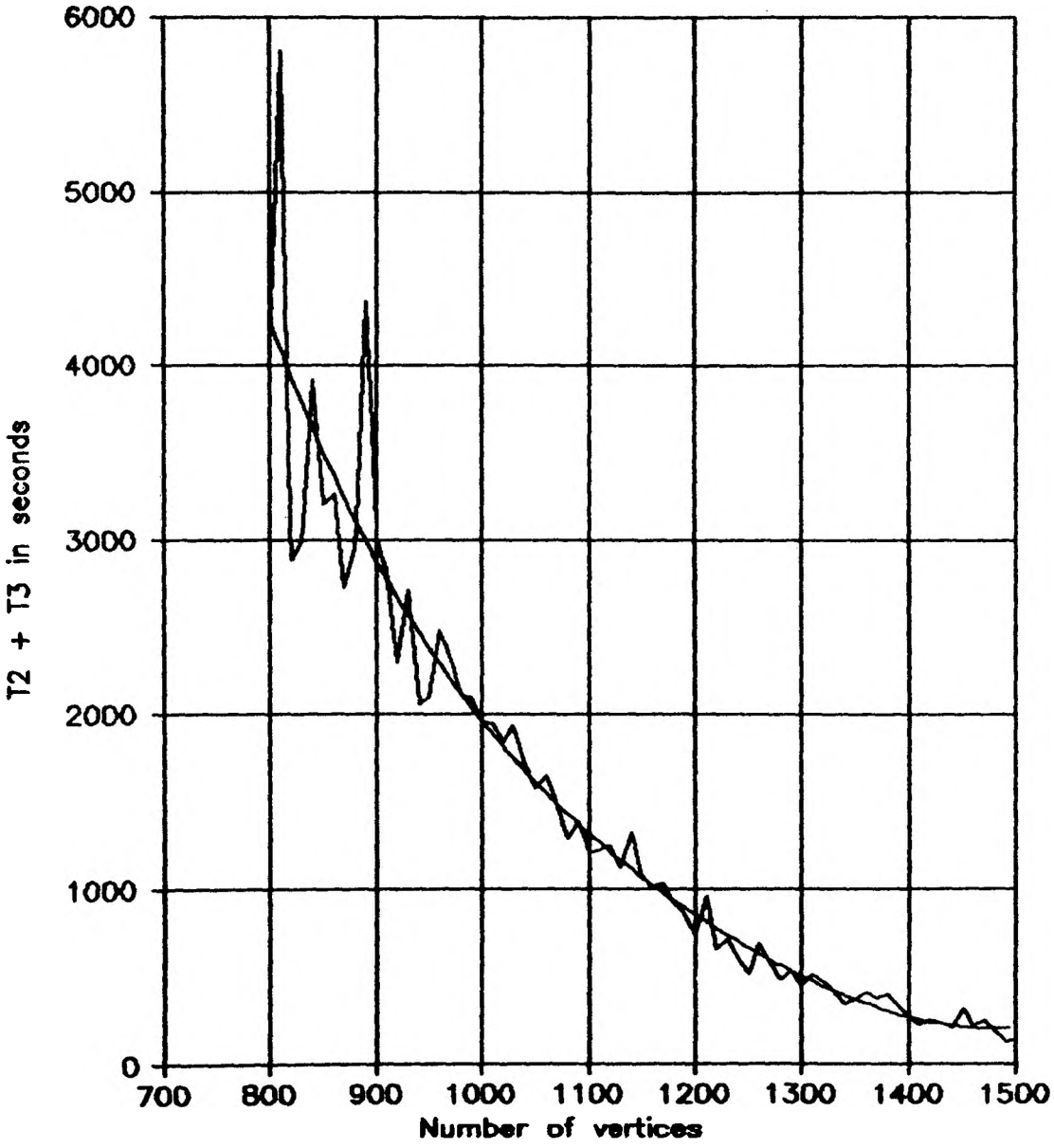

Fig.11
Time Used by Mincycle Program, |W| = 1100, large |V|/|W|

