NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

A BANDWIDTH REDUCTION ALGORITHM FOR TREES

Chandra GowriSankaran

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

August 1988

ISBN 0-315-51352-7

ABSTRACT

A Bandwidth Reduction Algorithm for Trees

Chandra GowriSankaran

The most successful bandwidth reduction algorithms for graphs are level structure algorithms. However, the algorithms currently in use are not very successful in reducing bandwidths of trees since they do not exploit the special characteristics of trees. This thesis studies a new bandwidth reduction algorithm for trees, algorithm LST, proposed by J. Opatrny and Z. Miller, which defines recursively a level structure for trees. Empirical studies were conducted to evaluate the performance (i.e. ability to reduce the width of level structure generated) of this algorithm, in particular in comparison with the most successful bandwidth reduction algorithm to date, the GPS algorithm. It is shown that in almost all examples of trees studied, algorithm LST produced level structures of smaller width than did GPS algorithm. In addition, it is shown that for a tree on n vertices with maximum degree d, algorithm LST will produce a level structure in time $O(d*n^{3/2})$. Considerations necessary in the implementation of the algorithm and alternative versions of the original algorithm are discussed.

ACKNOWLEDGEMENTS

I would like to thank Professor Jaroslav Opatrny for suggesting the topic of this thesis, for being always available for discussions, for his kind and valuable help in preparing this thesis, for his understanding and his toleration of my many failures to meet self-imposed deadlines.

I would also like to thank my family for their support, patience, encouragement, understanding and helpful participation in all my endeavours.

TABLE OF CONTENTS

INTRODUCTION

Informally, the bandwidth problem for graphs is to find a labelling of vertices in the graph such that all the non-zero entries in the adjacency matrix fall within a band of minimum width, the bandwidth, around the main diagonal. Finding the bandwidth of sparse matrices of large size makes possible efficient storage and fast operations on these matrices. Graph bandwidths have also proved to be significant in many fields of applications such as error correcting codes and electrical networks.

The problem of deciding, given any graph G and any integer k, whether the bandwidth of G is less than or equal to k has been proved to be NP-complete [19]. Therefore there has been an ongoing interest in proving theoretical results about special cases of the problem as well as finding time-efficient bandwidth reduction algorithms. The class of level structure algorithms has been proved to be capable of giving near optimum bandwidth in almost all cases. One of the level structure bandwidth reduction algorithms, the GPS algorithm, though very successful in some types of graphs, is not suited for trees. A new bandwidth reduction algorithm for trees, algorithm LST, was proposed by Opatrny and Miller [18]. For trees, algorithm LST generates recursively level structures of much smaller widths than those generated by algorithm GPS.

In Chapter 1, we present the definition and a brief history of the bandwidth problem. Chapter 2 describes two existing level structure bandwidth reduction algorithms, algorithm CUM and algorithm GPS. Known probabilistic results on the capabilities of the level structure algorithms are included.

Chapter 3 describes the new algorithm LST, its application demonstrated with an example. The time complexity of the algorithm is studied. An example where LST produces arbitrarily poor approximation to bandwidth and an example where LST produces near optimal or optimal bandwidth and far outperforms GPS are given. With a view to study empirically the level structure obtained, the author has implemented algorithm LST as a Pascal program and applied it to 2 groups of problems. Chapter 4 lists implementation considerations and some details of implementation. Chapter 5 presents the results and a comparative analysis of the results obtained by applying two versions of the algorithm and the GPS algorithm to 31 test problems.

In conclusion, we find that in almost all cases of trees, algorithm LST produces level structures of much smaller widths than those given by algorithm GPS. Therefore, combined with a labelling algorithm, LST should prove to be a better bandwidth reduction algorithm for trees than GPS.

The contribution of this thesis is a detailed implementation of the algorithm LST, proposal of alternative versions of some parts of the algorithm, the comparative study of different versions of the algorithm, including the empirical study of the test problems, and the comparative study of the algorithm and an existing algorithm, algorithm GPS, including again the empirical results with the test problems. In addition, a detailed analysis of the time complexity of LST is presented including some minor results on diameters in trees.

CHAPTER 1

THE BANDWIDTH PROBLEM

1.1. THE BANDWIDTH OF A GRAPH

THE MATRIX BANDWIDTH: The bandwidth problem can be stated in terms of matrices or graphs. The **matrix bandwidth problem** seems to have been originated in the 1950's when structural engineers first analysed steel frameworks by computer manipulation of the matrices representing their structure. This analysis involves the solution of large systems of linear algebraic equations with sparse, symmetric coefficient matrices. Fluid dynamics and network analysis have also been among the many engineering fields included in the application of sparse, symmetric matrices. In these cases, methods based on Gaussian elimination are not too efficient due to the presence of a large number of zero elements in the matrices. In order to conserve computer storage as well as computation time to solve such systems efficiently, it was found necessary to devise a storage saving scheme, which will allow fast operations on sparse matrices.

One way of achieving this objective is to find, for a given nXn matrix, an equivalent matrix in which all the nonzero entries lay within a narrow band of width W around

the main diagonal. Hence the name bandwidth problem. Clearly, such a matrix will reduce the storage requirement to an nXW array down from the original nXn array; because, now the matrix can be stored as a rectangular array of n rows and W colums, storing only the W entries in each row which lie within the band and ignoring all the zero entries outside the band. With only minor modifications, algorithms of the type of Gaussian elimination could be easily adapted to this data structure, thus minimizing time and storage ir application. This rationale led to the formulation of the bandwidth problem for matrices.

> Thus, to find the **bandwidth of a real symmetric matrix** $M = (m_{ij})$ is to find a symmetric permutation $M' = (m'_{ij})$ of rows and columns of M so that the maximum value of
>
> $$| i-j |$$
>
> taken over all nonzero m'_{ij} is a minimum.

THE GRAPH BANDWIDTH: The **bandwidth problem for graphs,** meanwhile, originated independently at the Jet Propulsion Laboratory at Pasadena in 1962: single errors in a 6-bit picture code were represented by edge-differences in a hypercube whose vertices were words of the code. At JPL, L. H. Harper and A. W. Hales sought codes which minimized the maximum absolute error and the average absolute error. Thus were born the bandwidth and bandwidth sum problems for a

particular class of graphs, the cubes. Not long after this, R. R. Korfhage began work on the bandwidth problem for general graphs (defined below) and F. Harary [14] publicized the problem at a conference in Prague. Since many problems of practical interest are represented using graphs, the bandwidth problem for graphs became significant to many fields of applications.

DEFINITIONS AND PROPERTIES

We refer to [1] for basic terminology in graph theory.

Let $G(V,E)$ be a graph with vertex set V and edge set E. Assume $|V| = n$. A one-to-one mapping f from V to the set $\{1,2,\ldots,n\}$ is called a **labelling** of G. The **bandwidth of a labelling f** of G, denoted $B_f(G)$, is defined as

$$B_f(G) = \max \{|f(u)-f(v)| : (u,v) \text{ is in } E \}$$

and the **bandwidth** of G, denoted $B(G)$, is defined as

$$B(G) = \min \{B_f(G): f \text{ is a labelling of } G \}$$

A labelling f of G is called a **bandwidth labelling** if

$$B_f(G) = B(G)$$

Thus to find the **bandwidth of a graph** G, is to find a labelling f of G such that $B_f(G)$ is minimum.

Clearly, a graph G has bandwidth $B(G)$ if and only if there exists a symmetric permutation of the adjacency matrix

of G for which all the nonzero entries are contained within B(G) diagonals above and below the main diagonal. On the other hand, given a symmetric matrix **M** of order n, one can construct a graph G on n vertices in which vertices i and j, $i \neq j$, are joined by an edge if and only if $m_{ij} \neq 0$. Then **M** has bandwidth B if and only if G has bandwidth B. Thus the two bandwidth problems are equivalent and we will, from now on, generally restrict ourselves to the bandwidth problem for graphs.

A survey of the developments in the bandwidth problem for graphs and matrices up until 1981 is found in [2].

1.2. THE NP-COMPLETENESS OF THE BANDWIDTH PROBLEM

The bandwidth problem restated: To determine the bandwidth of a graph has turned out to be a nontrivial problem. For a graph G on n vertices, there are n! labellings of G and although an exhaustive search through all the labellings will determine the bandwidth labelling of G, that obviously is not a practical solution, particularly for graphs with large n. There is no known algorithm which, for a given graph, will find its bandwidth in time polynomially proportional to the size of the graph. After the theory of intractable functions [10], one can restate more formally the **bandwidth problem** as consisting in deciding, for a given graph G and a given positive integer k, whether G has a

labelling whose bandwidth is less than or equal to k. Obviously, given a graph G, a labelling f of G, and a positive integer k, one can verify in time which is polynomially proportional to the size of G, whether $B_f(G)$ is less than or equal to k. Therefore, one can see that there is a nondeterministic algorithm which solves the bandwidth problem in polynomial time; the bandwidth problem belongs to the complexity class NP. As a matter of fact, in 1976, Papadimitriou [19] proved, in the following theorem, that the problem belongs to the class of the hardest problems in NP.

Theorem 1.1. (Papadimitriou) [19]: The bandwidth problem is NP-complete.

This result implies that it is unlikely for us to find a polynomial time algorithm to solve the bandwidth problem: an algorithm which for a fixed polynomial p in two variables will, for any positive integer n, any graph G on n vertices and any positive integer k, determine in time p(n,k) whether B(G) is less than or equal to k.

Indeed, much sharper theorems concerning the bandwidth problem were proved following the above theorem. The problem is proved to be NP-complete even when certain restrictions are imposed on the graph G admitted as an instance of the problem. Not only is the bandwidth problem NP-complete for trees in general, but also for trees with maximum degree 3, as proved in the following theorems.

Theorem 1.2 (Dewdney) [7]: The bandwidth problem for trees is NP-complete.

Theorem 1.3 (Garey, Graham, Johnson and Knuth) [9]: The bandwidth problem for trees with $\Delta(T)=3$ is NP-complete.

It might be asked whether imposing restrictions on k instead of on G results in an easier problem. Thus, for a fixed k, the **bandwidth-k problem** consists in deciding, for each graph G as input, whether B(G) is less than or equal to k. For k = 1, Gibbs and Poole [11] as well as Fulkerson and Gross [8] have found polynomial time solutions for the bandwidth-1 problem. They have shown that the bandwidth of a graph G is 1 if and only if G is a disjoint collection of paths.

Theorem 1.4: The bandwidth-1 problem has a linear-time solution.

Garey et al. [9] were able to find a linear time solution to determine whether or not the bandwidth of a graph is less than or equal to 2.

Theorem 1.5: (Garey et al.) [9] The bandwidth-2 problem has a linear-time solution.

Garey et al. [9] conjectured that the bandwidth-k problem was NP-complete for some k greater than or equal to

2. However, Saxe [20] proved in 1979 that for every fixed k the bandwidth-k problem has a polynomial time solution.

> **Theorem 1.6** (Saxe) [20]: Let k be a positive integer. Then there exists an algorithm which solves the bandwidth-k problem using time $O(n^{k+1})$ and space $O(n^{k+1})$, where n is the number of vertices in the graph.

Although theoretically a significant statement, Saxe's theorem gives an algorithm which is rather inefficient in view of its space and time requirements of $O(n^{k+1})$.

1.3. BANDWIDTH REDUCTION ALGORITHMS

In view of the NP-completeness of the bandwidth problem as stated in the preceeding section, one does not expect to find an exact polynomial time solution to the general bandwidth problem. On the other hand, by reducing as much as possible the width **b** of the band around the main diagonal which contains all the nonzero entries in an adjacency matrix of a graph, the time efficiency of any computation on this graph can be improved. Therefore, an alternative approach to the bandwidth problem is to try to find a time-efficient algorithm that gives a "good" approximation to the bandwidth of a graph rather than look for procedures to compute algorithmically its exact value. Such an algorithm will try to **relabel** the graph with a view to reduce as much as possible

the bandwidth of the labelling. An algorithm whose purpose is to reduce **b** as much as possible will be called **bandwidth reduction algorithm**. Such an algorithm is expected to produce a good approximation to the bandwidth at best; it is not guaranteed to produce always the exact bandwidth. In most applications a small variation from the exact bandwidth can be tolerated. Theoretically, it should be of interest to reduce algorithmically as much as possible the band-size **b** of the adjacency matrix of a given graph.

In the next chapter, we review some well known bandwidth reduction algorithms.

CHAPTER 2

REVIEW OF BANDWIDTH REDUCTION ALGORITHMS

2.1. THE LEVEL STRUCTURE LABELLING

As seen in Chapter 1, the bandwidth problem is NP-complete even for fairly simple families of graphs. Therefore, bandwidth reduction algorithms have been used for construction of a labelling whose bandwidth approximates the bandwidth of a graph G. As of 1979, 49 bandwidth reduction algorithms had been cited [2]. A brief survey of bandwidth reduction algorithms since 1965, is found in [2]. One of the most successful bandwidth reduction algorithms was introduced in 1969 by Cuthill and McKee [6]. This will be referred to as algorithm CUM. This was the first time a heuristic bandwidth reduction algorithm introduced the notion of a "level structure labelling" (defined below) of a graph. This algorithm was subsequently incorporated into NASTRAN. The algorithm described by Gibbs, Poole and Stockmeyer in 1976 [12], is closely related to and an improvement over the algorithm CUM. The Gibbs, Poole and Stockmeyer algorithm, to be referred to as GPS algorithm, is often used in practice and produces good approximations of bandwidths for certain kinds of graphs, in particular for gridded rectangles and

cylinders, as illustrated by the authors. Both GPS and CUM algorithms are described in [12]. We need the following terminology in order to describe these algorithms.

DEFINITIONS

Let G(V,E) be a **connected graph.** A **Level Structure** of G, denoted by **L(G)** or **L**, is a partition of the vertex set V(G) into sets N_1, N_2, ..., N_k, called **levels**, which satisfy the following condition:

> For $1 \leq i \leq k$, all vertices adjacent to vertices in N_i are in $N_{i-1} \cup N_i \cup N_{i+1}$, where $N_0 = N_{k+1} = \emptyset$.

In particular, if v is any vertex in V, the **level structure rooted at v**, denoted by $L_v(G)$ or L_v, is the level structure for which

$$N_i = \{ u \text{ in } V(G): d(u,v) = i-1 \}.$$

The **width of level structure L**, denoted by **w(L)**, is defined as

$$\max \{ |N_i| : 1 \leq i \leq k \}$$

The **depth of level structure L** is k, the number of levels in L.

Let L(G) be a level structure on G. Let f be a labelling of G which labels arbitrarily the vertices in V level by level, in the sense that for $0<i<k$, if u is in N_i and w is in N_{i+1}, then $f(u)<f(w)$. Such a labelling will be called a **level structure labelling** and a bandwidth reduction algo-

rithm which labels G with such a labelling will be called a **level structure algorithm.**

One can easily prove the following theorem.

Theorem 2.1: Let f be a level structure labelling of a graph G with level structure L, then

$$B(G) \leq B_f(G) \leq 2w(L) - 1.$$

Moreover, if L is a rooted level structure $L_v(G)$, then

$$B_f(G) \geq w(L_v).$$

Both CUM and GPS are level structure algorithms. The former uses a rooted level structure while the latter uses a more general kind of level structure. A brief description of these two algorithms follows.

2.2. THE CUTHILL-McKEE ALGORITHM

This algorithm works in two phases.

Phase I. Generating rooted level structures:

(1) For each vertex v of a suitably determined "low degree", generate a rooted level structure $L_v(G)$.

(2) From level structures generated in step(1) above, consider only those with minimum width as input for phase II.

Phase II.A. Numbering (labelling)

For each one of these rooted level structures, label the graph level by level using the following numbering algorithm.

Step 1. Assign number 1 to the root. From this step onwards, use consecutive positive integers for numbering.

Step 2. For each successive level do the following

(i) First, number in increasing order of their degrees, the vertices adjacent to the lowest number vertex in the previous level, breaking ties arbitrarily.

(ii) Next, number in the same manner, the unnumbered vertices adjacent to the next lowest number vertex in the previous level.

(iii) Repeat (ii) until all the vertices in the current level are numbered. Then apply the algorithm to the next level.

Phase II.B. Choosing the labelling with minimum bandwidth:

The labelling algorithm of the phase II.A. produces at most $|V(G)|$ candidate labellings for the graph G. For each such labelling f, compute the corresponding bandwidth $B_f(G)$. Select a labelling f for which $B_f(G)$ is minimum.

The CUM algorithm proved to be superior to its predecessors [5], [6], and was the most widely used algorithm during the 1970's. Yet, it has a number of shortcomings:

1. Exhaustive search through several rooted structures is necessary to find the ones with minimum width.
2. For each one of the level structures of minimum width, a labelling is generated and its bandwidth is computed.

3. In view of theorem 2.1, the labelling generated can never give a bandwidth smaller than the width of the rooted level structure used. The actual bandwidth of the graph could be considerably smaller.

2.3. THE GIBBS-POOLE-STOCKMEYER ALGORITHM

This algorithm largely removes from the CUM algorithm the shortcomings listed above. The improvements are as follows.

1. A root is selected after generating a relatively few rooted level structures.
2. The graph is labelled only once and the bandwidth is computed only once.
3. Combining two specific rooted level structures, a more general kind of level structure is defined, which results in a labelling whose bandwidth is, in general, closer to the bandwidth of the graph.

The GPS algorithm consists of the following three phases each one of which is described by the corresponding algorithm. First, let us recall some definitions.

Definitions. Let G be a graph. Let d(x,y) denote the distance between vertices x and y in G. Then the **diameter** D of G is given by

$$D = \max \{ d(x,y) \mid x, y \text{ are in } G \}$$

Any (x,y)-path such that d(x,y)= D is called a **diameter path** or simply a **diameter** of G.

Phase I. Finding a pseudo diameter

It was observed that an increase in the number of levels in a level structure will in general decrease the number of vertices in each level and thus, in general, result in a level structure of reduced width. In this sense, it would be ideal to generate level structures which are rooted at the end vertices of a diameter of a graph. Straightforward computation of the shortest distance and the shortest path between all possible pairs of vertices in a graph G on n vertices can be done in time $O(n^3)$. Certainly, this would not be a time-efficient (nor space-efficient for that matter) method to find a diameter in G. Therefore GPS algorithm finds the **endpoints of a pseudo diameter**, defined to be two vertices which are **nearly maximal distance** apart. The pseudo diameter produced by the first phase of the GPS algorithm is close to a diameter on the average and for a large class of graphs, including trees (see theorem 2.6), and the 19 graphs used by the authors [12] for empirical analysis of the algorithm, the pseudo diameter produced, in fact, connects two vertices at maximum distance. Algorithm I below describes the phase I.

