AN ANALYSIS OF THE STRUCTURE OF TREES AND GRAPHS

A Thesis

submitted to the

University of Newcastle upon Tyne

for the degree of

Doctor of Philosophy

C. R. Snow

July 1973

Acknowledgements

The author wishes to record his thanks to all his friends and colleagues on the staff of the Newcastle University Computing Laboratory for their help and encouragement during the preparation of this thesis, and particularly to Professors E. S. Page and B. Randell.

I am deeply indebted to Dr. H. I. Scoins for his interest and helpful supervision throughout the period of research, and for his critical comments on the various manuscripts of this thesis.

Thanks are also due to Miss E. D. Barraclough and the N.U.M.A.C. staff for the provision of computing facilities, and especially with regard to the production of this thesis.

During the period of research, I was funded from a grant from the Science Research Council, and later I was employed by the University of Newcastle upon Tyne.

Abstract

This thesis is concerned with the structure of trees and linear graphs. In particular an attempt is made to relate the structure of these objects to the known methods for counting them.

Although the work described here is essentially not computer oriented, the generation and decomposition of graphs and trees by computer is in the background, and so a short section on the computer representation of the various objects is included.

Trees are analysed bearing in mind the counting methods due first to Cayley, and a later method using Polya's classical theorem of enumerative combinatorial analysis. Various methods of representation and generation of trees are presented and compared.

This thesis then goes on to the substantially more difficult problem of analysing graphs using similar techniques, and attempts are made to relate the structure of graphs to the known techniques for enumerating graphs. This involves a more detailed study of Polya's theorem and an investigation into the underlying concepts such as permutation groups as they are applicable to the case under scrutiny. Representations are developed to aid these investigations.

In the following section of the work, methods are investigated for the decomposition of a linear graph, and a number of different decompositions or factorisations are looked at. One such factorisation considered in some detail is the problem of extracting a spanning tree from a graph, and the ways in which the remaining graph or co-tree graph may be manipulated. The complete decomposition of a graph into trees may be achieved using these methods, and the concept of the structure tree of the decomposition is introduced and its properties explored.

The techniques described have all been implemented, and a discussion of the problems of the implementation together with some estimates of timing requirements is also included.

Contents

I Introduction And Definitions.

The subject of Graph Theory falls naturally into three subdivisions. We are discussing here the subject of Graph Theory in a "pure" sense, i.e. we are disregarding the applications to which Graph Theory is a useful aid, in which case the subject becomes very much more diverse than just the three subdivisions.

The first subdivision is to do simply with the ability to prove (or not prove) theorems about graphs. This part of the subject is very closely linked to abstract algebra and the parallels between Graph Theory and Discrete Mathematics may easily be seen. Secondly, Graph Theory has proved to be a fruitful field for the "enumerators". There appears to be an inexhaustible supply of different types of graph, and these can all be examined with a view to counting them, and a large number have in fact been enumerated. Finally, there are many problems associated with graphs in which it is required to find an "efficient" method of deciding whether or not a graph possesses a certain property, or to find some particular property or subgraph of a graph.

The first subdivision, then, is the province of the pure mathematician and, more particularly, of the algebraist. The theorems which are proved or disproved are largely of the existence type, or proving equivalence between properties and so on. Occasionally the proof of such an existence theorem contains a construction of the required property or subgraph, and, even more occasionally, such a construction may be 'efficient' in some sense.

Secondly, we have a large class of enumeration problems some of which have been solved, and some of which have not. This area, largely of interest to combinatorial analysts, has made some use of the algorithmic type procedures, but only to compute the number of graphs of the various types. The third area is studied largely by Computer Scientists. These are the people to whom efficiency is of prime concern, and to whom a mere wave of the hand and the remark "there exists ..." is insufficient. It is clear that the last class of problems can make extensive use of much of the first area, proving theorems to enable more efficient algorithms to be developed.

In this thesis we embrace a little of all three, although we are largely interested in the second category. We are not content, however, unlike the combinatorial analyst, simply to discover how many there are of any particular species of graph, but also to ask: How can we produce them all? We examine here the various combinatorial techniques which have been developed to count graphs, and try to adapt them so that they illustrate better the way in which they correspond to the actual objects which they are counting. In some cases (see Page 1971) there is a method by which this can be done, particularly if there is a reasonably amenable recurrence relation associated with the objects concerned, but when techniques such as Polya's theorem (Polya 1937, de Bruin 1964, Liu 1968) are used to count the objects it is not at all clear how this problem should be attacked.