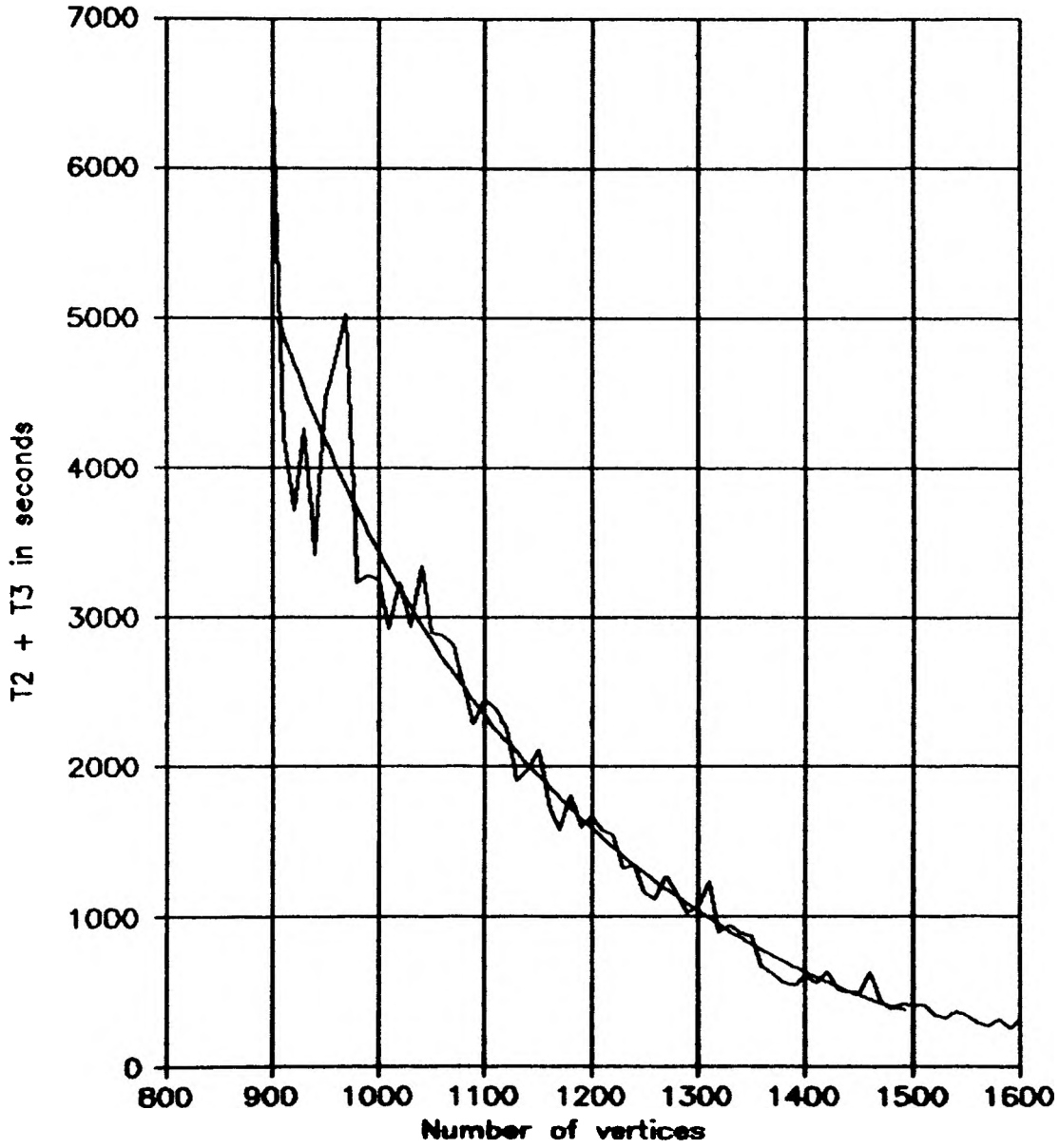

Fig.12
Time Used by Mincycle Program, |W| = 1200, large |V|/|W|