ALGORITHM I. Finding endpoints of a pseudo-diameter.

Step 1. Select a vertex **v** of minimum degree.

Step 2. Generate the rooted level structure L_v. Let S_v be the set of vertices in the last level of L_v.

Step 3. For each **u** in S_v and in order of increasing degree, generate L_u. As soon as a u is found such that **depth**(L_u)>**depth**(L_v), replace **v** by **u** and return to step 2.

Step 4. If for all **u** in S_v, **depth**(L_u)=**depth**(L_v), select a vertex **u** in S_v for which $w(L_u)$ is minimum. Then **v** and **u** are the end vertices of a pseudo diameter.

Phase II. Minimizing the level width

Phase I has generated L_v and L_u, where **v** and **u** are the endpoints of a pseudo diameter. In phase II the two level structures are combined into a new level structure whose width is usually less than that of either of the original ones, using the following algorithm.

ALGORITHM II. Minimizing level width.

A. Using the rooted level structures

$$L_v = \{L_1, L_2, \ldots, L_k\} \text{ and } L_u = \{M_1, M_2, \ldots, M_k\}$$

obtained from algorithm I, associate with each vertex **w** of G the ordered pair (i,j), called the **associated level pair**, where i is the index of the level in L_v that contains **w**, and k+1-j is the index of the level in L_u that contains **w**. Note that the pair (1,1) is associated with the vertex **v**, while the pair (k,k) is associated with the vertex **u**.

B. Assign the vertices of G to levels in a new level structure $L = \{N_1, N_2, \ldots, N_k\}$ as follows.

1. (a) For each vertex w of G do the following.
   If the associated level pair of w is of the form (i,i), then place vertex w in N_i; remove from the graph vertex w and all edges incident at w.
   (b) If $V(G) = \emptyset$, then STOP; otherwise go to step 2.

2. The graph G now consists of a set of one or more disjoint connected components $C_1, C_2, \ldots, C_t$ ordered so that $|V(C_1)| \geq |V(C_2| \geq \ldots \geq |V(C_t)|$.

The connected components C_q, $q = 1,2,\ldots t$, are now processed in order of decreasing size. For each component C_q, either the first index from the associated level pairs (i,j) is used for all vertices w in Cq or the second index is used for all w in C_q, whichever minimizes the width of the resulting level structure L, and the vertex w is assigned to the corresponding level, i or j, in L. This is described more precisely in the following.

For each connected component C_q, in order from C_1 to C_t, do the following:

(a) Compute the vector $(n_1, n_2, \ldots, n_k)$ where $n_i = |N_i|$

(b) Compute the vectors $(h_1, h_2, \ldots, h_k)$ and $(l_1, l_2, \ldots, l_k)$ where $h_i = n_i +$ (the number of vertices which would be placed in N_i if the first element of the associated

level pairs were used) and $l_i = n_i +$ (the number of vertices which would be placed in Ni if the second element of the associated level pairs were used).

Thus this step computes the k level widths of L resulting from addition of vertices in C_q using either choice of the index.

(c) Find $h_o = \max \{ h_i : h_i - n_i > 0, 1 \leq i \leq k \}$

and $l_o = \max \{ l_i : l_i - n_i > 0, 1 \leq i \leq k \}$

(i) If $h_o < l_o$, place all the vertices of the connected component in L in the levels indicated by the first elements of the associated level pairs.

(ii) If $h_o > l_o$, use the second element of the level pairs to determine in L the levels of all the vertices.

(iii) If $h_o = l_o$, use the elements of the level pairs which arise from the rooted level structure, L_u or L_v, of smaller width. If the widths are equal, use the first element.

Thus the step B.2. chooses, for each component, the resulting level structure of minimum width.

The algorithm terminates when each vertex of G has been assigned a level in the level structure L.

Phase III. Numbering

The third and the last basic algorithm employed by Gibbs et al. uses the level structure L generated by the foregoing algorithm and, like the CUM algorithm, assigns consecutive

positive integers to the vertices of G level by level. A few modifications to the CUM numbering algorithm were necessary, however, since unlike the rooted level structure of the CUM algorithm, the level structure obtained here is of a more general type. Moreover, with such a structure, it is possible to introduce several operations which serve to minimize level width. A basic version of the numbering algorithm is included here for the sake of completeness.

ALGORITHM III. Numbering.

A. If the degree of u is less than the degree of v, interchange u and v and reverse the level numbers of L by setting N_i to N_{k+1-i}, for i = 1,2,....,k.

B. Define numbering f: set f(v) = 1, set No = $\emptyset$.

For i = 1 to k carry out the following steps using consecutive positive integers from 2 onwards for numbering.

(a) Find the w in N_{i-1}, if it exists, with the lowest f-number over all numbered vertices in N_{i-1} adjacent to unnumbered vertices in N_i. Number the vertices of N_i adjacent to w in order of increasing degree.

(b) Repeat step (a) until all vertices of N_i adjacent to vertices in N_{i-1} are numbered.

(c) Find the w in N_i, if it exists, with the lowest f-number over all numbered vertices adjacent to at least one unnumbered vertex in N_i. Number the as yet unnumbered vertices of N_i adjacent to w in order of increasing degree.

(d) Repeat step (c) until all the vertices of N_i

adjacent to numbered vertices have been numbered.

(e) If any unnumbered vertices remain in N_i, number the one with minimal degree and go to (c).

IMPLEMENTATION, COMPLEXITY and PERFORMANCE of GPS

The GPS algorithm has been implemented in FORTRAN as ACM algorithm No.508 [4]. An improved FORTRAN version by Lewis [16], [17], ACM algorithm No. 582 is claimed to run slightly faster.

The worst case complexity of the GPS algorithm is in the order of n^3 where n is the number of vertices in the graph. The bulk of the time is taken by the pseudo-diameter algorithm. However, its average case complexity appears to be considerably better than this. Based on a set of gridded rectangles and cylinders, which have a close resemblance to many engineering structures, the algorithm had an average time complexity of approximately $O(n^{1.2})$[2].

The performance analysis of the GPS algorithm was provided through empirical studies [12], [13]. A set of 19 test matrices comprising of gridded rectangles and cylinders, which now forms a standard benchmark for production algorithms, was used as a collection of "typical" instances of graphs. On these matrices, the GPS algorithm consistently outperformed the previously popular, "standard" CUM algori-

thm giving a slightly better bandwidth on the average and reducing the time to 12% to 14% of the time required by Cuthill-McKee algorithm [2], [12], [13].

2.4. PROBABILISTIC ANALYSIS OF LEVEL STRUCTURE ALGORITHMS

Both CUM and GPS algorithms have been empirically proved to be far more successful in reducing bandwidth than any algorithms preceding them [5], [13]. This conclusion is theoretically justified by Turner [21]. Turner studies the performance of heuristic bandwidth minimization algorithms on random graphs. He notes that although the level structure algorithms have proved quite successful in practice, it is easy to construct examples where the performance of these algorithms can be arbitrarily poor. Consequently, the worst case analysis provides no insight to their practical success. Moreover, one can design more efficient algorithms using better understanding of their performance through probabilistic analysis. Turner shows that for an appropriate probability distribution on the space of n-vertex graphs with bandwidth less than or equal to k, suitably modified level structure algorithms produce a good bandwidth approximation for almost all graphs. The terminology in [21] is as follows.

For fixed positive integers n and k and $0<p<1$, Turner randomly generates an n-vertex graph G with $B(G) \leq k$ as

follows. For each u and v, with $1 \leq u < v \leq n$, such that $|u-v| \leq k$, u and v are joined by an edge with probability p. The vertices of the resulting graph are then randomly renumbered. This experiment defines the probability distribution $\Omega_n(k,p)$. A property is true for almost all G in $\Omega_n(k,p)$ if the probability that this property holds approaches 1 as $n \rightarrow \infty$. A **modified level structure** is a modified form of the rooted level structure of algorithm CUM. A **modified level algorithm** is a numbering algorithm for the modified level structure. A(G) denotes the bandwidth of the numbering produced by algorithm A on graph G. The **best possible choice** of the root is a vertex u such that the (modified) level structure rooted at u gives a minimum value of the maximum size of the union of two successive level classes. In the following theorems the labelling algorithms are assumed to make the best possible choice for the root.

Turner proves that the level algorithms perform quite well on random graphs.

Theorem 2.2 [21]: Let A be any level algorithm. Let $e>0$, $0<p<1$ be fixed, $k<n$, $\ln n = o(k)$. Then for almost all G in $\Omega_n(k,p)$, $A(G) \leq (3-e)(1+e)B(G)$.

Turner modifies the level structure suitably in order to obtain near optimal performance.

Theorem 2.3 [21]: Let A be any modified level algo-

rithm. Let $e>0$, $0<p<1$ be fixed, $k<n$, $\ln n = o(k)$. Then for almost all G in $\Omega_n(k,p)$, $A(G) \leq 2(1+e)B(G)$.

Turner further describes a specific modified level algorithm, denoted MLA1, and proves the following result.

Theorem 2.4 [21]: Let $e>0$, $0<p<1$ be fixed, $k < n/4$, $\ln n = o(k)$. Then for almost all G in $\Omega_n(k,p)$, $MLA1(G) \leq (1+e)B(G)$.

2.5. A NEW BANDWIDTH REDUCTION ALGORITHM FOR TREES

Although very successful when applied to gridded rectangles and cylinders, the GPS algorithm performs very poorly when applied to trees [18]. In view of Turner's result, it seems probable that a suitably modified level structure algorithm should be the solution to the problem of bandwidth reduction of trees. Opatrny and Miller [18] propose a new algorithm to compute a **level structure of a tree**, to be referred to as algorithm **LST**. For trees, this algorithm finds level structures of smaller widths than those produced by the GPS algorithm thus giving better approximations to their bandwidths.

In phase I, by computing the pseudo-diameter of graph G, the GPS algorithm tries to spread the vertices in V(G) in as many levels as possible, which is a well justified heuristic to minimize level widths in all cases. Algorithm LST also

uses the concept of pseudo-diameter. As noted earlier, (and proved later in theorem 2.6), in case of a tree the pseudo-diameter is actually a diameter of the graph. We prove a stronger statement in theorem 2.5 below.

Phase II of GPS cannot work very well with trees or one-connected graphs. Let x be a vertex on a diameter D of a tree G and let x be in level N_i. Let C be a connected component of G-D, such that d(x,C) = 1. Let

$$r = \max \{ d(x,y) : y \text{ in } C \}.$$

Then the phase II of GPS algorithm will place all vertices in C either in $N_{i-1} \cup N_{i-2} \cup \ldots \cup N_{i-r}$,

or in $N_{i+1} \cup N_{i+2} \cup \ldots \cup N_{i+r}$, instead of spreading them in $N_{i-r} \cup \ldots\ldots \cup N_{i+r}$, thus allowing a more uniform distribution of vertices in the level structure.

Algorithm LST remedies this situation by introducing a recursive algorithm for computing level structure of a tree. We will describe this algorithm in Chapter 3.

2.6. DIAMETERS IN TREES.

We close this chapter with the following results which will be used in Chapter 3.

Theorem 2.5: Let T be a tree. Let v be any vertex in T. Let u be any vertex in the last level of L_v. Then there is a diameter path starting at u.

Proof: Since T is a tree, there is exactly one path

connecting any two vertices in T. Let P be the (u,v)-path. Let x and y be vertices such that the (x,y)-path is a diameter path D in T. Assume $x \neq u$ and $y \neq u$, because otherwise the theorem is verified. Now either D and P have no common vertex (Figure 2.1 (a)), or D and P intersect at exactly one vertex (Figure 2.1 (b)).

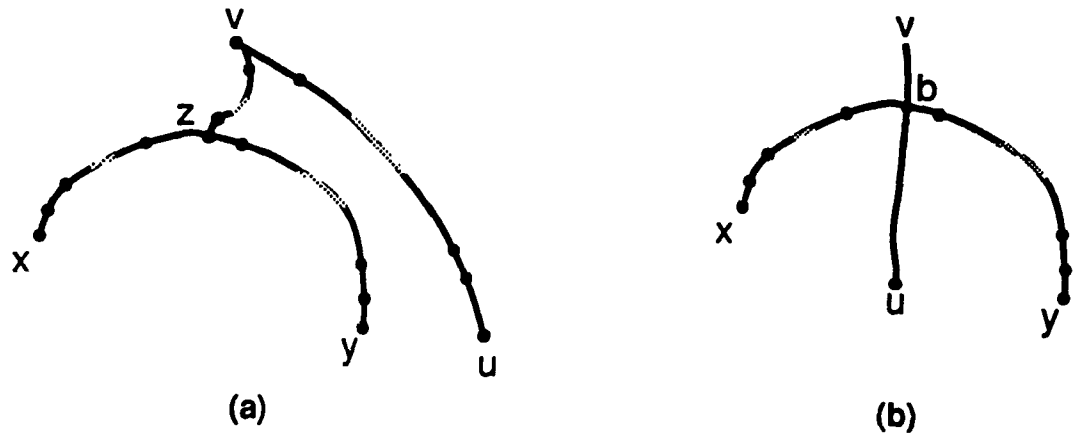


Figure 2.1

In case D and P have no common vertex, there is a unique vertex z on P which is closest to v. Let $|(x,z)| \geq |(y,z)|$*. Clearly, $|(v,z)+(z,y)| \leq |(v,u)|$. This implies that the path (u,v) + (v,z) + (z,x) which starts at u is longer than D, which leads to contradiction. We conclude that D and P must have a common vertex.

Let b be the vertex common to D and P (Figure 2.1 (b));

---

*Notation : (x,y) denotes the path from x to y.
$|(x,y)|$ denotes its edge length.

b may be any vertex on P, including v, except u, because in the latter case u will not be in the last level of L_v. Since u is at a maximum distance from v,

$$|(b,u)| \geq \text{maximum } \{|(b,x)|, |(b,y)|\}.$$

Let $|(b,x)| \geq |(b,y)|$. Then (u,b) + (b,x) is a path starting at u and $|(u,b) + (b,x)| \geq |(x,y)|$.

An immediate consequence is the following corollary.

Corollary 2.5.1: Let T be a tree. Let v be any vertex in E. Let u be any vertex in the last level of L_v. If

$$\text{length } (L_v) = \text{length } (L_u)$$

then (u,v) is a diameter path in T.

The following theorem is now obvious. It plays a crucial role in many results in Chapter 3.

Theorem 2.6: Let T be a tree. Then the pseudo diameter produced by phase I of algorithm GPS applied to T gives a diameter path of T.

CHAPTER 3

A NEW LEVEL STRUCTURE ALGORITHM FOR TREES

3.1. A NEW LEVEL STRUCTURE ALGORITHM FOR TREES

As we remarked in Chapter 2, the GPS bandwidth reduction algorithm, although very successful when applied to rectangular grids and cylinders, performs poorly when applied to trees. We also noted, in view of Turner's result, that the general heuristic of level structure algorithms can be very effectively used for the bandwidth reduction problem. Therefore it would seem reasonable that in order to find better bandwidth approximations for trees, we must define a new, modified level structure for trees which exploits the special properties of trees and thus results in smaller width than the width given by the GPS algorithm. This is exactly what is achieved by the algorithm **LST**, a new algorithm to compute a **level structure of a tree**, proposed by Opatrny and Miller [18].

First we need the following **notations and terminology.**

Let T be a tree and $P = (p_1, p_2, \ldots p_k)$ a diameter path in T. A **diameter level structure** of T is the level structure

$$N = \{ L_1, L_2, \ldots, L_k \}$$

where for each i, $1 \leq i \leq k$, $L_i = \{ p_i \}$.

T-P is the graph obtained by deleting from T all the vertices in P and all the edges incident on these vertices.

Any connected component of T-P will be called an **off-diameter subtree.**

Let C be a subtree and x a vertex not in C. Then **x is adjacent to C** if $d(x,C) = 1$.

Let us recall that the algorithm GPS computes, in phase I, a pseudo-diameter of the graph G, thus trying to spread the vertices in G in as many levels as possible. Algorithm LST also uses this well justified heuristic to minimize level widths and computes a pseudo-diameter of the input tree. As noted in theorem 2.6, a pseudo-diameter, thus obtained, is in fact a diameter of the tree.

However, the phase II of GPS cannot work very well with trees. Let x be a vertex on the diameter P of a tree T and let x be in level N_i. Let C be a connected component of T-P, which has a vertex z adjacent to x. Let

$$r = \max \{ d(x,y) : y \text{ in } C \}.$$

Then the phase II of GPS algorithm will place all vertices in C either in $N_{i-1}, N_{i-2}, \ldots N_{i-r}$, or in $N_{i+1}, N_{i+2}, \ldots N_{i+r}$. However, since C is a tree, there is no vertex on P other than x which is adjacent to C. It follows that as long as the adjacent vertices of C are in the same or consecutive levels, there is no constraint imposed on their levels other

than that z must be in level i-1, or i, or i+1. Moreover, C itself being a tree, must have its vertices spread out in branches with non-adjacent endpoints. Both these considerations allow us to "open up" the branches of the connected component C and spread C through levels $N_{i-r},.....,N_{i+r}$ and thus obtain a more uniform distribution of vertices in the level structure.

Algorithm LST exploits these properties of trees and introduces a recursive algorithm for computing level structure of a tree as described below in three broad steps.

The three main steps in algorithm LST

Step 1. Given a tree T, compute a diameter path P of T. Create the diameter level structure of P.

Step 2. For every connected component C of T-P, of size 3 or more do:

(a) Apply steps 1 and 2 of the algorithm recursively and find a level structure of C.

(b) Merge the level structure of C into the diameter level structure of P to obtain a level structure of T.

Step 3. Finally, merge all the components of size less than 3 into the level structure in such a way that the differences between the widths of consecutive levels are as small as possible.

An alternative version: Algorithm LST1.

The three main steps in algorithm LST as described above treat "small subtrees", that is subtrees of size 1 or 2, in a different manner than the rest of the subtrees, by postponing processing them till the very end. An alternative approach could be to treat all the subtrees exactly the same way irrespective of their size, and merge them into the "parent" diameter structure as and when they are encountered. This can be achieved by replacing Step 2 and Step 3 above with Step 2' as shown below. We will refer to this version as algorithm LST1.

Algorithm LST1

Step 1. Given a tree T, compute a diameter path P of T. Create the diameter level structure of P.

Step 2'. For every connected component C of T-P do

(A)I. If the size of C is less than 3, create a diameter level structure as follows:

(a) assign levels 1 and 2 to the vertices of C if there are 2 vertices.

(b) if C has only 1 vertex, assign level 1 to it.