Throughout this work, then, we are concerned with an examination of the structure of countable objects, and in particular with trees and linear graphs, and attempting to relate the structural properties of these entities to their combinatorial properties, with the hope that we may be able to develop general methods for the representation and manipulation of any objects which are enumerable by current combinatorial techniques.

The leading work in the field of graphical enumeration has been done by Harary in collaboration with a number of others and a list of some of the solved and unsolved problems in this area are given in Harary (1960), Harary (1964), etc. The main part of Harary's work which is used in this thesis is his exposition of the method of counting graphs (Harary 1955). Other workers in the same line are Nash-Williams and Tutte. A comprehensive bibliography of the literature of enumerative graph theory is also given by Turner(1969). With regard to the Computer Science interest in Graph Theory, a paper by Read (1969) gives an introductory survey of the types of algorithms that are being developed. The areas of interest here include Shortest Path Algorithms (see also Pohl 1969), Elementary Cycle Algorithms (Gotlieb and Corneil 1967, Paton 1969), Clique finding Algorithms (Augustson and Minker 1970, Mulligan 1972, Mulligan and Corneil 1972), and perhaps the classic graph problem, the Graph Isomorphism problem. More recently, a number of uses have been found for graph theory to describe certain situations which arise in computer science, notable in Assignment problems, Transportation problems, and also in the theory of programming, program correctness, compiler optimisation and many other applications. The tree also has long been an important tool in the syntactic analysis of programming languages and in many other types of data representation problem (see Knuth 1968).

The present work contains a mixture of the two approaches of enumeration and algorithm production. One of the more interesting problems in computational graph theory is the problem of graph isomorphism. This was the subject of a thesis by Corneil (1968), in which the problem of an efficient algorithm for isomorphism of graphs was solved for a restricted set of graphs. However, the point is made that the smallest known graph outside this restricted set has 26 nodes. We make use of Corneil's algorithms in our later work, but since the size of problem approached becomes unmanageable for graphs with considerably less than 26 nodes, we may employ Corneil's techniques in their simplest form with some confidence.

In this work we also discuss trees, particularly in chapters II and III, where we try to use the known combinatorial properties of trees to generate all of the trees of a certain size in some convenient order. Knuth (1968) devotes a considerable portion of the first volume of his book to the study of trees and their application to certain aspects of computer science, such as data structures, parse trees for compilers, sorting and searching and many other applications. Other authors have studied the isomorphism problem for trees (Snow 1966, Meetham 1968).

With regard to combinatorial problems associated with trees, the first approaches to the problem seem to have made by Cayley (1889) who used an empirical method to obtain expressions for counting sequences for trees. Cayley's results were confirmed by a different method due to Harary and Prins (1959) in which they made use of a very powerful combinatorial tool due to Polya (1937). In chapter III we examine both methods of counting, but it transpires that the tree indexing problem is much more amenable to treatment using the Cayley method than by the Harary and Prins method. Unfortunately, the only known solution to the graph counting problem is by Harary (1955) and this makes use of Polya's theorem, which makes the graph indexing problem correspondingly more complicated. Since the publication by Polya of his famous theorem, a large number of combinatorial counting problems in graph theory (and elsewhere) have now been solved which without the theorem seemed quite insoluble.

Harary (1967) gives a list of a number of unsolved problems in graphical enumeration, which he describes as UGEP III (revised from UGEP II (Harary 1964) and UGEP I (Harary (1960)).

In chapters II and III we consider trees, their representation and manipulation, and the ordering of trees using the counting methods of Cayley. In chapter IV we describe the work of Corneil on the graph isomorphism problem insofar as it is relevant to the present work. Chapter IV also contains some further work beyond that of Corneil which we will make use of in the following chapter. Chapter V is concerned with our attempts to use the combinatorial methods developed by Harary and others to find a method of indexing the set of non-isomorphic graphs, one of which is called the Sieve algorithm, and extends the automorphism partitioning algorithm of Corneil. Chapter VI contains a description of some methods for finding decompositions of a graph, and in particular, some attempts at extracting a spanning tree of a graph which is "best" in some labelling-independent sense. The extraction of a spanning tree is in fact a special case of the factorisation of a graph, and chapter VI also contains some brief considerations of other factorisation problems. The final chapter attempts to summarise the work of the whole thesis, together with a discussion of the practical aspects concerning some computer programs written to implement the techniques discussed in the earlier chapters.