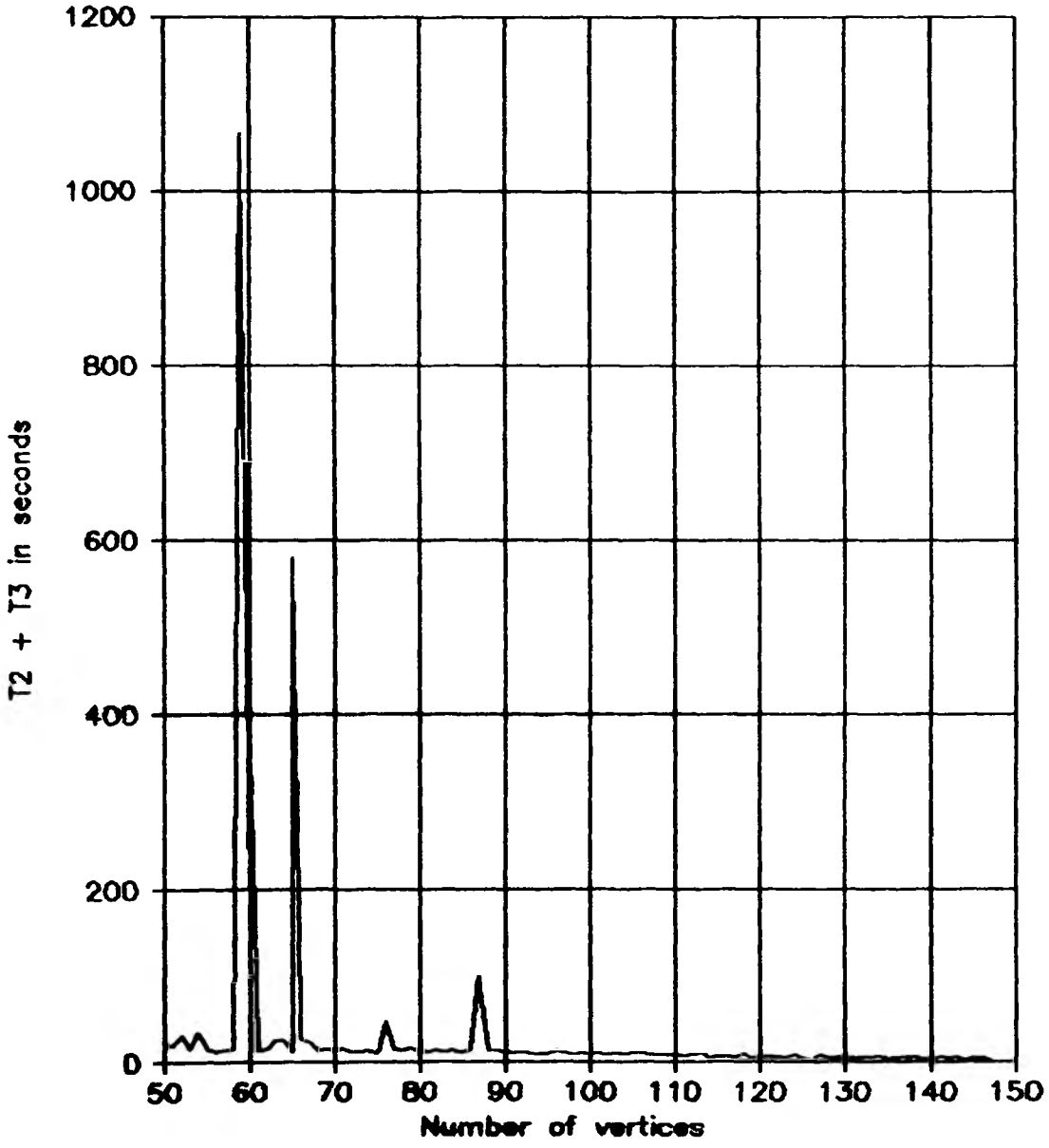

Fig.13
Time Used by Mincycle Program, |W| = 100, small |V|/|W|