II. If size of C is 3 or more, apply the algorithm recursively and find a level structure of C.

(B) Merge the level structure of C into the diameter level structure of P.

In the rest of this chapter, we generally refer to the original version of algorithm LST unless otherwise stated.

3.2. A PSEUDO-CODE FOR ALGORITHM LST.

A pseudo-code for algorithm LST is given below. Some explanation of terminology and notation used, as well as the procedures used, is necessary before presenting the pseudo-code for the main algorithm.

The reverse level structure L^R: Let $L = \{N_1, N_2, \ldots . N_m\}$ be a level structure. Then the reverse of level structure L is $L^R = \{N^R_1, N^R_2, \ldots . N^R_m\}$ where for all r, $N^R_r = N_{m+1-r}$. In other words,

$$L^R = \{N_m, N_{m-1}, \ldots . N_1\}.$$

3.2.1 Procedure DIAMETER (T, P, m):

For a given tree T, this procedure computes the diameter path $P = \{ u_1, u_2, \ldots . u_m \}$ of length m. The algorithm used is a modification of the standard algorithm to find a pseudo-diameter of a general connected graph, the algorithm I in the phase I of the GPS algorithm in 2.3. In case of a tree, one can considerably simplify the algorithm by noting the following:

(1) To find a vertex of minimum degree in a general graph, one has to scan the degrees of all the vertices in the

graph; whereas, in case of a tree one can choose the first scanned vertex of degree 1. Let this vertex be v. Then v can be used to generate the first rooted level structure L_v of the algorithm.

(2) Starting with the first rooted structure L_v, we note that all vertices in the last level of L_v are of degree 1, and therefore there is no need to sort them by degree as in GPS algorithm. In fact, it is sufficient to choose arbitrarily any vertex u in this level class and generate only one rooted level structure with u as the root. By theorem 2.5, it follows that u is an endpoint of a diameter path in T. Moreover, in view of the nature of the Step 2.A of our algorithm LST in 3.1.(details to follow), there is no need to choose that particular u from L_v for which L_u has the minimum width. Thus the algorithm produces a diameter of T with one endpoint at u, after generating at most three rooted level structures: with roots at v, at u, and if d(u,v) is not equal to the diameter of T, then at a third point w which is in the last level of L_u.

A pseudo-code for procedure DIAMETER (T,P,m).

```
Procedure DIAMETER (T,P,m);
{Computes a diameter path P={u1,u2,...um} of T, of length m}
  begin
    find a vertex v of degree 1;
    {find 2 vertices at maximum distance apart.}
```

```
   generate rooted level structure Lv;
   for  every vertex x in T do
       record levelv[x], the level of x in Lv;
   repeat
     pick a vertex u in the last level of Lv;
     generate rooted level structure Lu;
     for  every vertex x in T do
       record levelu[x], the level of x in Lu;
     if length (Lu) > length (Lv)
        then
          replace v, Lv, length(Lv), array levelv with
                   u, Lu, length(Lu), array levelu resp.
   until v not replaced with u;
   m := length(Lv);
{Now there are two rooted level structures, Lv and Lu, both
of equal length. Next, find the diameter path from v to u}
   for each vertex x in T do
     begin
       if (levelv[x] = i) and (levelu[x] = m+1-i)
          then set ui := x;
     end;
  end; {procedure DIAMETER}
```

3.2.2 Procedure MERGE (L,L',u,j):

Here $L = \{N_1, N_2, \ldots N_m\}$ and $L' = \{N'_1, N'_2, \ldots N'_{m'}\}$

are two level structures. The vertex u is assigned a level k in L', that is u is in N'_k. j is an integer, $1 \leq j \leq m$. The effect of this procedure is to assign to the vertices in L' their level numbers in L in the following manner:

(1) u and all the vertices in N'_k are assigned to the level class N_j.

(2) For any integer r, such that $1 \leq k+r \leq m'$, all the vertices in N'_{k+r} are assigned to the level class N_{j+r}.

The resulting level structure L is said to be obtained by merging level structure L' into level structure L. Clearly, this will result in increased widths for some level classes in L.

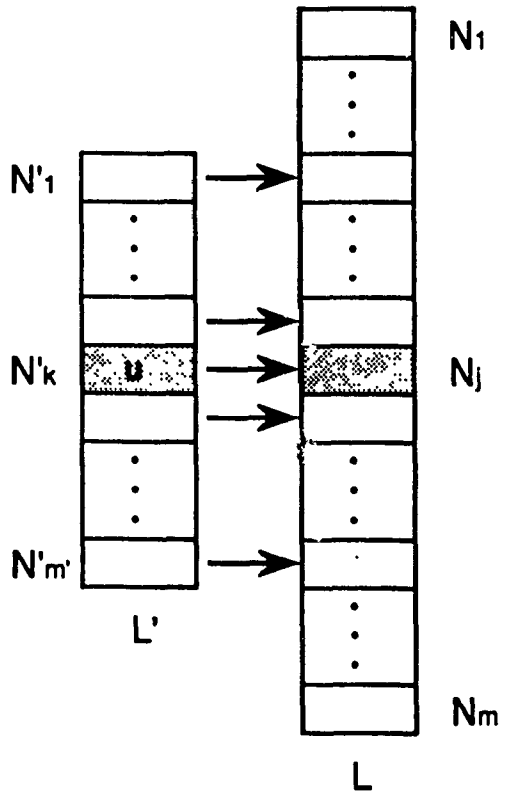


Figure 3.1

A pseudo-code for procedure MERGE(L,L',u,j);

```
procedure MERGE (L,L',u,j);
{Here L = {N1,N2,....Nm}  and L' = {N'1,N'2,....N'm'}  are
two level structures. The vertex u is in level k in L'.  The
two structures are merged in such a way that the level of u
in L is j. The merged structure is stored as L.}
begin
  k := index of the level of u in L';
  for every level class N'i in L' do
    begin
      {find the index of the level in L for vertices in N'i}
      t := j+i-k;
      for every vertex x in N'i do
          assign x to level t in L;
        {N'i gets merged into Nt}
    end;
update widths of level classes in L;
update width(L);
end; {procedure MERGE }
```

3.2.3 Function WIDTHMRG (L,L',u,j)

This function computes the width of the level structure produced by MERGE (L,L',u,j) without actually computing the

merged level structure. Algorithm LST uses this width as a criterion to choose the best possible merger out of a number of feasible alternatives.

A pseudo-code for function WIDTHMRG (L,L',u,j)

```
Function widthmrg (L,L',u,j);
{All the parameters are as in procedure MERGE.
Arrays width1[1..m] and width1'[1..m'] store the widths of
the corresponding level classes of L and L' resp. We may
assume that these widths are previously computed and saved
whenever a level structure is generated newly or augmented by
a merge operation. width(L) and width(L') are also assumed to
be available }
begin
  maxw := 0;
  k := index of the level of u in L';
  for i:= 1 to m' do
    begin
      {find t where N'i gets merged into Nt}
      t := i+j-k;
      w := width1[t] + width1'[i];
{find the width of the level class t after the implied
merger. At the same time, find the maximum of the resulting
class widths}
      if (w > maxw ) then maxw := w;
    end;
```

```
{ find maximum width of all levels in resulting L}
 widthmrg := max (maxw, width(L));
end; {procedure widthmrg }
```

3.2.4 A pseudo-code for algorithm LST now follows [18]

```
Algorithm LEVEL_STRUCTURE (T,L,f)
{Computes recursively the level structure L of tree T.  f is
a boolean variable which is true for T and false for a
subtree}
          begin

{PART I :Compute a diameter path P = {u1,u2,...um} of T}
          DIAMETER (T, P, m);
            for i:= 1 to m do
               begin
{initialize the diameter level structure of P}
                Ni := {ui};
                 mark ui;
               end;
```

```
{PART II :Compute level structures for off-diameter subtrees
of size 3 or more and merge these structures into the
diameter level structure of P}
          for i:= 2 to m-1 do
             for every unmarked neighbor x of u_i do
               begin
                  delete the edge (u_i, x);
                  generate the component C of T-P
                  containing x;
                 while |V(C)| ≥ 3 do
                  begin
                    LEVEL_STRUCTURE (C,L',false);
                    for j := 1 to 3 do
{Compute the widths of the three merges of  L and L'}
                         s_j := WIDTHMRG (L,L',x, i+j-2);
                    for j := 4 to 6 do
{Compute the widths of the three merges of   L  and L'^R}
                         s_j := WIDTHMRG (L,L'^R,x,i+5-j);
                    s := min {s_j : 1≤j≤6};
                    k := min {j : s_j = s};
{determine the best merger rule: the one with minimum width
for merged structure; in case of a tie, choose the rule with
smallest index.}      if k ≤ 3 then
                            MERGE (L,L',x,i+k-2)
                         else MERGE (L,L'^R,x,i+5-k);
                  end {while}
```

```
{PART III: Complete the level structure of T by assigning
levels to all "small"  subtrees, of size 2 or 1, of T.  This
part is not included in the recursive calls to process
subtrees of T.}
          if f then
               begin
                 for i := 1 to m do
                    while N_i contains a vertex  adjacent
                            to an unmarked vertex x do
                     begin
                       find the unmarked component C
                                containing x;
                       place vertices in C in N_(i-1), N_i, N_(i+1), so
                          that N_(i-1), N_i, N_(i+1) are as even
                                as possible;
                            mark vertices in C;
                     end
end { algorithm LEVEL_STRUCTURE }
```

3.3. AN EXAMPLE OF APPLICATION OF LST

The following example demonstrates the application of algorithm LST.

Example 3.1: Application of algorithm LST

The steps in this demonstration follow the version LST1 of the algorithm. Let T be the tree on 30 vertices illustrated in figure 3.2. The maximum degree in T is 5. Algorithm LST works as follows.

Step 1.1: Starting from vertex 1, and in the increasing order of vertex number, vertices are scanned to find the first vertex of degree 1. It is vertex 11. The rooted level structure L_{11} is obtained as in Figure 3.3. It has depth 12 and width 6.

Step 1.2: There are 3 vertices in the twelfth level of L_{11}. The first one among these, vertex 29 is chosen and the rooted level structure L_{29} is generated as shown in Figure 3.4. The depth of this level structure is 13 and its width is 5. Level 13 has exactly one vertex, vertex 27.

Step 1.3: Next the rooted level structure L_{27} is generated (not shown). One can verify that this level structure also has 13 levels. So vertex 29 is the start and vertex 27 is the end of a diameter D of T. These three steps also

illustrate the statement of Theorem 2.5.

Figure 3.4 also lists for each vertex its associated level pairs with reference to the structures L_{29} and L_{27} resp. From these we get the diameter path

$$D = (29,8,2,1,12,9,7,3,13,5,10,20,27).$$

Levels 1 to 13 are assigned to these vertices in order, to obtain the diameter level structure **LD** for T. The depth of **LD** is 13, each level in **LD** has width 1, and so the width of **LD** is 1.

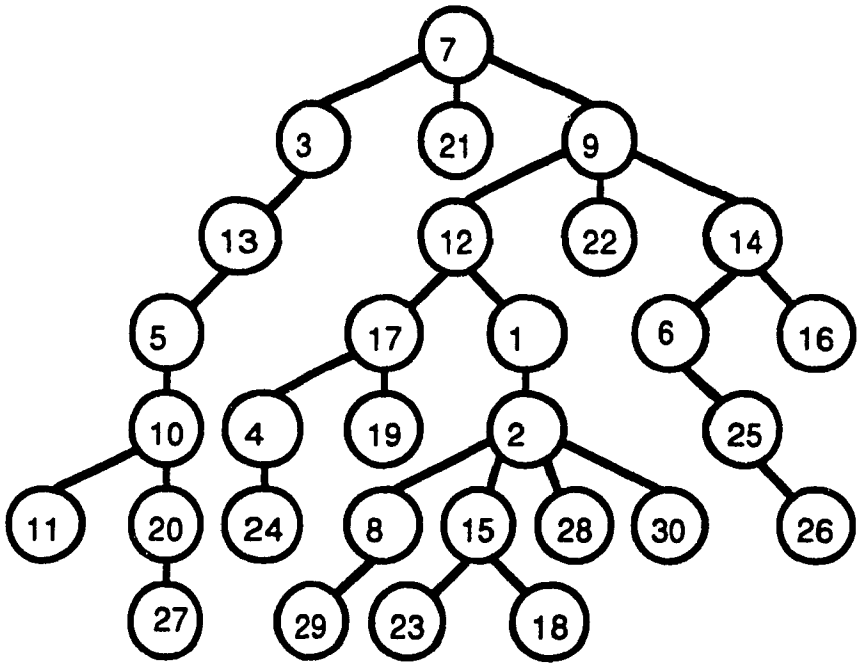


Figure 3.2

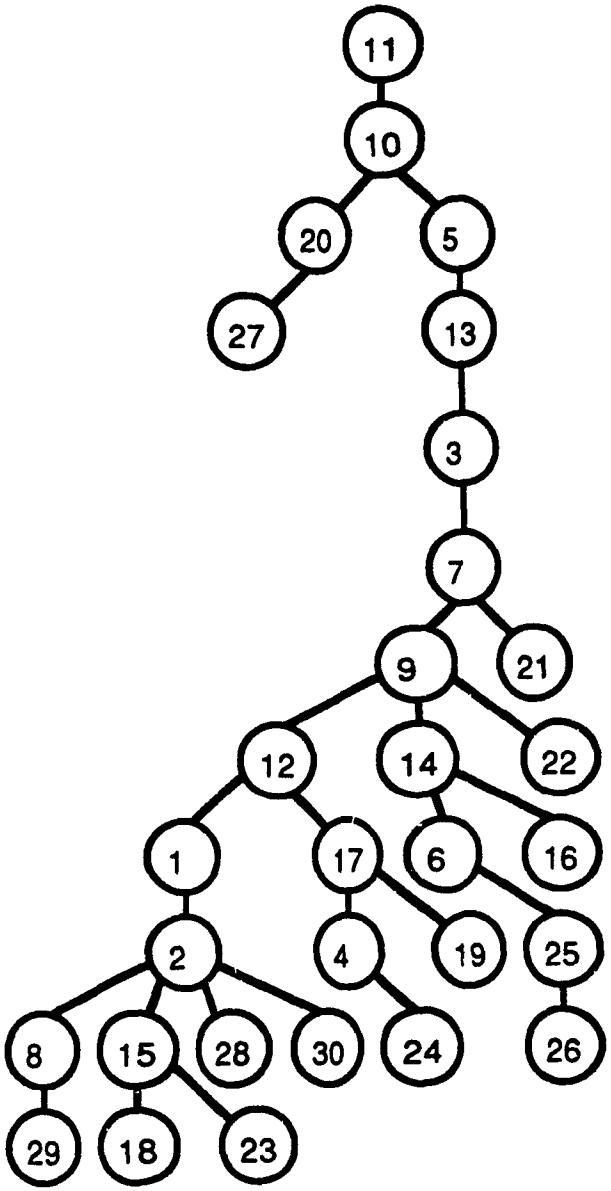

11
10
20
5
27
13
3
7
9
21
12
14
22
1
17
6
16
2
4
19
25
8
15
28
30
24
26
29
18
23

Figure 3.3

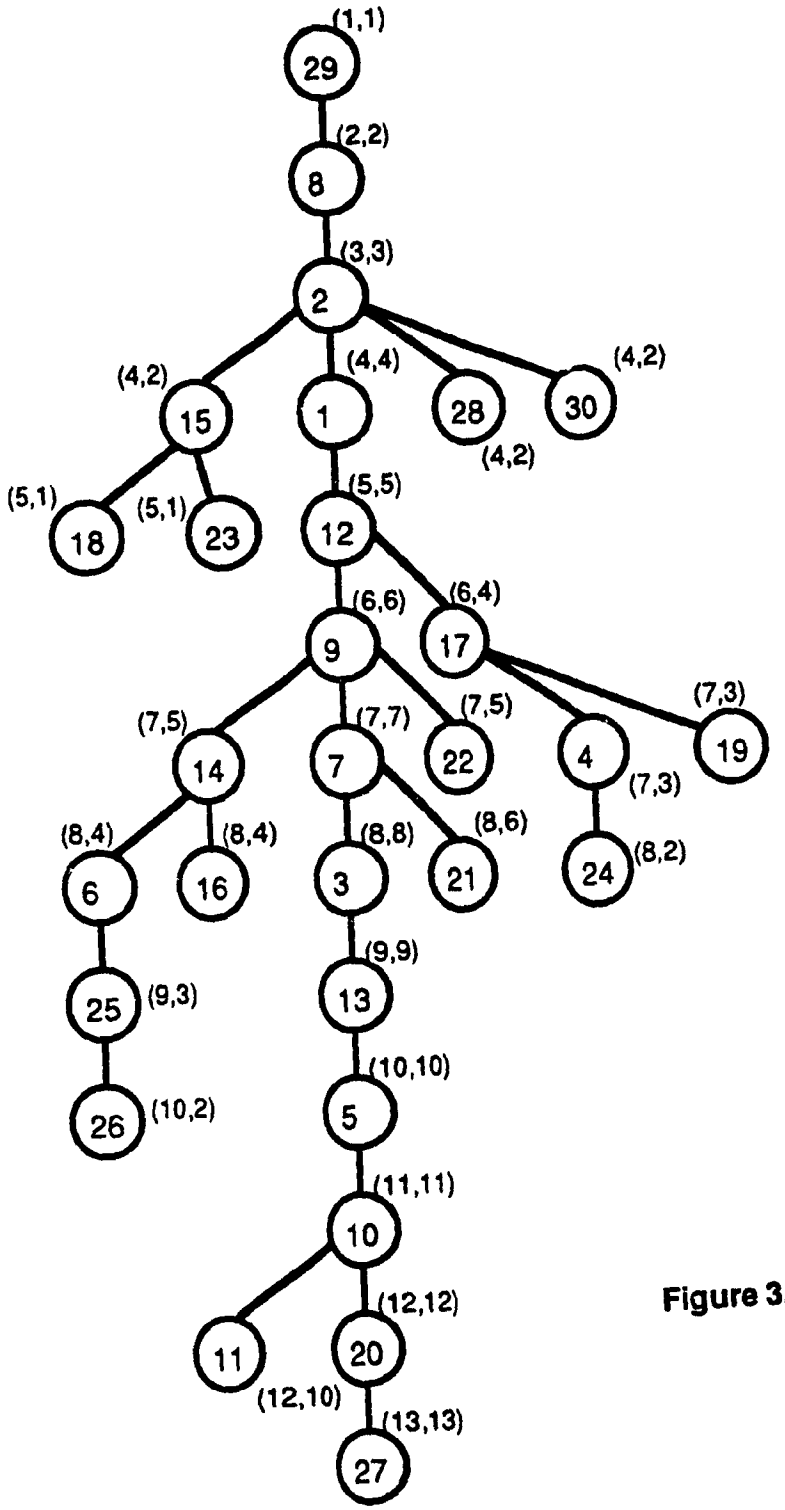

(1,1)
29
(2,2)
8
(3,3)
2
(4,2)
15
(4,4)
1
28
(4,2)
30
(4,2)
(5,1)
18
(5,1)
23
(5,5)
12
(6,6)
9
(6,4)
17
(7,5)
14
(7,7)
7
(7,5)
22
4
(7,3)
(7,3)
19
(8,4)
6
(8,4)
16
(8,8)
3
(8,6)
21
24
(8,2)
(9,9)
13
25
(9,3)
(10,10)
5
26
(10,2)
(11,11)
10
(12,12)
11
20
(12,10)
(13,13)
27

Figure 3.4

Step 2: Next, starting from level 2 in **LD** and going downward, the off diameter subtrees adjacent to the vertices on D are visited.