We conclude this introduction by defining more formally the terms which will be used (we hope consistently) throughout this thesis.

A graph is defined to be a set V of objects, known as points, nodes or vertices, together with a set E of edges or lines. The set E consists of pairs of elements from V. A graph is directed if the members of E are ordered pairs, and undirected otherwise. Conventionally, a line of a directed graph is known as an arc. A pair $(v,v) \in E$, where $v \in V$, is known as a loop.

In an undirected graph, two nodes $x,y \in V$ are said to be adjacent if there is a line (x,y) in E, and if $L = (x,y) \in E$, the line L is said to incident with x and with y, and x and y are called the end-points of the line L. The number of nodes to which a node x is adjacent is called the _degree_ of x. A graph whose nodes are all of the same degree is said to be _regular_.

A _path_ in an undirected graph is a sequence of lines (v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k) where each pair of adjacent lines has one end-point in common. A _directed path_ is a path in a directed graph, in which each line is directed, and the starting node of each line is also the finishing node of the preceding line. The length of a path is the number of lines in the path, and a path is said to pass through a node x if x is an end-point of at least two of the lines in the path. A path is uniquely defined by an ordered list of the nodes through which it passes. A path is said to be _simple_ if each line in the path appears only once in the path, and _elementary_ if each node is encountered only once. A _cycle_ is an undirected path in which the starting point and the finishing point coincide, and if the path is simple or elementary, then so is the cycle (except that the first node coincides with the last). A directed cycle is generally referred to as a _circuit_.

An undirected graph is said to be _connected_ if for every pair of nodes in the graph there is at least one path joining them.

In the case of directed graphs, we have several definitions of connectedness. A graph is _weakly connected_ if it is connected when considered as an undirected graph. A graph is _unilaterally connected_ if for every pair of nodes α and β, there exists a (directed) path from α to β, or from β to α. A directed graph is said to be _strongly connected_ if there is a path from any node to any other node. A graph is disconnected if the condition for weak connectedness is not satisfied. For a further discussion of directed graphs, see Harary, Norman and Cartwright (1965).

A particularly important special case of an undirected graph is a tree. A tree is defined as a connected graph which possesses no cycles. Berge (1958) shows that this definition is equivalent to a number of other properties by means of the following theorem:

Theorem.

Let G be a graph of order $|V| = n > 1$. Then any of the following properties characterises a tree:

(i) G is connected and possesses no cycles.

(ii) G has no cycles and has n-1 lines

(iii) G is connected and has n-1 lines.

(iv) G contains no cycles, and if an edge is added which joins two non-adjacent nodes, one (and only one) cycle is thereby formed.

(v) G is connected, but loses this property if any edge is deleted.

(vi) Every pair of nodes is joined by one and only one path.

We leave the reader to refer to Berge's book (chapter 16) for the proof of this theorem. The properties given above are shown by the theorem to be equivalent, and therefore any one of them may be considered as the definition of a tree.

We define a subgraph H of a graph G to be a subset U of the set of nodes V, together with all those lines of G whose end-points lie in the set U. We may define a relation P between nodes of a graph G such that xPy holds if and only if there exists a path in G joining the nodes x and y. In the context of undirected graphs this relation P can easily be shown to be an equivalence relation, and the nodes of G are divided by P into equivalence classes, where two nodes are in the same equivalence class if and only if the relation P holds between them. The graph G is now partitioned into subgraphs, where each subgraph is defined by an equivalence class, and each of these subgraphs is connected. Furthermore, there are no larger subgraphs which are still connected. These subgraphs are known as the <u>connected components</u> of G.

A particular case of a non-directed graph is a forest. A <u>forest</u> is any graph which has no cycles. By the definition of a tree, we see that each connected component of a forest is a tree. An analogous theorem to Berge's theorem can now be proved for forests.

<u>Theorem.</u>

Let G be a graph of order |V| = n > 1. Then any of the following properties charactarises a forest:

(i) G has no cycles.

(ii) G has p connected components and n-p lines.

(iii) G is such that if a line is added which joins two non-adjacent nodes, either

(a) one and only one cycle is thereby formed,

or

(b) the number of connected components is reduced by one.

(iv) If any line is deleted, the number of connected components is increased by one.