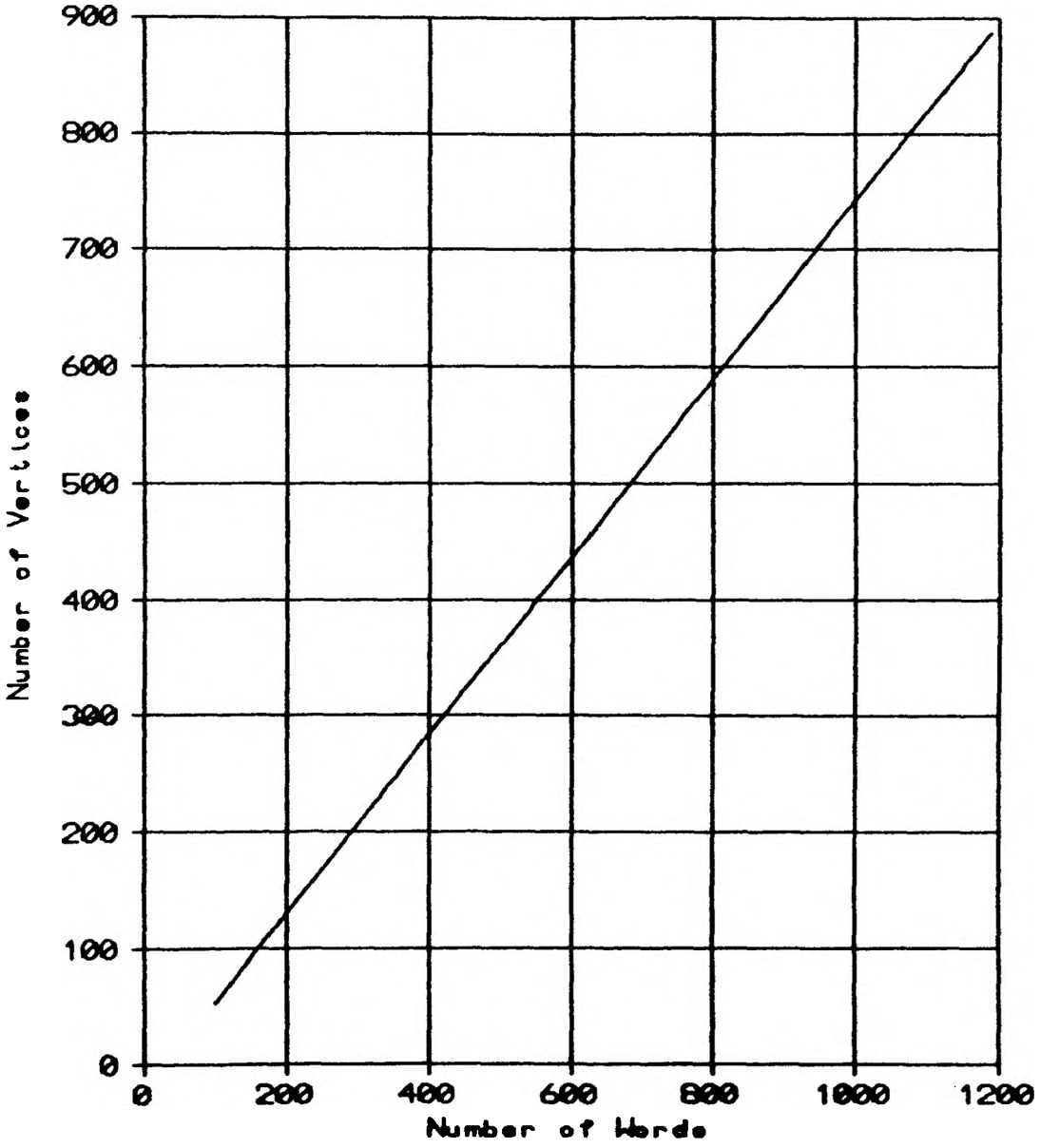

Fig.14
Recommended minimal number of vertices, |V|, for each number of words, |W|