Step 2.1: The subtree T_1 with vertex set {15,18,23} of 3 vertices is adjacent to vertex 2 in level 3 on D. Applying recursively Steps 1.1-1.3 to it, we get (23,15,18) as the diameter path D_1 for T_1. Levels from 1 to 3 are assigned to these vertices in order, giving the diameter level structure **LD1** for T_1. The depth of **LD1** is 3 and its width is 1. As a matter of fact, all the diameter level structures will have width 1 to start with.

Next, **LD1** is merged into **LD**. The 6 possible mergers are considered as shown in Figure 3.5.

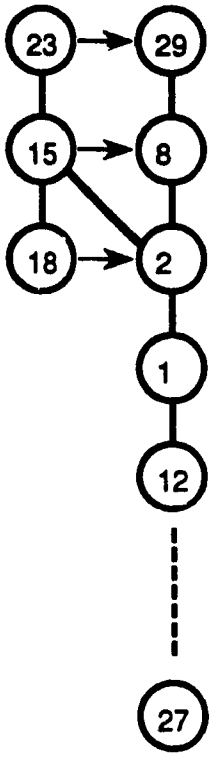
Since every one of the six merger rules will, if applied, increase the width of **LD** to 2, the first rule is chosen. The vertices 23, 15, 18 are assigned levels 1, 2, 3 resp. The widths of the first three levels in **LD** are increased by the widths of the corresponding merged levels of **LD1**, in this case by 1 each.

Step 2.2: Next to be considered is the subtree T_2 consisting of a single vertex 28, which is adjacent to vertex 2 in the level 3 of D. Since in this demonstration, we use the alternative step(2') in the description of the algorithm and consider vertex 28 as forming a single-level

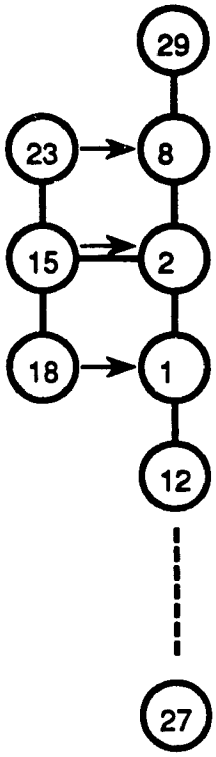
diameter level structure **LD2** of tree T_2. As in step 2.1 above, in order to merge **LD2** into **LD**, vertex 28 must be assigned to one of the three levels 2, 3, or 4 in **LD**, whichever increases the width of **LD** by a minimum. Since assigning 28 to either level 2 or 3 will increase the width of **LD** to 3, whereas the choice of level 4 maintains this width at 2, vertex 28 must be assigned level 4, by using merger rule 3. The width of level 4 in **LD** is increased to 2. The width of **LD** remains unchanged at 2.

Step 2.3: Subtree T_3 is also adjacent to vertex 2 and consists of exactly one vertex, 30. Working as in step 2.3, vertex 30 is assigned to level 2 in **LD**. Here, merger rule 1 is used. The width of level 2 in **LD** as well as the width of **LD** is increased to 3.

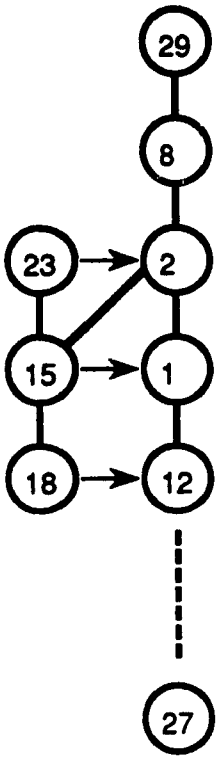
Step 2.4: Subtree T_4 is adjacent to vertex 12 in level 5 in **LD** and has vertex set {24,4,17,19} of 4 elements. Its diameter level structure **LD4** assigns levels 1, 2, 3, 4 to vertices 24, 4, 17, 19 resp. Merger rule 1 is used to assign vertices 24, 4, 17, 19 to levels 3, 4, 5, 6 resp. in **LD**, increasing their widths to 3, 3, 2, 2 resp. The width of **LD** remains 3.



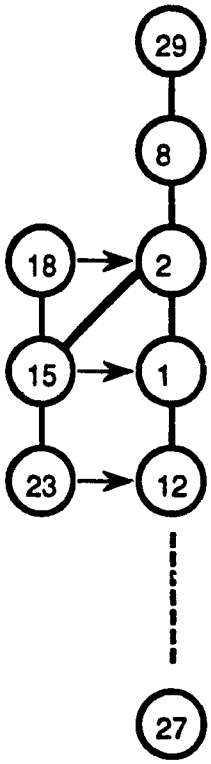
Rule 1



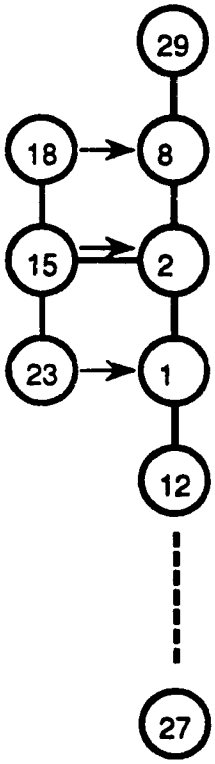
Rule 2



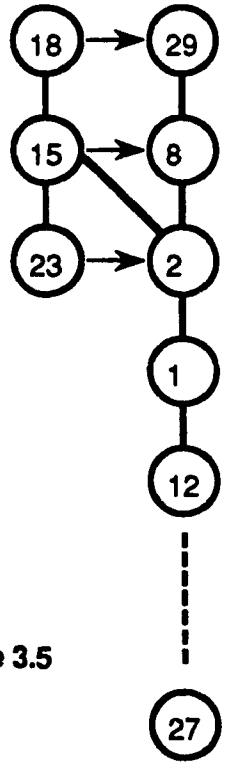
Rule 3



Rule 4



Rule 5



Rule 6

Figure 3.5

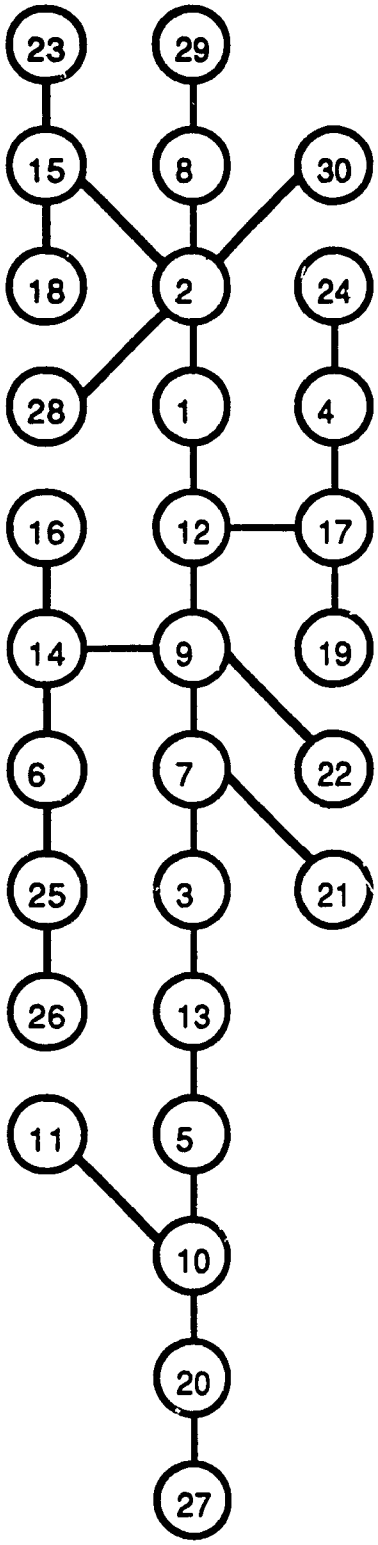

23
29
15
8
30
18
2
24
28
1
4
16
12
17
14
9
19
6
7
22
25
3
21
26
13
11
5
10
20
27

Figure 3.6

Step 2.5: Subtree T_5 is adjacent to vertex 9 in level 6 in **LD** and has vertex set {26,25,6,14,16} of 5 elements. Its diameter level structure **LD5** assigns levels 1, 2, 3, 4, 5 to vertices 26, 25, 6, 14, 16 resp. Since each one of the merge rules 1 to 4 will increase the width of **LD** to 4 whereas merger rules 5 and 6 will both keep this width unchanged, merger rule 5 is used. It assigns vertices 16, 14, 6, 25, 26 to levels 5, 6, 7, 8, 9 resp. in **LD**, increasing their level widths to 3,3,2,2,2 resp. The width of **LD** remains 3.

Steps 2.6, 2.7, 2.8: The three remaining subtrees are processed as above and vertices 22, 21, 11 are assigned levels 7, 8, 10 resp. in **LD**. The width of **LD** remains 3.

The resulting level structure **LD** is the outcome of algorithm LST. Figure 3.6 shows the transformed, stretched out tree. Each horizontal line denotes one level class. A level structure labelling f can be shown to give bandwidth 4.

If algorithm GPS is applied to this tree, it creates a level structure of width 4. Further application of the labelling algorithm of GPS results in bandwidth 5. This example appears as problem no. 3 in Table 1 in Chapter 5.

3.4 TIME COMPLEXITY OF LST

This discussion of time complexity of LST is based on the second version of the algorithm, algorithm LST1. It is

not difficult to see that the time complexity function for the additional step (3) in version 1 of LST will not surpass the complexity function derived in this section.

Let T be a tree on **n** vertices and let **d** be the maximum degree in T. We assume that T is sparse, meaning that the adjacency matrix of T is sparse. We assume that T is stored in one of the typical compact forms used for storing sparse graphs, **the adjacency table,** which is an **n*d** array organized as follows. If the vertex k has degree r, then the k^{th} row of the adjacency table lists the r vertices adjacent to vertex k, followed by d-r zeroes. Using the adjacency table instead of the adjacency matrix results in a significant reduction in the size of the input, which is a parameter in the complexity function. Let us recall the two main steps in algorithm LST.

Step 1: Find a diameter level structure L for T.

Step 2: For each off-diameter subtree T' of T do

(i) find a level structure L' of T' recursively and

(ii) merge L' into L.

thus making L a level structure for T.

First we determine the time required for the main steps in the computation of a diameter of a tree on n vertices with maximum degree d.

1. To find the degrees of all vertices will take O(d*n) time. However, this step is executed only once for the main

tree and not repeated when computing diameters of subtrees.

2. To find a vertex of degree 1 will require O(n) time.

3. To find a rooted level structure at a vertex v is essentially breadth first search. Starting with the root, vertices are entered in a queue in the order of their distance from the root. Every entry in the adjacency table is read exactly once to determine all the neighbors of a vertex at the top of the queue and to make successive additions to the queue. Time taken will be O(d*n). This will include time taken to record the level of each vertex, as well as to find the width and the depth of the structure generated.

4. By theorem 2.5, it is necessary to generate at most 3 rooted level structures to get a diameter path. Therefore the time needed is still O(d*n). Replacing one vertex and its rooted level structure with another vertex and its rooted level structure resp. will not cost more than O(n) time.

5. When the ends of a diameter path are found, to form the associated level pairs for each vertex and to assign levels to the vertices on the diameter path will cost only O(n) time.

Thus we can state the following Lemma.

Lemma 3.1: Let T be a tree on n vertices with maximum degree d. Then a diameter level structure L of T can be found in time O(d*n).

Let T_1, T_2, ... T_r be the connected components of size n_1, n_2, ... n_r resp. of T-D. Note that T_1, T_2, ... T_r are off-diameter subtrees of T. Clearly,

$$n_1 + n_2 + \ldots + n_r < n$$

Further, in each of these subtrees the maximum degree will be at most d. Therefore, by lemma 3.1, the total time to compute diameter level structures for all these k subtrees will be

$$\begin{aligned} & O(d*n_1) + O(d*n_2) + \ldots + O(d*n_r) \\ &= O(d*n_1 + d*n_2 + \ldots + d*n_r) \\ &< O(d*n). \end{aligned}$$

We view this task of finding the r diameter level structures as being on the second level of recursion. We may conclude that the time required to compute all the diameter level structures on the second level of recursion is at most O(d*n). Now let D_1, D_2, ... , D_r be diameters in T_1, T_2, ... , T_r resp. as computed in the second level of recursion. We next want to investigate the third level of recursion, and to determine the time required to compute the diameter level structures of all the subtrees of T_1-D_1, plus all the subtrees of T_2-D_2, ... up to and including all the subtrees of T_r-D_r. Using the same reasoning as above, we can see that

for all i, $1 \leq i \leq r$, the time to compute the main diameter

structures of all the subtrees of $T_i - D_i$ will be $O(d*n_i)$. Adding over all i, $1 \leq i \leq r$, we see that the total time to compute the diameter level structures of all the subtrees on the third level of recursion is

$$O(d*n_1) + O(d*n_2) + \ldots + O(d*n_r)$$
$$= O(d*n_1 + d*n_2 + \ldots + d*n_r)$$
$$< O(d*n).$$

Generalizing inductively, we can conclude that the time required to compute all the diameter level structures on any level of recursion is $O(d*n)$. Adding the contributions to execution time from all the levels of recursion gives us the following result.

Lemma 3.2: Let T, n and d be as in lemma 3.1. Let k be the maximum depth of recursion attained by algorithm LST when applied to T. Then the time required to compute all the diameter level structures is $O(k*d*n)$.

Next we find a bound on the depth of recursion, k, used in lemma 3.2. In the following discussion, **S(P)** denotes the number of vertices in the path P. First we note the following property of diameter paths in trees.

Lemma 3.3: Let D be a diameter path in a tree T. Let T' be a connected component of T-D. Let D' be a diameter path in T'. Then $S(D) \geq S(D')+2$.

Proof: Let vertices u on D and v on D' be such that d(u,v) is minimum. (See Figure 3.7). Let P be the path in T from u to v. Then $|P| \geq 1$, where absolute value denotes the number of edges in P. Let u divide D into 2 paths A and A' and let v divide D' into 2 paths B and B'. Let us assume $|A| \leq |A'|$ and $|B| \leq |B'|$. B may be void. We claim that $|B'| < |A|$. Because, if $|B'| \geq |A|$, then the path in T given by A'+P+B' will be longer than D. Therefore A has at least one more vertex than B'. It follows that D has at least 2 more vertices than D'.

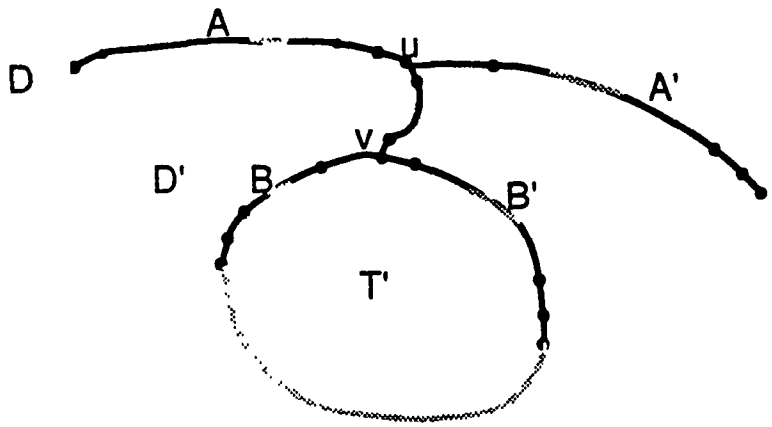


Figure 3.7

We can now state the following theorem.

Theorem 3.1: Let T be a tree on n vertices. Then in an application of algorithm LST to T, the depth of recursion cannot exceed $\sqrt{n}$.

Proof: Let $T_1 = T$ and let D_1 be the diameter of T_1 computed when algorithm LST is applied to T_1. Let k be the maximum depth of recursion. Then there is a sequence of subtrees $T=T_1, T_2, \ldots, T_k$, and a sequence of corresponding diameter paths $D=D_1, D_2, \ldots, D_k$, such that for all i, $2 \leq i \leq k$,

(1) T_i is a connected component of $T_{i-1}-D_{i-1}$,

(2) Algorithm LST computes D_i as a diameter path in T_i.

Since D_k has at least 1 vertex, by lemma 3.3 D_{k-1} has at least 3 vertices. Applying lemma 3.3 to all the preceding diameters in the sequence, we have

$$1 + 3 + \ldots + (2k-1) \leq S(D_k) + S(D_{k-1}) + \ldots + S(D_1) \leq n$$

Adding the arithmetic sequence on the left, $k^2 \leq n$, therefore $k \leq \sqrt{n}$.

Example 3.2: Consider the tree on 36 vertices in Figure 3.7 (an inverted conifer). An application of algorithm LST may choose $D_1 = (1,2,3,4,5,6,7,8,9,10,11)$ as the diameter in the first recursive call. In this case the remaining graph has only one connected component. Similarly, if in every successive recursive call, LST chooses a hori-

zontal line as the diameter path, the remaining graph is a tree. The depth of recursion in this case will be 6, which is precisely square root of 36. Any other choice of diameter at any stage of recursion, except the very last stage, will reduce the depth of recursion, as can be easily verified. Clearly, for any k, such a tree with k horizontal levels and k^2 vertices can result in recursive calls of depth k.