(v) Every pair of nodes is joined by at most one path.

Proof:

Property (i) is simply the definition of a forest.

(i) ==> (ii): As already observed, the connected components of a forest are trees. Let the i-th connected component contain n_i nodes. Then we have:

$$\sum_{i=1}^{p} n_i = n$$

Also, since the i-th component is a tree it contains $n_i - 1$ lines. Thus the whole graph contains

$$\sum_{i=1}^{p} (n_i - 1) = \sum_{i=1}^{p} n_i - p = n - p \text{ lines.}$$

(ii) ==> (iii): By the theorem stated earlier, no connected graph can have less than n-1 lines and in this case it contains no cycles (i.e. it is a tree). If each connected component contains n_i nodes, it must have at least n_i-1 lines. But since the total number of lines is n-p, each component must have exactly n_i-1 lines and therefore G contains no cycles since each component is a tree.

Now if we add one more line to G, either this line joins two nodes in the same component, in which case G now has exactly one cycle, since by the previous theorem the component which gains the line now contains exactly one cycle, and all the other components are unchanged, or else the inserted line joins two nodes not in the same component. In this case, a path now exists which joins a node in one component to a node in another component and so these two components become a single connected component in the new graph, so that the number of components is reduced by one. This new line cannot form a cycle since if this were so, its two endpoints would be joined by a path not including the new line contradicting the assumption that the two points were in different components.

(iii) ==> (iv): From the proof of the previous theorem, we have the fact that if any line is deleted from a tree, then the number of components is increased from one to two. Since each connected component of a forest is a tree, the deletion of one line increases the number of components in that subgraph of the forest containing the endpoints of the line from 1 to 2. The remaining p-1 components are unaffected. Thus the total number of components after deletion of the line is p+1.

(iv) ==> (v): Suppose there were two nodes having two distinct paths joining them. Then it would be possible to delete a line in one of the paths so that the number of components remains the same. Thus every pair of points must be joined by at most one path.

(v) ==> (i): If there was a cycle in the graph, there would be two distinct paths between any pair of nodes in the cycle, thus there can be no cycles in the graph.

A _partial graph_ Q of a graph G has the same vertex set as G, but possesses only a subset of the lines of G. A partial graph G is said to _span_ G. A particularly important partial graph of a connected graph G is the _spanning tree_. This is simply any partial graph of G which is also a tree. The analogous concept for a disconnected graph is the spanning forest. A partial subgraph is a partial graph of a subgraph.

We will also require some general definitions relating to trees. A tree will sometimes be referred to as a free tree to emphasise the fact that it has no special properties. A rooted tree is a tree in which one (and only one) node has been singled out as being a reference point for the tree. This node is called the root of the tree. Rooted trees will be shown in the diagrams in this thesis as having their root at the bottom of the picture with all other nodes above the root. It now appears to be common practice to draw trees the opposite way with the root at the top of the diagram. There are two schools of thought; one which says that trees should look similar to natural trees, which have their roots in the ground, and another which considers trees as being generalisations of such objects as family trees. In this work we subscribe to the former view, and our terminology (below, up left, etc.) reflects this.

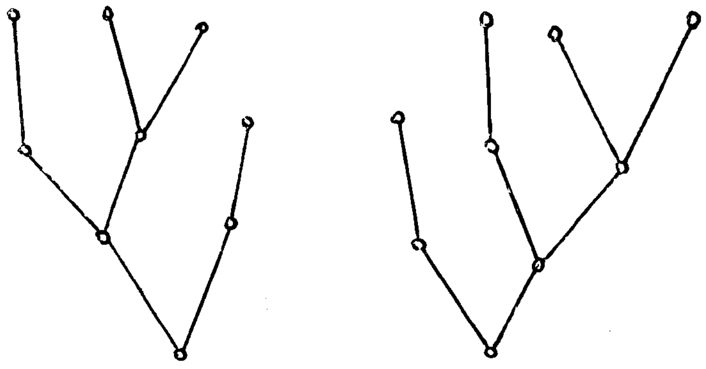


Fig. 1.

The act of drawing a tree on paper immediately imposes an ordering on the nodes, i.e. a relationship of "right" and "left" on the points in the diagram. An ordered rooted tree takes account of these spatial relationships between nodes. In fig. 1. we see two trees, both rooted, which are different when considered as ordered rooted trees, but equivalent when considered as rooted trees, since only the interconnection of the nodes and the position of the root are significant.