Table I
Performance Tests on Mincycle Method,
$|W| = 100$, small $|V|/|W|$

Number of vertices	Number of tests	Number of completed tests	% of test that completed
44	3	0	0
45	3	1	33
46	3	1	33
47	4	3	75
48	7	2	28
49	19	13	68
50	22	14	64
51	25	22	88
52	23	20	87
53	20	20	100
54	16	15	94
55	15	15	100
56	17	17	100
57	7	7	100
58	7	6	86
59	7	7	100
60	7	7	100
61	6	6	100
62	1	1	100
63	1	1	100

Table II
Performance Tests on Mincycle Method,
$|W| = 200$, small $|V|/|W|$

Number of Vertices	Number of tests	Number of completed tests	% of completed tests
117	11	6	54
118	22	14	63
119	27	20	74
120	28	15	53
121	20	16	80
122	22	18	82
123	15	13	87
124	15	9	60
125	13	8	62
126	17	10	59
127	12	10	83
128	10	9	90
129	7	7	100
130	10	9	90
131	10	6	60
132	10	4	40
133	12	9	75

Table III
Performance Tests on Mincycle Method,
|W| = 300, small |V|/|W|

Number of vertices	Number of tests	Number of completed tests	% of completed tests
191	14	6	43
192	20	15	75
193	30	17	57
194	44	35	80
195	42	26	62
196	35	24	68
197	12	7	58
198	15	10	67
199	10	7	70
200	10	9	90
201	5	4	80
202	5	3	60
203	5	4	80
204	5	4	80
205	5	4	80
206	10	9	90
207	5	4	80
208	5	5	100
209	5	4	80

Table IV
Performance Tests on Mincycle Method,
$|W| = 400$, small $|V|/|W|$

Number of vertices	Number of tests	Number of completed tests	% of completed tests
266	10	7	70
267	15	8	53
268	41	23	56
269	45	32	71
270	24	15	62
271	25	18	72
272	15	12	80
273	15	12	80
274	10	9	90
275	10	9	90
276	10	7	70
277	10	8	80
278	10	9	90
279	15	12	80
280	10	9	90
281	10	9	90
282	5	5	100

BIBLIOGRAPHY

1. Biles, W. E. and Swain, J. J. "Optimization and Experimentation," Addison-Wesley Publishing Company, Reading, Mass (1980)
2. Knuth, D. E. "The Art of Computer programming," Vol III: Sorting and Searching, Addison-Wesley Publishing Company, Reading, Mass (1978), pp.506
3. Sprugnoli, R "Perfect Hashing Functions: A Single Probe Retrieval Method for Static Sets," Comm. ACM, 20, 11, (Nov, 1977) 841-850
4. Cichelli, R. J. "Minimal perfect hash functions made simple," Comm. Acm, 23, 1 (Jan. 1980), 17-19
5. Jaeschke G. "Reciprocal hashing: A method for Generating Minimal Perfect Hashing Functions," Comm. ACM, 24, 12,(Dec.1981) 829-833
6. Chang, C.C. "The Study of an Ordered Minimal Perfect Hashing Scheme," Comm. ACM, 27, 4,(April 1984) 384-387
7. Knuth, D. E. "The Art of Computer programming," Vol I: Fundamental Algorithms, Addison-Wesley Company, Reading, Mass (1978), pp.14
8. Sager. T. J. "A Polynomial Time Generator for Minimal Perfect Hash functions," Comm. Acm 28, 5(May 1985), 523-532
9. Aho, A .V., Hopcroft J. E. Ullman J.E. "The design and Analysis of Computer Algorithms," Addison-Wesley Publishing Company, Reading, Mass (1974)