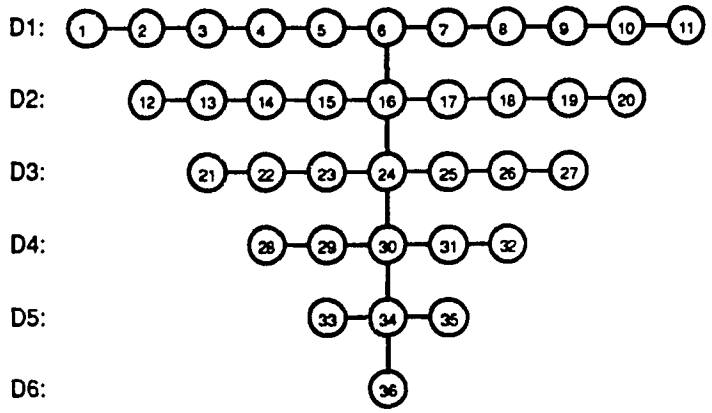


Figure 3.8

Combining lemma 3.1 and theorem 3.1 gives the following result.

Theorem 3.2: Let T be a tree on n vertices and let $\Delta(T)=d$. Then the time required for algorithm LST to compute all the diameter level structures in T is $O(d*n^{3/2})$.

The merge operation : When a level structure L' is merged into a level structure L, the following two steps are executed.

Step (1): The vertices in L' are assigned new level numbers.

Step (2): The widths of levels in L are modified with the additions of the widths of the merged classes of L'.

First, let us compute the time required for step (1). Note that since the depth of recursion cannot exceed $\sqrt{n}$, a vertex may be part of a merge operation at most $\sqrt{n}$ times and therefore may get relabelled at most $\sqrt{n}$ times. Therefore this step in all the merge operations can be executed in total time $O(n*\sqrt{n})$. Relabelling of vertices can be avoided by opting for more book-keeping and labelling the vertices only once, which in turn will reduce this time to $O(n)$.

Next, we note that every level structure L' is merged only once in some level structure L of bigger length. While computing the widths of the augmented structure L, one may ignore those levels in L which were not affected by the merge operation. Then the number of levels modified in step (2) above will be equal to the length of L'. Since the lengths of all level structures generated during the application of the algorithm LST cannot exceed n, the time for modifying all the level widths during all the merge operations will be $O(n)$. Thus the execution of the two steps above together will cost time $O(n^{3/2})$.

We can now state the following theorem.

Theorem 3.3: Let T be a tree on n vertices and let $\Delta(T)=d$. Then the algorithm LST computes a level structure of T in time $O(d*n^{3/2})$.

Proof: By theorem 3.1, the time to compute all the diameter level structures is $O(d*n^{3/2})$. As seen above, the time required to compute all the merge operations is $O(n^{3/2})$. Therefore the total time required by algorithm LST is $O(d*n^{3/2})$.

3.5. PERFORMANCE OF LST

In this section we evaluate by means of examples the ability of algorithm LST to compute near optimum results.

Near optimum results:

It is easy to see that algorithm LST, when applied to some simple trees, like spiders with legs of length greater than 1, will construct a level structure which can lead to a bandwidth labelling. In this section we will show that algorithm LST can be combined with a labelling algorithm to obtain the bandwidth of a nontrivial family of trees for which the performance of the GPS algorithm is particularly poor.

Example 3.3. Bandwidth labelling of a nontrivial family of trees (Opatrny and Miller) [18]:

Let T_1 and T_2 be as shown. For $n \geq 3$, define tree T_n recursively as follows:

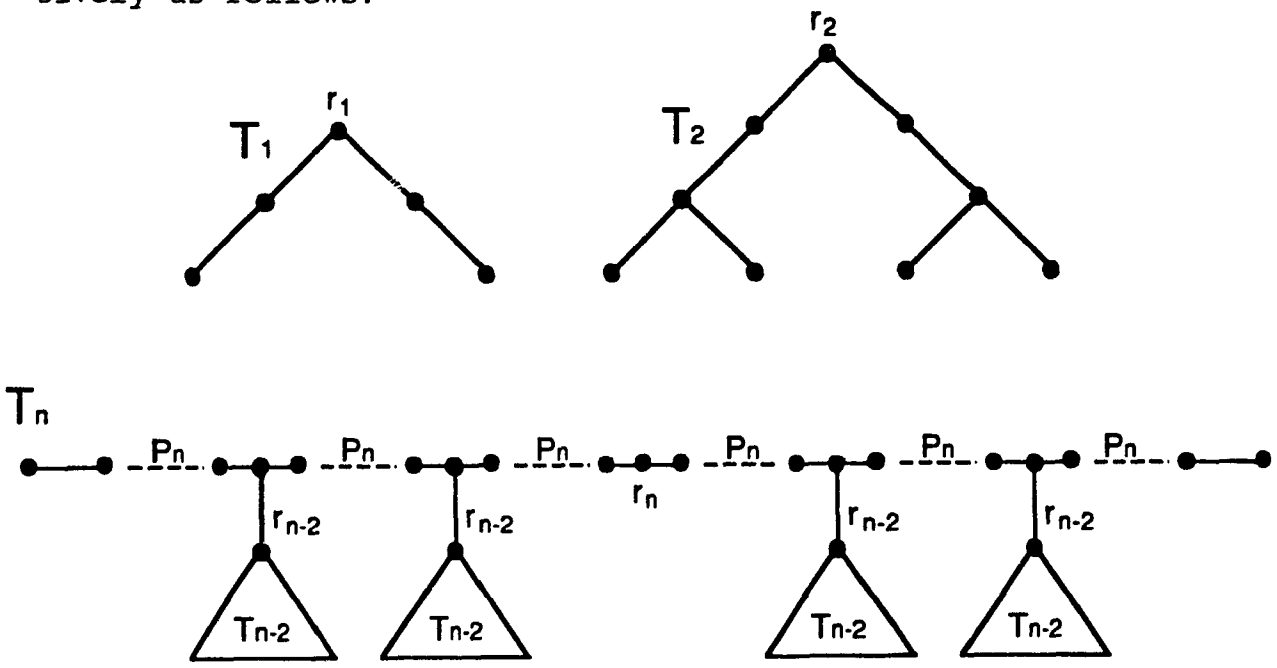


Figure 3.9

where for $n \geq 3$, P_n is a path of length $2*(\text{depth}(T_{n-2}))+4$.

Theorem 3.3: For every $n \geq 1$, algorithm LST constructs a level structure of T_n of width $\lceil (n+1)/2 \rceil$.

Proof: We prove the theorem by induction on n. Clearly, for n=1 and n=2 algorithm LST constructs a level structure T_1 and T_2 of width 1 and 2 resp.

Assume that for $n \leq k$ algorithm LST creates a level structure of T_n of width $\lceil (n+1)/2 \rceil$. Consider now application

of LST to T_k and let N be the level structure generated. Clearly, the path consisting of six P_k paths is the diameter of T_k and algorithm LST will place one vertex of the path in each level of N. Further, recursive application of algorithm LST will produce level structures N^1, N^2, N^3, N^4 for the four copies of T_{k-2}. By induction hypothesis,

$$\text{width}(N^i) \leq \lceil (k-2+1)/2 \rceil = \lceil (k-1)/2 \rceil .$$

Since the length of P_k is greater than $2*\text{depth}(T_{k-2})$, by merging N^1, N^2, N^3, N^4 into N, we obtain a level structure N such that

$$\begin{aligned} \text{Width } (N) &= 1 + \max \{\text{width } (N^i) \mid 1 \leq i \leq 4\} \\ &\leq 1 + \lceil (k-1)/2 \rceil = \lceil (k+1)/2 \rceil . \end{aligned}$$

One can further construct a labelling f such that

$$B_f(T_k) \leq \text{width } (N) = \lceil (k+1)/2 \rceil .$$

Since it can be proved [18] that $B(T_k) \geq \lceil (k+1)/2 \rceil$, it follows that algorithm LST constructs optimum solutions for the family of trees T_n. Now for tree T_{n-2}, there exists i such that there are $2^{n/2}$ vertices at distance i from r_{n-2}. Thus if algorithm GPS is applied to T_n, it will construct a level structure of width greater than or equal to $2^{n/2}+1$. Thus while LST creates a level structure of T_n with width $O(n)$, the level structure created by GPS has width $O(2^{n/2})$ which clearly demonstrates the advantage of algorithm LST

over algorithm GPS for such a family of trees.

Poor Performance of LST:

As is the case with any heuristic algorithm, one can find examples in which the performance of algorithm LST is very poor. In some instances, a decision to choose a merger rule over another, with a view to keep to the minimum the width of the resulting level structure, may prove to result later in increased width. One can find examples of trees in which the width of the level structure produced by algorithm LST exceeds the bandwidth by arbitrarily large numbers. The following theorem makes a general statement in this direction.

> **Theorem 3.5:** For every positive integer $k \geq 3$, there exists a tree G_k such that $B(G_k) = 3$ and the width of the level structure of G_k produced by algorithm LST is $k+1$.

Proof: For each $k \geq 3$ one can construct a tree G_k as shown in Figure 3.10. G_k is a caterpillar. The diameter path in G_k is (u,v). The centre of the diameter is w. P(t) denotes a path of length t. The legs from w towards u are paths $P(2), \ldots, P(2^{k-2}), P(2^{k-1}), P(2^k)$. The first one of these, P(2), is at distance 2 from w. For $t \geq 1$ the part of the diameter between legs $P(2^t)$ and $P(2^{t+1})$ is a path $P(2^t)$ at the centre of which is a leg P(1). The part of the

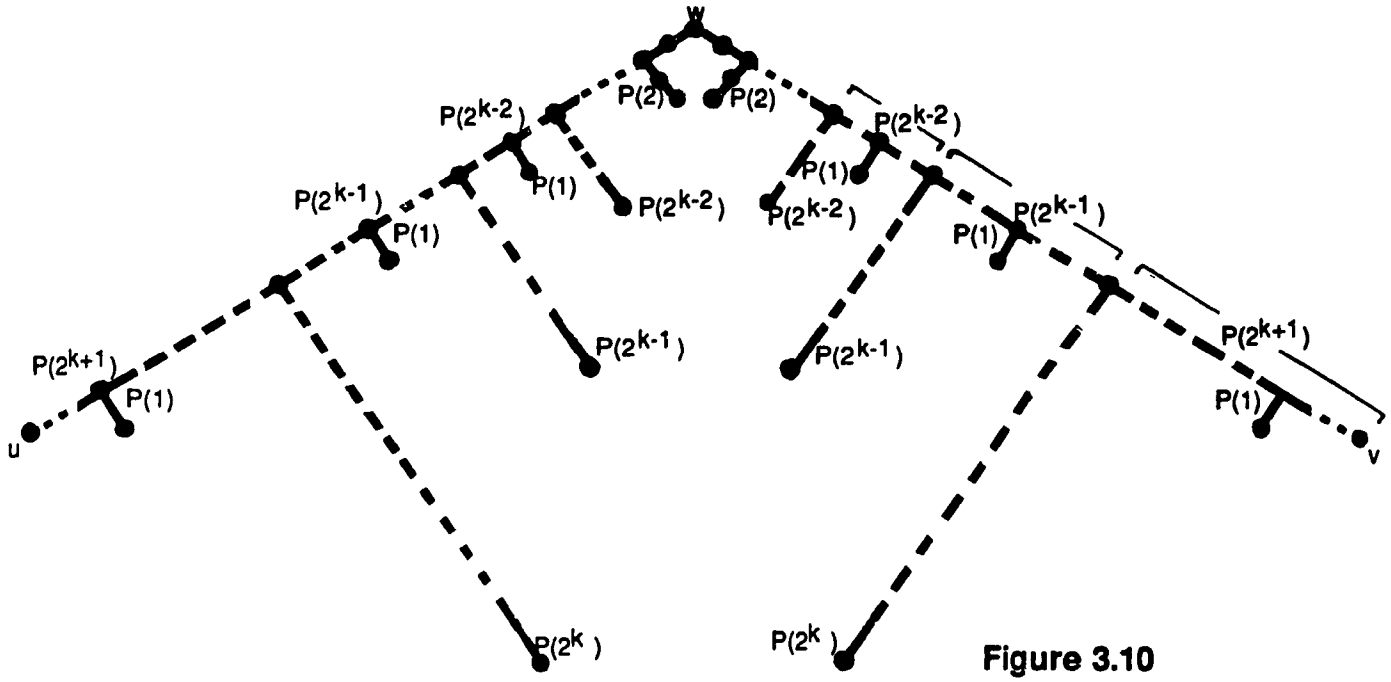


Figure 3.10

diameter between the leg $P(2^k)$ and vertex u is a path $P(2^{k+1})$ on which there is a leg P(1) at distance $3*(2^{k-1})$ from u. The graph is symmetric about w and can be described similarly between w and v. Clearly $B(G_k) = 3$.

Let us first consider the application of algorithm LST1 which processes all subtrees by the same recursive rule. Now algorithm LST1, after finding the diameter (u,v) and merging the first leg P(1) closest to u, will merge the leg $P(2^k)$ by turning it towards w to avoid creating at this point a level structure of width 3. Similarly, after merging the second small leg P(1), the leg $P(2^{k-1})$ will be merged by turning it towards w to avoid creating a level structure of width 4 at this point. Thus all the legs of length greater than 1 are turned towards w resulting in a level structure of width k+1.

If on the other hand each one of these longer legs is turned to the left towards u, and those between w and v towards vertex v, then the resulting level structure has width 3. It is not difficult to see that one can assign a numbering following this level structure, which results in bandwidth 3.

Algorithm LST, which processes all small subtrees at the end, will give a level structure of width 3 in this case. However, by replacing all the P(1) subtrees by P(3) one can construct similar examples in which the performance of LST can be shown to be arbitrarily poor.

We note that algorithm GPS will fare equally poorly in this particular case.

3.6. Modifications to algorithm LST

The performance of algorithm LST can be improved in some instances by applying a "level smoothing operation" after all the vertices are assigned levels. By comparing widths of three consecutive level classes, vertices can be reassigned to neighboring level classes to further redistribute them as uniformly as possible. This could reduce in some cases the width of the level structure generated.

CHAPTER FOUR

IMPLEMENTATION OF LST

4.1. DETAILS OF IMPLEMENTATION.

We implemented algorithm LST in Turbo Pascal on IBM-PC. Several versions of the algorithm were created to study empirically the influence of various alternatives to some parts of the algorithm on the resulting level structure of the graph. Included among them, in particular, are the following.

(1) The original algorithm LST which merges all "small subtrees" right at the end.
(2) The alternative totally recursive algorithm LST1.
(3) Both LST and LST1 with final "level smoothing" operation described in 3.6.

A listing of **Program Level_structure, version 8.0,** the implementation of LST1 (which is (2) above) is included as Appendix 1.

4.2. THE OBJECTIVE OF THE IMPLEMENTATION

The **primary objective** of this implementation of LST in PASCAL language is to demonstrate that for trees the algorithm generates a level structure of much smaller width than does any of the existing algorithms, in particular, the GPS algorithm. As such, the implementation does not claim to be the most efficient in execution time or storage space requirement. This aspect of the implementation of LST will be considered in our future work. Here, an attempt is made to keep the implementation clear and simple and the data structure and the program layout is chosen accordingly.

Many alternative theoretical approaches have been tried in some parts of the algorithm. For example:

(1) In computing a diameter, is it necessary to choose u in L_v for which L_u has minimum width?

(2) In LST, how to merge a subtree when the vertex adjacent to the parent diameter is not processed?

(3) Which one of the six merger rules should be chosen in case of a tie in the maximum width of levels?

This has resulted in the implementation being modified several times as the work on this thesis progressed, as opposed to being completely planned at the outset. We opted for maintaining a satisfactory, working implementation rather than make drastic changes each time. Besides, our main

interest so far has been to compare the different heuristic approaches as to their effect on reducing level widths. We have been running several versions of the algorithm with alternative components for purposes of overall comparison and to optimize a particular version was not considered necessary at this point.

A **secondary objective** regarding efficient storage did evolve during the work. In order to save a level structure of a subtree, it is necessary to store the level of each vertex, the vertices in each level and the width of each level. Since linear arrays were chosen to represent level structures, space requirement was seen to be potentially excessive in case of a large number of vertices and a large depth of recursion. At this point we questioned whether it is possible to implement the algorithm so that the space requirement would be independent of the depth of recursion. The answer was in the affirmative, and will be explained in section 4.3.4.

The algorithm and the driver together constitute **Program Level_structure**, which takes as input a tree and produces as output a level structure.

4.3. Data structures, types, variables and constants.

A brief description of these is necessary to discuss the implementation.

4.3.1. Constants

Maxnodes is an upper bound on the number of vertices in the tree. **Maxdeg** is an upper bound on the maximum degree in the tree.

4.3.2. Types

number. Throughout the program, each one of a large number of variables will hold an integer which stands for a vertex in the tree. Therefore a subrange of integers, between **0** and **maxnodes,** is taken as a basic unit of information. The **type** of this subrange is **number.**

vector. Another **type** of a basic storage unit. A one dimensional array of type **number** and length **maxnodes.**

4.3.3. The Global Variables

Some variables are kept global to keep the parameter list shorter in procedure calls. Since the algorithm and the driver are parts of one PASCAL program, most of the procedures work in the same environment. Therefore the graph, the number of vertices, their degrees and the maximum degree, which are all part of this environment, appear as global variables.

Some output variables like arrays **level** and **levelwidth** are parts of the environment shared by successive recursive calls, each call accessing only some parts of these arrays. These arrays are also kept as global variables.

INPUT variables:

Input variables **n** and **d** hold the number of vertices in the tree and maximum degree in the tree respectively.

graph is an input variable. The tree is represented by an adjacency table in this implementation. This table is a two dimensional array, **graph**, in which the number of rows is equal to **n** and the number of columns is at least as large as **md**. The k^{th} row lists the vertices adjacent to vertex k, the order in which they are listed being irrelevant. The nonzero entries in each row are left justified and the rest of the row is filled with zeroes.

grfdata is a text file which holds the input graph. The first line in this file contains values of **n** and **md**. Lines 2 onwards give the adjacency table in the same format as variable **graph** above.

OUTPUT variables:

level is of type vector. It outputs the level of each vertex as assigned by algorithm LST.

depthv gives the number of levels in the level structure.

levwidth is of type vector. The first depthv locations store the widths of the levels in the level structure generated.

4.3.4. Storage management in recursive calls

LST is a recursive algorithm as described in pseudocode. Its implementation, however, is not completely recursive. It is truly recursive only after the main diameter is computed. Although LST extracts a diameter from the main tree as well as from each one of the subtrees, the subtree processing differs from the main tree processing in two ways.

(1) First, some of the outcome of the main tree processing must be saved which is not the case for the subtrees.

(2) More significantly, the subtrees are not available in the same format as the main tree. This results in a certain loss of recursive elegance.