In a rooted tree, we define the height of a node x to be the length of the path from the node x to the root. The height of a tree is the maximum height of any point in the tree. In any tree, a node which is only adjacent to one other node in the tree is called a terminal node (or leaf), all other nodes being non-terminal nodes. A tree whose root is a terminal node is called a planted tree.

Two other special cases of trees are also of interest. We define a (strictly) binary tree to be a (rooted) tree in which every non-terminal node has exactly two nodes above it. If the root is a terminal node then the tree consists of just this node. Another way of expressing this is to say that either the root has degree zero, or the root has degree 2 and all the other non-terminal nodes have degree 3. A property of these binary trees is that the number of terminal nodes exceeds the number of non-terminals by one.

This can be shown as follows:

Let a binary tree with n nodes have t terminal nodes. Now each node except the root has one line below it. Thus the number of lines in the tree is n-1 (which we knew anyway from the definition of a tree). But each non-terminal node has two lines above it. Thus the number of lines is also given by 2(n-t). Hence

$$n - 1 = 2(n - t) \text{ or } t = (n + 1)/2$$

and the number of non-terminals is:

$$n - (n + 1)/2 = (n - 1)/2 = t - 1.$$

Knuth (1968) in his book makes extensive use of a tree which we here call a bifurcating tree. A _bifurcating tree_ is a tree in which the number of nodes above any node must not be greater than two. This type of tree has a number of applications in the area of sorting and searching.

The definitions given here are those which will be referred to more commonly in later chapters. In addition, further definitions which pertain to later work are given when required.

II Computer Representation.

It is clear that throughout this work on the manipulation of trees and linear graphs within a computer, some method must be used to identify the nodes of the graph. Thus in all our work whether we be dealing with labelled or unlabelled graphs we have to attach a labelling to any graph for the purposes of computer representation. The problem then becomes one of trying to carry out manipulations in a manner which is independent of any labelling we may impose. The other alternative is to reduce any graph to some canonical form, and then we can with confidence use the labelling which is imposed by the canonical form.

In some previous work (Snow 1966) we showed some ways of representing rooted trees and free trees and then using them to ascertain whether two free trees were topologically equivalent. We now describe this work with some extensions.

II.1 Ordered Rooted Trees.

The representation of ordered rooted trees requires that some notion of "next to" between nodes must be carried in the representation. We first make some definitions which are applicable to ordered rooted trees. A package is a node x together with all those nodes y such that x is the second node (y being the first) in the path from y to the root. The node x is said to be the package head of the package. Thus, in fig. 1., the nodes x and y_i $i=1,\ldots,k$ make up the package whose head is x. The nodes y_i are considered to be ordered within the package, and we define the positive neighbour of a node y to be that node which is encountered next after leaving y as the nodes in the package are traversed in a clockwise sense about the package head. In fig. 1., the node y_{i+1} is the positive neighbour of the node y_i for $i=1,\ldots,k-1$. In this case there is no positive neighbour for y_k. We define a node p to be the negative neighbour of the node q if and only if q is the positive neighbour of p. The up left of a node x is defined to be that node in the package whose head is x (sometimes referred to as the package on x) which has no negative neighbour. Again referring to fig. 1., y_i is the negative neighbour of y_{i+1} for $i=1,\ldots,k-1$, and the node y_1 has no negative neighbour and is therefore the up left of x. It is now possible to specify an ordered rooted tree entirely in terms of the up left and the positive neighbour of each of its nodes. It is also convenient to make some further definitions of relations between the nodes of a tree. In the path from a node y to the root of the tree, the second node x is called the below of y. Thus the below of y is the head of the package of which y is a member (but not the head).

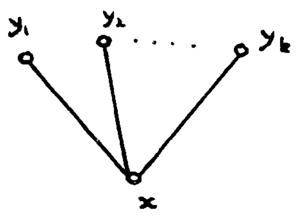


Fig. 1.