The solution to the first problem is to save some of the information in global variables when the main diameter structure is computed. The second difficulty is overcome by listing all the vertices in a subtree in a one dimensional array **grf** before the subtree processing begins, and reading the connections between vertices from the adjacency table **graph**.

Intermediate Global Variables:

(1) Variables used to save information about the main diameter.

deg is of type **vector**. stores the degree of each vertex

in the main tree.

depthv, widthv, widev are variables of type **number**. They hold the depth, the width and the size of the last level class resp. of the main diameter level structure. Initially, variables **level** and **levwidth** hold the level of a vertex in this diameter level structure and the width of each level class respectively. As each new subtree is processed and its level structure is merged into the main level structure, these arrays are modified and finally become part of output as listed above.

nodesqv and **templev** are of type **vector**. Let v be the starting vertex of the main diameter. Then **nodesqv** lists all the vertices in the order of their distance from v and **templev[k]** stores the level of vertex k in the rooted level structure L_v.

(2) Many variables used in the processing of subtrees are kept global to avoid a large list of parameters.

grf of type **vector** is one such variable. It is introduced to list all the vertices of a subtree in consecutive addresses, so that they can be easily used in a loop. The adjacency table **graph** is used to read the neighbors of a vertex. Thus the processing of a subtree is somewhat different from that of the main tree.

(3) In addition, the next section lists some global variables used for storing the generated level structures

efficiently.

Efficient Storage of Level Structures.

A diameter level structure of the tree or a subtree must be stored until all its subtrees are processed. Obviously, it would be impractical to save this information locally in each recursive call, because in that case storage requirement will be of the order of the depth of recursion multiplied by the maximum length of an array. Another problem is representing the dynamically growing level structures each having different length, width and lists of vertices which must be merged to create new lists of vertices belonging to the same level class. This creates a necessity to find a suitable format for storing the diameter level structures obtained in successive recursive calls and keeping track of relations among them for the later merge operation. Dynamic storage with pointer variables and linked lists were ruled out as too time consuming.

Terminology. There is a natural partial order on the set **F** of all the **diameter level structures** computed by algorithm LST when applied to a tree T. We introduce the following terminology in keeping with this partial order. First note that for every diameter computed by LST, there is the **associated subtree** of which it is the diameter. If L and L' are in **F** with associated subtrees S and S' resp., then we say that L is **lower** than L'

(and L' is **higher** than L) if S is a subtree of S'. If S is an off-diameter component of S' then we say that L'is **the parent diameter structure** of L. If there is no L in **F** lower than L' then we say that L' is a **minimal** structure. The **highest** structure in **F** is the diameter of T, which is the **main diameter structure.** When a minimal diameter is computed, it is merged into its parent structure and thus **augments** it. When all the lower (possibly augmented) diameter structures are merged into a diameter structure then that in turn gets merged into its (possibly augmented) parent structure. In this implementation every level structure encountered is seen as a **diameter level structure,** being either in its initial diameter form or in its augmented form after absorbing some lower structures. A vertex goes from being part of a level structure to being part of a higher level structure. At any time a vertex **belongs** to at the most one level structure, the highest one in which it has been assigned a level.

For any vertex, all the information regarding the structure to which it currently belongs is stored in global variables and this information is updated whenever its structure is merged into a higher one. This is done as follows.

The array **nodesqv** is used to list all the diameter level structures. Initially, it lists the main diameter level

structure. When a diameter of a subtree is computed, it is listed in nodesqv starting at the next available location as given by **stnode**. The array **levwidth** records the widths of the level classes in parallel to **nodesqv**. The location **levwidth[j]** gives the width of the level class indicated by **nodesqv[j]**. When initially a vertex enters **nodesqv[j]** as part of a diameter, **levwidth[j]** is equal to 1. The variables **stdiam, lendiam, pnter** are global variables of type **vector**. For any vertex k, **stdiam[k]** gives the location in array **nodesqv** of the start of the diameter level structure to which k currently belongs while **lendiam[k]** gives the number of classes in this level structure. The array **pnter** is used to link all the lower diameter structures merged into a structure L so that when L is merged into its parent diameter structure, all the lower level structures also get merged with L. The entries in array **pnter** run parallel to **nodesqv**. The location corresponding to the end of a diameter contains the address of the beginning of the first diameter merged into it. If no such merged diameter exists, then this location contains 0. If one subtree T_2 gets merged into a diameter P after another subtree T_1 has been merged before, then T_2 will be linked to P through the last subtree of T_1 linked to T_1. The array **level** gives the level of a vertex in the level structure to which it currently belongs.

4.4. Procedures.

In this section we briefly describe a few aspects of some main procedures of LST.

Procedure getdata. This procedure obtains interactively from the keyboard particulars about the input. Input is either from a diskfile, in which case the file name is read from the keyboard, or it may be a randomly generated tree. In the latter case the number of vertices **n**, the maximum degree **md** and a number p such that $0<p\leq 1$ are read from the keyboard. Each vertex has **md** neighbors, each with probability **p**. For generating a (connected) tree, **p** should be not too small. A safe choice seems to be **p** such that $(\mathbf{md})*p \geq 3$.

Procedure gentree. Given **n**, **md**, and **p** as above this procedure generates a rooted tree, vertex 1 being the root, and writes the adjacency table in array **graph.** It also computes as byproducts the length and the level widths of the rooted level structure L_1 and prints these. These values may be later compared with the output of algorithm LST.

Procedure rooted_tree. Starting with a vertex v of degree 1, this procedure generates a rooted level structure L_v. The level of each vertex in L_v is recorded in the array **tlevel.** In addition, vertices are listed in the order of their levels in **nodesq.** The procedure also computes the width, depth and the size of the last level of this rooted structure.

Procedure diameter. As explained in chapter 3, starting with a vertex **v** of degree 1, this procedure uses procedure **rooted_tree** and returns in **nodesqv** a diameter path, **nodesqv[k]** being the k^{th} vertex on this diameter. When a vertex becomes part of a diameter, it is called **processed.** The variable **v** returns the starting vertex of the diameter, and **depthv** returns the length of the diameter. On the first call to this procedure, **tlevelv** will save levels of all the vertices in L_v.

Procedure process_branches. After the main diameter structure is computed, this procedure is called. It initializes the arrays **stdiam, lendiam, pnter** before going into the recursive processing of subtrees.

Procedure process_subtree. Given a vertex, this procedure finds the maximal subtree which consists of only non-processed vertices and which contains the given vertex. The vertices in this subtree are entered in array **grf.** Unlike in the case of the main tree, first the size of the subtree is found. If the size is smaller than 3, then **procedure small_subtree** is called to generate a level structure of the tree. If the size is greater than or equal to 3, then the global array **tlevelv** is used to find the starting vertex which is farthest from the starting vertex of the main diameter of T. After this the procedure **diameter** is called and the recursive processing continues.

Procedure merge_levels. Given the vertex k on the parent diameter structure of subtree T', and vertex nbr in T' which is adjacent to k, this procedure merges the level structure of T' into the parent structure. As explained in 3.2, six merger rules are investigated and the one resulting in the minimum width structure is chosen. Arrays stdiam, lndiam are used to locate the level structures. Stdiam[k] gives the starting location in nodesqv of the diameter structure of k. Length[k] gives its length. The array levwidth is used to find the current widths of its level classes. Starting from levwidth[stdiam[k]] the next lndiam[k] locations in array levwidth give the widths of level classes in this structure. Similarly the diameter structure of vertex nbr is retrieved. The widths in the structure of k are updated to account for the increase in the class sizes. The array level is updated by assigning new levels to the vertices on the diameter path of T'. The array pnter is used to trace all the lower structures merged into this diameter path and their levels are updated as well. Further, array pnter is used to link the starting location of the diameter structure of nbr to the last location of the diameter structure of k in array nodesqv.

4.5. FURTHER CONSIDERATIONS IN IMPLEMENTATION.

4.5.1 Priority in the choice of merger rule.

Procedure merge_levels presented some interesting possibilities. If the merger of two level structures using the six feasible merger rules is studied, and if more rules than one result in the minimum width for the merged level structure, which rule should be used? **Procedure min_width_index** takes the six level structure widths and returns the index with the minimum width. As the standard algorithm for finding minimum of an array goes, in case of a tie, the smaller index will be chosen. For the level structure this means that when all the widths are equal, rule 1 will be chosen. Rule 1 tries to push the subtree towards the lower-numbered level classes on the parent diameter. What if we change the algorithm for finding the minimum, and choose the higher index in case of a tie? Or what if we first randomly permute the six numbers before finding their minimum, which is same as applying the algorithm to find the minimum with permutation applied to the six indices of the array? Will such choices reduce the level structure width? These approaches were tried on some data and the conclusion reached was that the first method chosen generally results in lowest structure width. Since the vertices on the diameter are scanned from lowest to highest level numbers for processing subtrees adjacent to them, the heuristic of trying to

push subtrees as much as possible in the low index levels seems to be the best course of action.

4.5.2. LST vs. LST1.

The two versions of the algorithm were studied on many sample data. The widths obtained were comparable in all cases. If at all, algorithm LST posed some problems both in implementation and the results obtained.

(1) Increased width in LST. If there are many "small subtrees", i.e. of size 1 or 2, near the start of the diameter, and if processing of these subt ees is postponed till the end, as is done in LST, then the lower levels in the structure may continue to be of small width. That may result in some subtrees being assigned to the lower levels by using merger rule 1 rather than to higher levels. This can result in an intermediate structure in which levels with low indices have more width than levels with high indices. When finally the small subtrees near the lower levels are processed, they must be assigned to these lower levels. In this case,the resulting level structure may have very large width.

(2) Problem in merging in LST. Let **k** be a vertex on a diameter P and let **nbr**, a vertex adjacent to k, be the root of a off-diameter subtree T_1. While processing the subtree T_1, nbr may not be processed because it may be part of a "small subtree". In this case nbr will have no level assigned to it

when T_1 is to be merged into P. This difficulty is overcome in the version 9.0 of the implementation by writing a **procedure process_root.** This procedure makes sure that the root of a subtree, as nbr above, is processed before its level structure is to be merged into the parent structure. The distance between such an unprocessed root and the set of processed vertices in the subtree is at most 2. Thus it is possible to process the root without too much computation and without processing all the other small subtrees.

(3) Additional procedures in LST. When all subtrees except the small subtrees have been processed, every small subtree is adjacent to exactly one processed vertex, to whose level all the vertices in the subtree are temporarily assigned. Then starting from the lowest level class, vertices are moved one level up or down, if necessary, to make class widths as even as possible. Additional book-keeping is needed to make sure that two adjacent vertices in "small subtrees" do not get assigned to two non-consecutive levels in this process.

4.6. POSSIBLE IMPROVEMENTS IN THE IMPLEMENTATION.

As mentioned at the beginning of this chapter, there is scope to improve this implementation. Among possible modifications are:

(1) The range of array indices used in subtree processing can be shortened to the size of the subtree by making

some minor changes and making all references to a vertex through indirect address calculation via variable grf.

(2) To find the width of the resulting merged structure as well as to update the widths of the resulting structure it is sufficient to work with only as many classes as there are in the smaller of the two structures.

(3) Most important of all, a level of a vertex need not be changed every time its current level structure gets merged into a parent level structure. It is possible to store, along with each diameter level structure, the merger rule as well as the connecting vertices used in the merge operation and not actually execute the merge operation. This way, a level will be assigned to every vertex only once. As seen in theorem 3.1, there may be as many as $\sqrt{n}$ diameter merges in a tree on n vertices.

CHAPTER FIVE

EMPIRICAL EVALUATION

5.1. THE METHOD OF EVALUATION.

The performance of any heuristic algorithm must be evaluated empirically. In this chapter, by performance of algorithm LST we mean its ability to minimize the width of a level structure produced. The empirical study of time efficiency of the algorithm is not within the scope of this work. The results obtained by applying the algorithm to some examples (data) can be studied in two ways.

(1) If the theoretical value of the bandwidth of a tree is known, as is the case with binary trees, stars, caterpillars and some other graphs, one can compare the results obtained with the optimum results. In our case, since algorithm LST computes only a level structure of a tree, to know how good the level structure is we must either apply a known labelling algorithm and compute the bandwidth for the level structure generated or we must resort to a theoretical statement which relates the width of the level structure with the bandwidth of a labelling. A theorem relating the width of a rooted level structure with the bandwidth resulting from a labelling algorithm is found in [15]. However for a general level

structure there is no known result of this nature except theorem 2.1.

(2) In general, the exact bandwidth of a tree is not known. As is the standard practice, one can evaluate the performance of the algorithm by comparing the results with those obtained by some well known algorithms. As we have seen in the preceding chapters, algorithm GPS outperforms all known bandwidth minimization algorithms. Therefore we evaluate the performance of algorithm LST by using the same test data for both algorithms LST and GPS and comparing the results obtained. This method is also instructive in those cases where the theoretical bandwidth of the graph may be known. In this situation, one can study the relative performance of the two algorithms as well as the closeness of the results to the optimum value of the bandwidth. Here again we will compare the widths of the level structures generated by the two algorithms.

We include the two versions LST and LST1 of the algorithm in this comparative study. Although the ACM algorithm 582 is the most time and space efficient version of algorithm GPS, we opted for the simpler version, ACM algorithm 508, which has the same mathematical properties. This was run on the VAX 8500 system of Concordia University. Algorithm LST and LST1 were run on an IBM-PC. The three algorithms LST, LST1 and GPS are applied to all the problems in the data

selected.

5.2. THE DATA

The ability of an algorithm to obtain near optimum solutions must be judged by applying the algorithm to different types of problems. Two groups of problems were chosen in the comparative analysis of the three algorithms.

GROUP 1. Here we have a group of 16 problems, numbered 1 to 16, most of whom are chosen for some special feature. The small size of many of the problems was very helpful in analyzing the nature of the tree, the operation of the algorithm and the intermediate level structures generated. Some of the problems in this group are listed below.

No. 3. The tree in example 3.1.
No. 5. A rooted complete binary tree.
No. 6. A rooted binary tree.
No. 7. A tree where off-diameter subtrees are connected to the main diameter by "small subtrees".
No. 8. Similar to No. 7. However there is only one off-diameter subtree here.
No. 9. The tree in example 3.2.(A conifer)
No.10. The tree in example 3.3.(the optimum result example)
No.11. The complete binary rooted tree on 63 vertices.
No.12. A star.

No.13. A tree with just one off diameter subtree and only a unique choice of diameter.

No.14. A caterpillar.

No.15. A spider.

No.16. A large tree with overlapping spider subtree.

The level widths obtained for these problems are tabulated in Table 1.

Problem No.	|V(T)|	(T)	W(LSTl)	W(LST)	W(GPS)	B(GPS)
1	10	4	2	2	3	3
2	20	3	2	3	3	3
3	30	5	3	4	4	5
4	34	4	7	6	8	8
5	31	3	5	4	5	5
6	25	3	4	4	5	5
7	50	5	7	7	6	7
8	22	4	3	3	5	5
9	36	4	4	5	4	5
10	103	3	3	3	5	5
11	63	3	8	8	9	9
12	9	8	3	4	4	4
13	26	4	3	3	6	6
14	35	6	4	4	5	6
15	15	6	3	3	3	3
16	108	7	7	4	7	7

Table 1.Comparison Of Level Widths in T By The Three Algorithms

|V(T)|: Number of vertices in T. (T): Maximum degree in T

W(LSTl): Level width by LSTl W(LST): Level width by LST

W(GPS): Level width by GPS B(GPS): bandwidth by GPS

The complete level structures produced by all the three programs reveal the special logic of each algorithm and its behaviour in these cases.

GROUP 2. The second group consisted of 15 problems. The problems in this group are trees generated randomly by **procedure gentree.** Given the number of vertices n, the maximum degree d and the probability p of an edge in the tree, this procedure generates a tree as follows. Starting with vertex 1, at every vertex, d edges are created, each with probability p. The endpoint of a new edge is assigned the next vertex number. This process continues until n vertices are created. Various values of n, d and p were arbitrarily chosen to generate these 15 trees. Some problems of large size were included in this group. Problem No. 31 and 32 are complete binary trees which are the exception to the rule of random generation. All the three algorithms were applied to these 15 problems. Procedure gentree, which generates each tree randomly starting with vertex 1 as the root, also computes in each case, as byproducts, the rooted level structure L_1 and level widths of L_1. The results are tabulated in Table 2.

No.	|V(T)|	Δ(T)	p	W(T)	W(LSTl)	W(LST)	W(GPS)	B(GPS)
21	25	4	0.60	9	4	4	4	4
22	25	4	0.70	12	5	4	5	6
23	45	4	0.60	20	6	6	7	7
24	30	5	0.50	17	5	6	6	7
25	50	3	0.65	11	5	5	7	7
26	50	7	0.35	25	8	9	10	10
27	40	3	0.80	15	4	5	5	6
28	40	5	0.50	23	7	7	7	8
29	50	5	0.45	23	5	6	10	10
30	40	6	0.40	18	5	5	7	8
31	127	3	1.00	64	15	13	17	17
32	255	3	1.00	128	24	24	33	33
33	200	4	0.45	59	14	14	18	18
34	300	5	0.40	54	26	29	29	35
35	200	4	0.40	36	12	11	13	17

Table 2. Comparison Of Level Widths of Randomly Generated Trees By The Three Algorithms.

p: probability of an edge Δ(T): maximum degree in T

W(T): Level width of L_1 |V(T)|: No. of vertices in T

W(LSTl): Level width by LSTl W(LST): Level width by LST

W(GPS): Level width by GPS B(GPS): bandwidth by GPS

3. OBSERVATIONS and CONCLUSIONS.

By studying the individual examples some observations regarding the capabilities of the three algorithms can be made. Based on these observations as well as on the procedures and the theoretical basis of the algorithms certain conclusions may be drawn.

1. From the two tables it is clear that LST1 almost always produces level structures of smaller level width than does GPS, and the difference is quite considerable in most cases. The difference in the level widths of LST1 and GPS may not look significant in small size problems however the increase in the level widths from LST1 to GPS range between 13% to 66% in all but 2 cases in Table 1 and between 8% to 100% in all cases but one in Table 2. This fact alone is very likely sufficient to conclude that algorithm LST1 is superior to algorithm GPS for minimizing the bandwidth of a tree. The widths produced by LST are comparable to those by LST1 though in most cases slightly higher. These tables show that algorithm LST1 has a slight edge over LST.

2. If the tree has one dominant off-diameter subtree as in problem NO. 13, GPS seems to produce really poor results. The level structure width can be as much as double that of LST1. This should be expected since the recursive application of LST first distributes the vertices of the subtree in

as many levels as possible whereas GPS uses **the rooted level structure** for the subtree choosing the connecting vertex as the root. Similar results should be expected if there are an odd number of components of T-D rather than just one. GPS has the same disadvantage when the nodes connecting these components to the diameter are sufficiently far apart.

3. GPS tries to overcome this problem, whenever possible, by finding a diameter starting at the vertex with the smallest width for its rooted level structure. This helps reduce the width considerably as in No.9. When there is a unique diameter, this strategy is useless as in problem no. 10 and 13.

4. If the tree is fairly regular and the density evenly spread, GPS and LST come quite close in their output as in No.4 and No.11. The same tendency is also observed in the second group of problems. Here a smaller value of p and larger value of d indicate that there may be variation in the degrees of vertices. In this case the GPS level width is as much as double that of LST/LSTl.

The relationship between the level structure width and the concept of even distribution of density can be expressed more clearly as follows. Let T be a tree satisfying the following two properties:

(1) the length of the diameter is "sufficiently" large

(2) there is a vertex x and a positive integer p such that

$|S| = |\{ y \mid d(x,y) = p \}|$ is much larger than B(G) then algorithm GPS cannot find a good level structure.

5. While assigning levels to a subtree connected to a vertex x on the diagonal, GPS does not assign any vertex from the subtree to the same level as x. This can prevent a possibly even distribution of vertices, especially when there are a large number of small subtrees. A good example is Problem No. 1.

6. In Table 2, the width of the generated rooted structure L_1 is entered for the sake of comparison. Obviously, one cannot expect the width of L_1 to be close to that of LST or GPS. The starting vertex is more likely to be at the centre of the tree and its maximum level width will be close to double the width of the GPS or LST structures, if the tree is somewhat well balanced.

7. One can compare the results for the complete binary tree with the (theoretical) bandwidth of a binary tree. Chung [3] has proved that for a complete (rooted) binary tree T_k with k levels

$$B(T_k) = \lceil (2^{k-1} - 1)/(k-1) \rceil .$$

Problems No. 5, 11, 31, 32 are complete binary trees with k 5, 6, 7, 8 levels (k) respectively. Chung's formula gives their bandwidths as 4, 6, 11, 19 respectively. Our results are not particularly close to the bandwidth, though they are

closer than GPS. The GPS results seem to be getting farther from optimum value as the problem size gets larger.

8. One may ask whether a level structure of smaller width necessarily results in smaller bandwidth after labelling? The answer to this question is no. All the three algorithms find bandwidth level structure for problem No.15, a spider. In problem No.12, algorithm LST tries to spread the vertices of a star in three levels of equal widths. However, an application of the GPS numbering algorithm to this will not result in a bandwidth labelling of the star. For the same problem the GPS algorithm produces a level structure of larger width, yet with GPS numbering this results in a bandwidth labelling of the star.

9. Comparison between LST and LST1

(i) Problem no. 16 shows a smaller level width by LST than by LST1. This is due the different order among the 6 merger rules used in the two versions. This shows that this order is very important and can make a big difference.

(ii) No.28 gave an example of what can happen when a lot of small trees are connected to one vertex on the diameter. In this case that particular level class has a lot of "remaining" vertices. They can only be distributed to the two neighboring level classes. This can result in a lopsided distribution of vertices, assuming that in the first pass an attempt was made to make the levels of uniform widths.

(iii) Only 6 out of 31 problems list a smaller width for LST than LST1. Two of these are binary trees. This result seems to show that the additional processing to redistribute the vertices in the second pass is not worthwhile. On the other hand, smaller levelwidth does not necessarily imply smaller bandwidth.

(iv) There are other problems in trying to redistribute vertices. These are already explained in 4.5.2.

IN CONCLUSION, this empirical investigation shows that algorithms LST and LST1 consistently generate level structures of much smaller width than does algorithm GPS. This fact seems to indicate that LST/LST1 will be more successful in reducing bandwidths than algorithm GPS.

5.4. POSSIBLE EXTENSIONS OF THIS WORK

1. Compare bandwidths with GPS by combining existing labelling algorithms with LST.
2. Find a labelling algorithm suitable for LST.
3. Find a better estimate of the bandwidth of LST labelling.
4. Can bandwidth after LST be smaller than level width?
5. Write a time efficient version and evaluate the time efficiency in comparison with GPS.

REFERENCES

1. Bondy J. A. and U. S. R. Murty. **Graph Theory with Applications.** The Macmillan Press Ltd. (1976).

2. Chinn P. Z., J. Chvátalová, A.K. Dewdney and N. E. Gibbs. "The bandwidth problem for graphs and matrices: a survey". **Journal of Graph Theory** 6 (1982). pp. 223-254.

3. Chung F. R. K. "Some problems and results on labelings of graphs". **Bell Laboratories.** Murray Hill, New Jersey.

4. Crane H. L. Jr., N. E. Gibbs, W. G. Poole Jr. and P. K. Stockmeyer. "Algorithm 508. Matrix bandwidth and profile reduction [FI]". **ACM transactions on Mathematical Software** 2, 4 (1976). pp. 375-377.

5. Cuthill E. H. "Several Strategies for reducing the bandwidth of matrices". **Sparse Matrices and Their Applications.** D. J. Rose and R. A. Willoughby, Eds. Plenum Press. New York. (1972).

6. Cuthill E. and J. McKee. "Reducing the bandwidth of sparse symmetric matrices". **Proceedings of the 24th National Conference of ACM.** (1969). pp. 157-172.

7. Dewdney A. K. "Tree topology and the NP-completeness of tree bandwidth". **Department of Computer Science Research Report** #60, UWO. London. Ontario. (Nov.1980).

8. Fulkerson D. R. and O. A. Gross. "Incidence matrices and interval graphs". **Pacific Journal of Mathematics.** 15(1965). pp. 835-855.

9. Garey M. R., R. L. Graham, D. S. Johnson and D. E. Knuth. "Complexity results for bandwidth minimization". **SIAM Journal of Applied Mathematics.** 34 (1978). pp.477-495.

10. Garey M. R. and D. S. Johnson. **Computers and Intractability: A Guide to the Theory of NP-Completeness.** W. H. Freeman and Co., San Fransisco. (1979).

11. Gibbs N. E. and W. G. Poole Jr. "Tridiagonalization by permutations". **Communications of ACM.** 20 (1974). pp.20-24.

12. Gibbs N. E., W. G. Poole and P. K. Stockmeyer. "An algorithm for reducing the bandwidth and profile of a sparse matrix". **SIAM Journal of Numerical Analysis.** 13 (1976). pp. 235-251.

13. Gibbs N. E., W. G. Poole and P. K. Stockmeyer. "A comparison of several bandwidth and profile reduction algorithms". **ACM Transactions on Mathematical Software.** 2, 4 (1976). pp. 322-330.

14. Harary F. Problem 16 in **Theory of Graphs and its applications.** M. Fiedler, Ed. Czechoslovak Academy of Science. Prague. (1967).

15. Hare E. D., W. R. Hare and S. T. Hedetniemi. "Another upper bound for the bandwidth of trees". **Congressus Numerantium** 50 (1985). pp. 77-83.

16. Lewis J. G. "Implementation of the Gibbs-Poole-Stockmeyer and Gibbs-King algorithms". **ACM Transactions on the Mathematical Software.** 8, 2 (1982). pp.180-189.

17. Lewis J. G. "Algorithm 582. The Gibbs-Poole-Stockmeyer and Gibbs-King algorithms for reordering sparse matrices". **ACM Transactions on the Mathematical Software.** 8, 2 (1982). pp.190-194.

18. Opatrny J. and Z. Miller. "A bandwidth reduction algorithm for trees". **Presented at the 18th Southeastern Conference on Combinatorics, Graph Theory and Computing.** Boca Raton. (1987).

19. Papadimitriou C. H. "The NP-completeness of the bandwidth minimization problem". **Computing.** 16 (1976). pp. 263-270.

20. Saxe J. B. "Dynamic programming algorithms for recognizing small-bandwidth graphs in polynomial time". **SIAM Journal on Algebraic and Discrete Methods.** I (1980). pp.363-369.

21. Turner J. "Probabilistic Analysis of bandwidth minimization algorithms". **Proceedings of the 15th ACM Symposium on Theory of Computing.** (1983). pp.467-486.