Given the two sets of quantities, up left and positive neighbour, it is possible to deduce the values of the below for each node, but this can involve a considerable amount of 'searching back', i.e. Operations of the type: 'find that element x whose positive neighbour is the element y'. This type of operation is not an easy one to perform on a computer. It is therefore convenient to introduce some further information into the quantities we have already defined. The positive neighbour vector contains information corresponding to those nodes only which have a positive neighbour, and those which do not, effectively have a blank position set aside for them. Suppose now we introduce a new vector of values called 'positive or down', which we will abbreviate to 'pord'. This quantity can be defined to be:

```
     pord (x)  = positive neighbour (x)          if one exists
               = -below (x)                      otherwise
```

The minus sign is simply a marker to tell us that we are to interpret this value as the 'below' of x rather than as the 'positive neighbour' of x. The operation of finding the below of a node in the tree now becomes very much easier and can be illustrated in terms of the recursive function:

```
     below (x) = if pord (x) <= 0 then - pord (x)
                             else below (pord (x)).
```

We have however to be a little careful in considering the below of the root node. We may adopt any one of a number of conventions on this, but in this present work it has been found most convenient to make the definition:

```
     below (root) = root
```

and this is the only node in the tree for which this is true.

Since the root has no positive neighbour, we also have:

pord (root) = - root.

It has also proved necessary to scan through the nodes of tree in some sort of standard way in order to give a canonical labelling to an unlabelled ordered rooted tree. This labelling is as follows:

1. Label the root as node 1.
2. If a node x has just been given the label i, then the node to be given the label i+1 is:
   (a) up left of (x) if this exists
   (b) otherwise the positive successor of x (pos succ (x)).
3. When the root is reached a second time then the whole tree has been labelled.

The positive successor function which was used in step 2 of the above labelling process is defined as:

pos succ (x)	= positive neighbour of x	if it has one
	= pos succ (below (x))	otherwise
	= x	if x is the root.

Using the tree in fig. 2. as an example, we obtain the following table:

Node Label	1	2	3	4	5	6	7	8	9	10	11	12
Up Left	-	-	2	1	-	4	8	-	12	-	6	-
Positive Neighbour	3	-	-	5	7	9	-	-	-	-	-	10

Below	4	3	4	6	6	11	6	7	11	9	11	9
Pord	3	-3	-4	5	7	9	-6	-7	-11	-9	-11	10
Pos Succ	3	5	5	5	7	9	9	9	11	11	11	10
Position In Canonical Ordering	4	6	5	3	7	2	8	9	10	12	1	11

This ordering, since it makes use of the positive successor relation to such a large extent should perhaps be called the positive labelling . Knuth (1968) calls this method of traversing a tree pre-order.

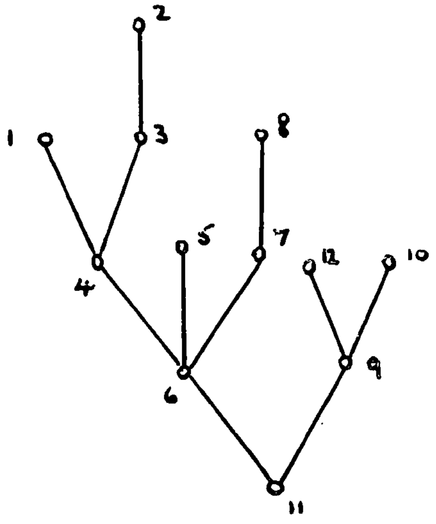


Fig. 2.

These definitions of functions are sufficient to specify a labelled tree completely, and further, we are able to move both up and down the tree without difficulty.

In the case of an unlabelled ordered rooted tree, we may represent this more succinctly. This is clearly because we do not need to carry the relationship between node labels in our representation. Two methods of representing an unlabelled tree have been devised, both of which will be discussed in a different context in chapter III.

In passing it is worth noting that the two vectors 'pord' and 'upl' (up left) can be regarded as the defining functions for a labelled tree. In fact, pord itself contains some redundant information in that the below parts of the vector can (with a certain amount of searching) be deduced from the positive neighbour parts. However given an 'upl' vector and a consistent 'pord' vector we may construct functions to discover any of the other information which we have discussed. Also, a change in either the 'pord' or the 'upl' vectors would have the effect of changing the shape of the tree. Note that since 'pord' contains some redundant information, a change in 'pord' or 'upl' may force other changes to be made to these vectors in order to preserve consistency.

II.2.1 Weight Representation.

Of the two methods of representing an unlabelled tree, the first is the weight representation. In both of the following representations we shall use a vector of small integers to describe the tree, and in each case we assume that the i-th element of the vector corresponds to the i-th node in the positive ordering of the nodes of the tree.