----ooooO OOooo----

APPENDIX 1

PROGRAM LISTING

```
 program level_structure;

{ Author : Chandra GowriSankaran }

{This program generates a level structure of a tree.
maxnodes is maximum size of the vertex set.
maxdeg is maximum degree in the tree.
These two constants should be set to cover problem size.}

const maxnodes =320; maxdeg = 5;

type  number = 0 .. maxnodes;
      vector = array [1..maxnodes] of number;
      sixer = array [1..6] of number;

var   probnum, n{no. of vertices}, md{ max. deg.} :number;
      stnode, v, depthv ,widthv, widev:number;
      printout, subtree, testing : boolean;

       graph: array[1..maxnodes, 1..maxdeg] of number;
       deg, level, templev, nodesqv, levwidth: vector;
       grf, stdiam, lendiam, pnter,
           nodeq,levsl,levwidsl:vector;
       tempwids:vector; sixwids: sixer; {in merge}
       processed : array [1..maxnodes] of boolean;
       grfdata: text;

{$I gentree.pas}
{a procedure to generate a random tree}


procedure swap (var u,v:number);
{swaps two numbers}
  var k :number;

  begin {swap}
   k:=u; u:=v; v:=k;
  end {swap};
```

```
procedure getdata;

{asks for details of input. Input may be from a disk file.
The first line of the file gives the number of vertices and
maximum degree in the tree. The following lines correspond
to the adjacency table of the tree. Input may also be  a
randomly generated tree from procedure gentree. The input is
read in global array graph, the adjacency table of the tree}

var  nb,mb,gr :integer;  p:real; printlevs, binary :boolean;
     i,j:number; ans, ans1: string [3]; s:string[20];

begin {getdata}
   writeln (' NOTE GRAPH SIZE  ', maxnodes:4,' X',maxdeg:3);
   WRITE (' GIVE PROBLEM NUMBER.  '); READLN (probnum);
   write (' GENERATE A RANDOM TREE ? Y/N  ' ); readln (ans);
   if upcase (ans ) = #89 then
     begin
{specifications for generating the tree. As a special case,
this procedure will generate a complete binary tree }
     write (' GIVE NUMBER OF  VERTICES  '); readln (n);
     write (' GIVE MAXIMUM DEGREE : '); READLN (md);
     WRITE(' GIVE PROBABILITY. REAL NUMBER :'); READLN (P);
     binary := false;
     write (' GENERATING A BINARY TREE ? Y/N ');
             readln (ans1);
     if upcase (ans1) = #89 then binary := true;
     printlevs := false;
     write(' printing levels required? Y/N '); readln(ans);
     if ( upcase(ans) = #89) then printlevs := true;
     gentree (probnum, n,md,p, printlevs, binary);
     {This procedure also writes the adjacency table on a
  disk file}
     end
   else
       begin
        write (' GIVE DATA FILE NAME: ');READln (s);
        assign (grfdata,s);
        reset (grfdata);
        read (grfdata,nb,mb);
        n:=nb; md := mb;
        for i:= 1 to n do
         for j := 1 to md do
          begin read (grfdata,gr);
              graph[i,j] := gr; end;
       end;
end {getdata};
```

```
procedure printdata;
{writes input on the screen. printing of the results is
optional}

var i,j: number;  ans: string[10]; RANTREE:boolean;

begin {printdata}
  printout := false;
  write ('  PRINTING REQUIRED ? Y/N  '); READLN (ans);
  if upcase(ans) =#89 then printout:=true;
  write ('  IS IT RANDOMLY GENERATED TREE ? Y/N  ');
  READLN (ans);

  if upcase(ans) =#89 then RANTREE:=true;
  if rantree then writeln ('printing generated random tree')
  else  writeln( '  printing data from diskfile.  ');
  writeln ( ' PROBLEM NUMBER', PROBNUM:4,'  number of
       vertices =',n:5 , '  maxdeg =', md :5 );

  for i:= 1 to n do
    begin
     write (i:5,'   ',#124);
     for j := 1 to md do IF graph[i,j]<>0
              then write (graph[i,j]: 4)
       else write (' ':4);
     if((MD <= 7) and (i mod 2 = 1))
       then write ('  ',#124#124) else writeln;
    end;
  writeln;



 if printout then
  BEGIN
    write(' ADJUST PAPER ?'); REPEAT UNTIL keypressed;
    {for the noncooperative printer}
    writeln(lst,  #13#10);
    writeln(lst,' ':8, 'PROBLEM NUMBER ', PROBNUM :4,
         #13#10,' ':8, ' VERSION 8.0: ', '  Economy of
             storage.',  #13#10  ,' ':8, ' smalltrees
              processed recursively.'#13#10  );


   if rantree
      then writeln (lst, ' ':8, ' Randomly generated tree');
      writeln (lst, ' ':8,  ' number of vertices =', n:5 , '
                maxdeg=', md :5 );
      writeln (lst,' ':8, ' THE vertex    ITS NHBRS. ');
```

```
      for i:= 1 to n do
       begin
       if i mod 2 = 1 then write (lst, ' ':8);
       write (lst, i:4,' ',#124);

       for j := 1 to md do
         IF graph[i,j]<>0 then write (lst, graph[i,j]:4)
              else write(lst,' ':4);


       if(( md<= 7) and (i mod 2 = 1))
          then write (lst, ' ',#124#124)
          else begin writeln (lst); if (i mod 2 =1) then
           write (lst,' ':8);end;

       end;
     writeln(lst);

  END;
end {procedure printdata};
```

```
procedure initialize;
{ initializes arrays. levels of all vertices are set to 0.
the diameter structure of each vertex is initialized to zero
entries}

var  k:number; ans:char;

begin {initialize}
 testing:=false;
 write (' TESTING ? Y/N  '); readln( ans);
 if upcase(ans) =#89 then testing:=true;
 for k:= 1 to n do
   begin  level[k]:=0; pnter[k]:=0; processed [k]:= false;
       stdiam[k]:=0; lendiam[k]:=0;  end;
 stnode:=0;   { next available location in the lists of
diameters}
end {initialize};



procedure find_deg;
{ finds degree of each vertex from the array graph. stores
it in array deg.}

var i,j, degree :number;
begin {find_deg}
 for i := 1 to n do
   begin
    degree := 0;
    for  j := 1 to md do
    if (graph[i, j]<>0) then degree:= degree +1;
    deg[i] := degree;
   end;
 writeln ( ' The  number      Its degree');
 for I := 1 to n do
  begin write (I:10,deg[i]:10,' ',#124,' ');
     if i mod 2 =0 then writeln;end;
 writeln;
 if testing then repeat until keypressed;
end {find_deg};
```

```
function min_deg_node ( start,fin :number ):number;
{find a vertex of minimum degree from locations start to fin
of array deg}

var k, mdnode ,mindeg:number;
begin function {min_deg_node}
  mindeg := md;
  for k := start to fin do
   begin
    if deg[k] < mindeg then
     begin  mindeg := deg[k]; mdnode :=k; end
   end;
  min_deg_node := mdnode;
 if testing then writeln ('  Min. deg. node is ', mdnode:5);

end { min_deg_ node};
```

```
procedure rooted_tree (v, num :number;
var width, depth, wide :number; var tlevel,nodesq: vector);
{ drops a tree from v.  finds max. levelwidth, depth and
size of the last class, wide, for the rooted level
structure.  stores assigned levels in tlevel.  nodesq lists
the vertices in the order of their levels. num is size of
the tree. On the first call num=n}

var  k,w, nhbr, next, top, bottom, lev,count :number;

begin {rooted_tree}
 for k:= 1 to n do
 begin tlevel[k]:= 0;nodesq[k]:=0;  end;
 top:= 1; bottom:= 1; width:= 1; depth:= 1; count := 1;
 nodesq[1]:= v; next:= 2; tlevel[v]:=1; lev:= 2;
 if testing then begin writeln
     ( ' Printing Levels of tree rooted at ',v:5 );
 write ( '  level of ', v:5, '  is ', lev-1:5, #124); end;



{enter nodes in a queue in order of their distance from v}
 repeat
  wide :=0; { set width of a new level class to 0 }
  while top <= bottom do  {limits for the preceding level }
    begin
       w:= nodesq[top];
       for k:= 1 to deg[w] do
         begin
           nhbr:=graph [w,k];
           if (tlevel[nhbr] = 0) and (not processed[nhbr])
          then
             begin
               tlevel[nhbr]:= lev; nodesq[next]:= nhbr;
               next:=next+1;
               wide:= wide+1;
               if testing then
                     begin
                       write ( '  level of ', nhb:5, ' is ',
                       lev:5, #124);
                       count:= count+1;
                       if count mod 2 =0 then writeln;  end;
             end ;
         end;
       top := top+1;
    end;
  if wide > width then width :=wide;
        {to find max class width}
  top:= bottom+1; bottom:= next-1; lev:= lev+1;
        {limits for next class}
 until bottom = num; {all vertices seen}
```

```
 depth := lev-1; { number of levels}
 {wide gives the width of the last level and n desq
[num-wide] to nodesq[num] give all the nodes in
  the last level.}
 if testing then
     begin writeln; repeat until keypressed; end;
end; {rooted_tree}


procedure newvert(var v,widthv,depthv,widev:number;
                  var nodesqv,tlevelv : vector;
    u,widthu,depthu,wideu:number; nodesqu,tlevelu:vector);

{replaces vertex v by vertex u, and Lv by Lu}

var i:number;

begin {newvert}
 v:=u;
 widthv:=widthu;
 depthv:=depthu;
 widev:=wideu;
 for i:= 1 to n do
 begin
   nodesqv[i]:=nodesqu[i];
   tlevelv[i]:=tlevelu[i];
 end;
end {newvert};


procedure diameter (var v, nmax, deptnv, widthv, widev:
       number; var  nodesqv, tlevelv, levwidth :vector;
      subtree:boolean);

{starting with vertex v, finds the diameter of the tree.
next four procedures are within the scope of this procedure}

var i, u, w, index, depthu, depthw, widthu, widthw, wideu,
  widew :number; nodesqu, nodesqw, tlevelu, tlevelw
:vector; switched :boolean;

procedure assign_wtov;

{replaces the starting vertex v by w}

begin {assign_wtov}
      switched :=true;
      newvert(v,widthv,depthv,widev, nodesqv,tlevelv,
      w,widthw,depthw,widew, nodesqw,tlevelw);
end {assign_wtov};
```

```
procedure min_width_vert;

{if width of the rooted structure of w is smaller than that
of u then replace u by w, replace the Lu by Lw. Used to
choose a vertex of minimum width structure from the last
level of Lv}

begin {min_width_vert}
   if widthw< widthu
      then  newvert(u,widthu,depthu, wideu,nodesqu,tlevelu,
                    w,widthw,depthw, widew,nodesqw,tlevelw);
   index := index + 1;
end {min_width_vertex};


procedure switchuv;

 var i:number;
{swaps the structures of u and v}

begin {switchuv}
    if testing then writeln (' the switchuv was called',
       ' print the two levels');
    swap (u,v);
    swap (depthu,depthv);
    swap (widthu,widthv);
    swap (wideu,widev);
    for i:= 1 to n do
      begin swap (tlevelu[i],tlevelv[i]);
            swap (nodesqu[i],nodesqv[i]);
        if testing then begin write(i:10, tlevelu[i]:10,
                                    tlevelv[i]:10, #124);
                              if i mod 2 = 0 then writeln;end;
      end;
   if testing then repeat until keypressed;
end {switchuv};


procedure assign_levels (var nodesq:vector);

{ computes the associated level pairs for Lv and Lu, and
locates vertices on the diameter. assigns levels to these.
enters the nodes on the diameter in nodesq in order of
levels. computes levelwidths. }

var i,k,q: number;

begin {assign_levels}
 for i:= 1 to n do begin levwidth[i]:= 0; nodesq[i]:=0; end;
 if testing then writeln (' from assign_levels', ^M^J,
                          ' i   tlevelv    tlevelu    level ');
```

```
 for i:= 1 to n do
   begin
    k:= depthv+1-tlevelu[i];
    if tlevelv[i]=k
        then
           begin processed[i]:=true;
                 nodesq[tlevelv[i]] :=i;
                 level[i] :=tlevelv[i];
            levwidth[tlevelv[i]]:= levwidth [tlevelv[i]] +1;
           end ;
    if testing then begin write (i:5, tlevelv[i]:5,
                    tlevelu[i]:5, level[i]:5,' ',#124, ' ');
                     if i mod 2 =0 then writeln; end;
   end;


 if testing then
    begin writeln;
     for i := 1 to n do
      if nodesq[i]<>0 then writeln (' level', i:5,'vertex',
                     nodesq[i]:5);
     writeln; end;
end {assign_levels};



Begin {diameter}
  rooted_tree( v,nmax,widthv,depthv,widev,tlevelv, nodesqv);
  if testing then
   begin writeln(' from diameter', ' v=',v:4,
        ' width, depth, widev',widthv:5,depthv:5,widev:5);
   repeat until keypressed;
   for i := 1 to nmax do
    begin write ('testsudo',i:5, ' nodeq',nodesqv[i]:5, '
              tlevl', tlevelv[nodesqv[i]]:5,#124);
          if i mod 2 =0 then writeln end;
   writeln; repeat until keypressed;
   end;
  repeat
   u:=0; widthu:= nmax; depthu:=0; wideu:=0;
   switched:= false; index := nmax +1-widev;

   for I := 1 to nmax do
     begin tlevelu[i] := 0; nodesqu[i]:= 0; end;

      while (index<=nmax) and (not switched) do
       begin
        w:=nodesqv[index];
        if testing then
                   writeln (' from psudo-diam','  w=', w:5);
        rooted_tree(w,nmax, widthw,depthw, widew,tlevelw,
                  nodesqw);
```

```
        if testing then writeln(' from diameter', 'w=',w:4,
           'widthw,depthw,widew',widthw, depthw:5,widew:5);
        if depthw > depthv
            then assign_wtov
            else min_width_vert;
      end;
  until not switched;
  if testing then writeln (' beg.of diam.',v:5,
                           '  end of diam.', u:5);
  if widthu< widthv then
   begin
    switchuv;
    if testing then writeln (' beg.of diam.',v:5,
                             '  end of diam.',u:5) end;
    assign_levels (nodesqv);
    if testing then repeat until keypressed;
End; {diameter}



procedure find_size (root :number; var size :number;
                     var graf:vector);
{finds size of the component containing vertex root. writes
the vertices in this component in array graf }

var k,p,next,nbr: number;
    counted: array[1..maxnodes] of boolean;


begin {find_size}
  size:=1;
  for k:= 1 to n do
     begin graf[k]:=0; counted[k]:=false;  end;
  p:= root;  next:=1; graf[1]:=root; counted[p] :=true;


  repeat
    for k := 1 to deg[p] do
     begin
     nbr:= graph[p,k];
     if (not processed[nbr]) and(not counted[nbr]) then
       begin size := size +1; graf[size]:= nbr;
            counted[nbr]:=true; end;
     end;
    next := next+1;  p:= graf[next];
  until p=0;
  if testing then writeln (' subtree with root',root :5,
                           '  has size', size:5);
  if testing then repeat until keypressed;
end {find size};
```

```
procedure find_starting_vertex (num:number; var stv:number;
                                grf: vector);
{for a subtree finds a vertex farthest from the vertex
adjacent to the diameter of the parent structure}

var  stlevel,j,k :number;

begin {find_starting_vertex}
  stv := 0; stlevel:=0;
  for j := 1 to num do
   k:= grf[j];
   if templev[k]>stlevel then
      begin  stlevel:=templev[k]; stv:= k; end;
end {find_starting_vertex);

function maximum (var num :vector; var low,high :number)
              :number;
{finds maximum of array num from low  to high}

var  k,big : number;

begin
 big:=0;
 for k:= low to high do
  if big < num[k] then big:=num[k];
 maximum:=big;
end;


function min_width_index (var wids:sixer): byte;
{from the array of 6 widths, finds the index with minimum
width}

var k,j,minimum:number; index:byte;

begin {min_width_index}
 minimum:= maxnodes;  index:=0;
 if testing then writeln (' FROM MIN_WIDTH_INDEX ');
 for k:=1 to 6 do
   begin
   if wids[k]< minimum then
      begin minimum:=wids[k];index:=k end;
   if testing then
       writeln( ' k=',k:4, ' width[k]=',widS[k]:4);
   end;
 min_width_index :=index;
end {min_width_index};
```

```
procedure merge_levels (i,nbr:number);
{merges level structure of nbr into level structure of i}

var k,j,t,y,newlev, ymax,start,lnbr,x,link:number;
    s:byte; leader:boolean;

begin
 if testing then
  begin
   writeln( '  levels before merger ', ^m^j ,   '  k   level
         stdiam   lendiam  ',' pointer  nodeqv levwidth ' );
   for k:= 1 to n do
   writeln (  k:4, level[k]:5, stdiam[k]:8, lendiam[k]:9,
              nodesqv[k]:9,levwidth[k]:9,pnter[k]:9 );
   repeat until keypressed;
  end;


{find the widths by the 6 merger rules}
 for j:= 1 to 6 do
  begin
    if testing then writeln (' tempwids  ',' j=', j:4,^M^J,
              '     k  ', '   tempwids[k]');
    for k:= 1 to lendiam[i] do
            tempwids[k]:=levwidth[stdiam[i]-1+k];
    t:=1;
    for y:= 1 to lendiam[nbr] do
      begin
        k:= nodesqv[stdiam[nbr]-1+y];
        if j<=3 then
                 newlev:= level[i]-level[nbr]+level[k]+j-2
                else
                 newlev:= level[i]+level[nbr]-level[k]+5-j;
        tempwids[newlev]:= tempwids[newlev]
                          +levwidth[stdiam[nbr]-1+y];
      end;

   sixwids[j]:= maximum(tempwids,t,lendiam[i]);

   if testing then
       begin for k:= 1 to lendiam[i] do
                 writeln ( k:10, tempwids[k]:10);
             repeat until keypressed;
       end;
  end;


   { for j:= 1 to 6 do
     writeln (' sixwids[',j:2,']= ', sixwids[j]:3);}
  {choose the index with the minimum width }
  s:= min_width_index (sixwids);
```

```
  if testing then begin
        writeln (' merging levels of root', nbr:5,
                          ' into levels of root', i:5);
        writeln (' merger done by rule no. ',s:5); end;


  {compute new level numbers of the vertices}
  start:= stdiam[nbr]; ymax:= start+lendiam[nbr]-1;
  lnbr:=level[nbr];
  y:= start; leader:= true;
  repeat
     k:= nodesqv[y];
     if s<=3 then newlev:= level[i]-lnbr+level[k]+s-2
             else newlev:= level[i]+lnbr-level[k]+5-s;
     level[k]:=newlev;

     {update level widths. set up reference to new level
       structure for vertices in the merged level structure}
       if leader then  levwidth[stdiam[i]-1+newlev]:=
              levwidth[stdiam[i]-1+newlev]+levwidth[y];
       stdiam[k]:=stdiam[i]; lendiam[k]:=lendiam[i];
       if testing then writeln(' y=', y:4, ' k=', k:4,
                         '  levelk =', level[k]:4 );
       y:= pnter[y]; if y > ymax then leader:= false;
  until y=0;
  if testing then repeat until keypressed;


 {procedure linking: links all the subtrees of
 merged structure to the parent structure}

  x:= stdiam[i]+lendiam[i]-1;
  link:=pnter[x] ;
  while link <> 0 do
    begin x:=link; link:=pnter[x]; end;
  pnter[x]:= start;


  if testing then
    begin
       writeln( '  levels after merger ');
       for k:= 1 to n do
        begin
         if k mod 10 = 1 then writeln( '  k  level  stdiam
               lendiam',  'nodeqv   levwidth   pointer ');
         writeln (k:4, level[k]:5, stdiam[k]:8,lendiam[k]:9,
                  nodesqv[k]:9, levwidth[k]:9, pnter[k]:9 );
         if (k mod 10=0) or (k=n) then
                          repeat until keypressed;
        end;
    end;
end {merge_levels};
```

```
procedure small_subtree (root1,size1:number;
                                        var depl :number);
{assign level structures to subtrees of size 2 or 1}
var k,j:number;

begin
 if testing then writeln (' from small subtree ','stnode=',
                          stnode:4, ^M^J ' k, grf, level,
                        stdiam, lendiam, nodeq, levwidth');
 depl:=size1;


 for k:= 1 to depl  do
  begin
      j:= depl+1-k;
      nodesqv[stnode+k]:=grf[j];level[grf[j]]:=k;
      levwidth[stnode+k]:=1 ; stdiam[grf[j]]:=stnode+1;
      lendiam[grf[j]]:=depl; processed[grf[j]]:=true;
      if testing then writeln( k:4,grf[j]:5,level[grf[j]]:6,
      stdiam[grf[j]]:7,  lendiam[grf[j]]:7,
      nodesqv[stnode+k]:6,
      levwidth[stnode+k]:7) ;
  end;

 if depl=2 then pnter[stnode+1] := stnode+2;
 if testing then repeat until keypressed;
 stnode :=stnode+depl;
end;
```

```
procedure process_subtree (rootl:number; var sizel,
                           depl :number; subtreel:boolean);
{recursive procedure to find a level structure of a subtree.
rootl is the node connecting the subtree to the parent
structure. sizel returns the size of the subtree, as given
by find_size. depl returns the length of the diameter of the
subtree as given by diameter. subtreel is true if the call
from a subtree,false if the call is from the main tree.}

var    i,j,k,nbr,stv,widl,widel,size,dep,rtlevel:number;

begin

  find_size (rootl,sizel,grf);
  if sizel<3 then small_subtree (rootl,sizel,depl)
             else
   begin
     find_starting_vertex (sizel,stv,grf);
     diameter (stv, sizel, depl, widl, widel,
                    nodeq, levsl, levwidsl, true);
     for j:= 1 to depl do
     begin
        {enter the diameter structure in the list of
         diameters in array nodesqv. set up reference
          arrays}
      nodesqv [j+ stnode]:= nodeq[j];
      pnter [j+ stnode]:=j+stnode+1;
      levwidth [j+stnode]:=1;
      stdiam[nodeq[j]]:=stnode+1; lendiam[nodeq[j]]:=depl;
     end;
     stnode := stnode+depl;
     pnter[stnode]:=0;

     for  k:= 1 to depl do
      {find all subtrees adjacent to vertices on the
       diameter}
      begin
        i:= nodesqv[stdiam[nodeq[1]]-1+k] ;
        for j:= 1 to deg[i] do
          begin
            nbr := graph[i,j];
            if not processed[nbr] then
               begin  process_subtree (nbr, size,dep,true);
                      merge_levels (i,nbr); end;
          end;
      end;

   end;
end;
```

```
procedure process_branches (depthv:number;
                            var nodeq: vector);
{called from the main program. processes the subtrees of the
main diameter and merges its level structure into the main
level structure. nodeq holds the main diameter of length
depthv.}

var i,j,k,y,nbr,size,q,dep:number;

begin {process_branches}
 for i := 1 to depthv do
        begin  stdiam[nodeq[i]]:=1;
          lendiam[nodeq[i]]:=depthv;  pnter[i]:=i+1; end;
 pnter[depthv] := 0;
 stnode :=stnode +depthv;

 for  y:= 1 to  depthv do
   begin
     i:= nodeq[y];
     for j:= 1 to deg[i] do
       begin
         nbr := graph[i,j];
         if not processed[nbr] then
             begin  process_subtree (nbr, size,dep,false);
                    if testing then writeln (' nbr', nbr:5,
                                             ' size', size:5,
                                             'dep', dep:5);
                     merge_levels (i,nbr);
             end;
       end;
     if testing then repeat until keypressed;
   end;
end {process_branches};



procedure print_results (var depthv:number);
{prints the levels of all the vertices and all the level
widths
of the level structure generated}

var  lev,i:number;

begin {print_results}
 clrscr;
 writeln (' THE LEVEL STRUCTURE : version 8.0');
 writeln ('       vertex.     level');
  for i := 1 to n do
   begin write ( i:14,level[i]:10);
   if i mod 2 = 1 then write( '     ', #124) else writeln;
end;    writeln;
```

```
REPEAT UNTIL KEYPRESSED;
writeln (' THE LEVEL_WIDTHS ');
for lev:= 1 to depthv do
   writeln(' level',lev:5, 'level width ',
                          levwidth[lev]:5);
repeat until keypressed;

 if printout then
 begin write ('  ADJUST PAPER? '); REPEAT UNTIL KEYPRESSED ;
       writeln (lst, #13#10, ' ':8, ' THE LEVEL STRUCTURE ',
                                          #13#10);
       writeln (lst, '      vertex.    level',' ':5,
                     '      vertex.    level');

       for i:= 1 to n do
         begin write(lst,i:10,level[i]:10);
             if i mod 2 = 1 then write (lst,'    ', #124)
                          else writeln(lst) end;

       writeln(LST);
       write ('  ADJUST PAPER ? '); repeat until keypressed;
       writeln (lst, ' ':8, ' THE LEVEL_WIDTHS ');
       writeln (lst, ' ':8, ' _________________');
       writeln(lst,' ':8, ' level', ' ':8,  '
                                    level width');
       for lev:= 1 to depthv do
          writeln(lst, ' ':8, lev:5, ' ':14,
                       levwidth[lev]:5);
  end;
end {print_results};



begin {program Level_ structure}
  getdata;
  printdata;
  initialize;
  find_deg;
  v:= min_deg_node(1,n);
  diameter (v,n, depthv,widthv, widev, nodesqv, templev,
                          levwidth, false);
 .process_branches (depthv,nodesqv);
  print_results (depthv);
end {program level_structure}.
```

```
Procedure Gentree ( probnum, n, m :number; p: real;
                    printlevs,binary:boolean);
{Generates a random tree on n vertices. max. degree is m.
prob. of each new branch is p. take mp>2 for a connected
tree. var. graph is an nXm array in the calling program}

var   i,j,t,r,k,col,l,u,v :number;
      lev:array [1..maxnodes]  of number;
      s:string[10];

begin

     {initialize}
   for i:= 1 to n do
    lev[i]:= 0; {rooted level structure}
   for i:= 1 to n do
    for j:= 1 to  m do
    graph[i,j]:=0; {adjacency table}

    t:=1; k:=0; l:=1; u:=1; lev [1]:= 1; v:=1;

   [ t is no. of vertices in the graph so far. k is the
vertex getting branches. w = 0 or 1 indicates whether or not
first vertex. v is the current level in rooted structure. lev
gives width of each level. l and u give lower and upper
bounds for the vertices in the  current level. }

while ((t<n) and (k < t ))  do
 begin
  k:=k+1;
  if k = 1 then col:=1 else col := 2;
  if (not binary and (k =1)) then r:=m  else r:= m-1 ;
  {first time m branches. later m-1 branches.
   except when binary tree.}
    repeat
         r := r - 1;
         if ( random <= p )  then
            begin
               t:=t+1;    {one more vertex}
               graph [k,col] := t; {show connections}
               graph [t,l] := k;
               col := col+1 ;
            end;
    until ((r = 0) or (t = n));

  if ((k = u) or (t = n))
    then BEGIN v := v+1; lev[v]:= t-u;  u:= t; END;
end;
```

```
writeln (' At the end ', 'vertices generated = ' , t:3,
          ' last vertex to get branches = ', k:3);
if (k>t) then writeln (' disconnected forest ? ')

    else {print level widths }
     begin
        writeln ( '   THE LEVEL WIDTHS IN THE' ,
                  ' GENERATED ROOTED LEVEL STRUCTURE' );
        for i := 1  to v do
         writeln ( ' LEVEL = ', i:3, ' level width =',
             lev[i]:3);
        if printlevs then
        begin
         write(' ADJUST PAPER ? '); repeat until keypressed;
         writeln (lst, #13#10);
         writeln (lst, #13#10, ' ':8, 'PROBLEM NUMBER ',
                       PROBNUM:4);
         writeln (lst, #13#10 , ' ':8, 'RANDOMLY GENERATED
                                          TREE');
         WRITELN (lst, ' ':8, 'vertices ', n,
                    '    max. deg.=', m:4, '    prob.=', p:4:2);
         writeln (lst, #13#10, ' ':8, ' THE LEVEL WIDTHS IN
            THE' , ' GENERATED ROOTED LEVEL STRUCTURE' );
         for i := 1  to v do
           writeln (lst, ' ':8,  ' LEVEL = ', i:3,
                    '    level width =', lev[i]:3);
         WRITELN (lst, #13#12);
        end;
     end;


{write the generated tree on a disk file to be used by GPS
  and other versions of LST.}
{grfdata is global var. type text in the calling program}
write (' GIVE DATA OUTPUT FILE NAME.', #13#10,
       ' UPTO 6 CHARS. PLUS ENDING WITH .dat '); READLN (s);
assign (grfdata, s);
rewrite (grfdata);
writeln (grfdata, n:5,m:5);
for i := 1 to n do
  begin
   for j := 1 to m do
    write (grfdata, graph[i,j]:5 );
    writeln(grfdata);
  end;
close (grfdata);
end {gentree};
```