The weight representation of a tree was introduced in our earlier study of trees (Snow 1966) where it was described as the integer representation. This term could equally well apply to either of the representations to be described, so that it was thought to be a sensible idea to change the name to weight representation. For any rooted tree (not necessarily ordered), each node in the tree is the root of some subtree (if the node is a terminal node, then the subtree consists only of that node and if the node is the root of the whole tree, then the subtree is the whole tree). Let the <u>weight representation</u> of a tree be a vector $\underline{w} = (w_1, w_2, \ldots, w_n)$ where w_i is the number of nodes in the subtree whose root is the node i. It is clear that since the nodes are ordered according to the positive ordering of the nodes, and therefore that node 1 is the root, we have $w_1 = n$, where n is the number of nodes in the complete tree. We may also make the observation that when labelling a tree using the positive ordering, the root of any subtree is labelled before any of the other nodes in the subtree and also that once the root of a subtree has been labelled, all the nodes in that subtree are labelled before any further nodes are labelled outside that subtree. Thus, for any node j in the tree, the subtree whose root is j has a corresponding subvector in the vector $\underline{w}$. This subvector is of length k, where k is the number of nodes in the subtree on j. Also, by definition, $w_j = k$. Thus, given a vector $\underline{w}$ which represents a tree, we may choose any element w_j, and pick out the subvector $w_j, w_{j+1}, \ldots, w_{j+k-1}$ where $w_j = k$, and this defines a subtree in the tree. Another interesting property of this vector is as follows:

$$\text{let } p_1 = w_2$$

$$p_i = w_i + \sum_{j=1}^{i-1} p_j \qquad \text{for } i >= 2$$

then we have $p_1 + p_2 + \ldots + p_t = n-1$, where t is such that p_t does not refer to an element outside the vector $\underline{w}$, but p_{t+1}

would. This is a consequence of the fact that if the package whose head is the root contains the root and t other nodes, then the n - 1 nodes in the tree other than the root must all be contained in the subtrees standing on the t other nodes in this package. Since $p_1 = w_2$, we know that the first such subtree contains p_1 nodes, and that the next tree contains p_2 nodes, and so on. This relation between members of the vector $\underline{w}$ also holds within the subvectors which represent subtrees. The numbers p_i i = 1,...,k form a composition (or ordered partition) of the number n - 1. When we turn to the topic of the canonical ordering of subtrees within a tree, we shall see that these sequences of numbers p_i will become true partitions of the number n - 1. This will be dealt with in the following chapter.

II.2.2 Height Representation.

The other representation to be considered here is the height representation. Again the tree is to be represented as a sequence or vector of small integers. We have defined the height of a node x to be the length of the path between x and the root of the tree. The height representation of a tree is a vector $\underline{h} = (h_1, h_2, \ldots, h_n)$ in which the nodes of the tree are again labelled by the positive labelling, and the i-th element of $\underline{h}$, h_i is the height of the node labelled i. The node 1, which of course is the root, has height 0, and thus for any tree, $h_1 = 0$. Further, since we label the tree either by moving up the tree to the next node above, or by moving across (and possibly down) the tree, we have the relation $h_i <= h_{i-1} + 1$ for i = 2,...,n.

Again we observe that since the positive labelling labels all the nodes in a particular subtree before labelling any subsequent nodes, certain subsequences represent subtrees within the tree. Given any node in the tree, say node j, the height of this node ~~is given node~~ is given by h_j. Now we look along the vector $\underline{h}$ until we find another element k such that $h_k <= h_j$, and k is the smallest number greater than j for which this is true. Then the subsequence $h_j, h_{j+1}, \ldots, h_{k-1}$ has all its elements $h_i > h_j$, for i = $j+1, j+2, \ldots, k-1$. Thus if we consider the vector $\underline{h}' = (h_j - h_j, h_{j+1} - h_j, \ldots, h_{k-1} - h_j)$ this is a valid height representation for some tree. (This is true since the first element = $h_j - h_j$ = 0, and if $h_i <= h_{i-1} + 1$, then $h_i - h_j <= h_{i-1} - h_j + 1$). The vector $\underline{h}'$ hence represents a tree, and in fact it represents the subtree which has the node labelled j as its root. The following theorem demonstrates an interesting connection between the weight and height representation for trees:

<u>Theorem 1.</u>

For any rooted tree with n nodes, the height vector and the weight vector are related by

$$\sum_{i=1}^{n} h_i = \sum_{j=1}^{n} w_j$$

<u>Proof:</u>

Consider any node k. For each node j which lies on the path between k and the root r (including both k and r), the node k lies in the subtree of which j is the root. Thus the node k makes a contribution of 1 to each of the terms w_j of the vector $\underline{w}$ for each j in this path. Thus k makes a total contribution of l_k to the sum $\sum_{j=1}^{n} w_j$, where l_k is the number of nodes in the path k to r. Hence the sum of all the l_k gives the total sum of the w_j's and thus

$$\sum_{k=1}^{n} l_k = \sum_{j=1}^{n} w_j$$

But the number of lines in the path from the node k to the root is $l_k - 1$ and this is just the height of the node k. Thus

$$\sum_{i=1}^{n} h_i = \sum_{k=1}^{n} (l_k - 1) = \sum_{j=1}^{n} w_j - n$$

using the fact that $h_1 = 0$ and $w_1 = n$ for all trees with n nodes we have

$$\sum_{i=2}^{n} h_i = \sum_{j=2}^{n} w_j$$

Q.E.D.

II.3 Relationship Between Ordered Rooted And Rooted Trees.

As has been pointed out on a number of previous occasions (Obruca 1966, Snow 1966, Scoins 1967) there is an interesting connection between the set of ordered rooted trees of n nodes and the set of strictly ordered rooted binary trees with n terminal nodes. It can be shown that if a binary tree has n terminal nodes, then it has n - 1 non-terminal nodes. It may also be shown that these two sets of trees have the same cardinality, i.e. the trees of n nodes are equinumerous with the binary trees with n terminal nodes. It is likely therefore that we could construct a one-one mapping from one set onto the other. In fact, at least two such mappings exist. We shall describe one of them, the other being obtained by reading "right" for "left" and "left" for "right" in the following description of the mapping process.

The mapping may be described recursively. Given any ordered rooted tree, we may divide it uniquely into two parts, each of which is itself an ordered rooted tree. We arbitrarily decide that this division is carried out by "cutting" the right-most branch at the root. Let this cut be represented by a non-terminal node of the binary tree. Fig. 3. shows the tree T as being composed of two smaller trees T_1 and T_2 together with the line that will be "cut" in the decomposition process. The trees T_1 and T_2 are of course ordered rooted trees in their own right. In the binary tree, we have represented the cut by a non-terminal node, and therefore this node will have two other nodes above it, by definition of the strictly binary tree. We now carry out the mapping process recursively on the trees T_1 and T_2 so that T_1 maps onto the left subtree of the binary tree, and T_2 maps onto the right subtree. If at any stage of the recursion, the tree on either side of the cut is a single node, then this node maps onto a terminal node of the binary tree.

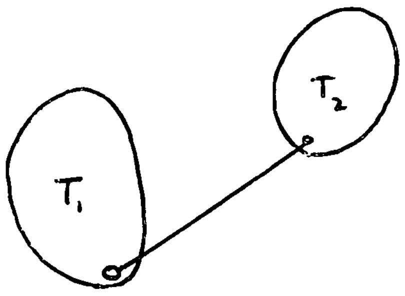


Fig. 3.

Having shown that a mapping from the set of ordered rooted trees of n nodes to the set of binary trees with n terminal nodes exists, and we may construct the reverse mapping without difficulty, we can consider the possibility of representing an ordered rooted tree in terms of a representation of its corresponding binary tree. There is a very succinct representation of a binary tree, and it is thought that this method is approaching the minimal representation of an ordered rooted tree in terms of the information content of the representation and the computer storage required.

Given a binary tree, we know that each node has only two nodes above it (or none at all). Let us represent a terminal node by the symbol *, and each non-terminal node by the ordered pair (s_1, s_2) where s_1 and s_2 are the representations of the left and right subtrees respectively on that node. The representation of the tree, which is taken as the representation of the root node, then consists of a sequence of opening and closing brackets and asterisks. Furthermore, since there are n terminal nodes in the tree, there are n asterisks in the sequence, and also there are n-1 pairs of opening and closing brackets corresponding to the non-terminals in the binary tree. Also we know that within each pair of brackets there are exactly two sub-sequences, which may themselves be bracketed sub-sequences or simply asterisks. Now since these bracketed sub-sequences also have this property, we may remove the closing brackets without any loss of information. This is because we can scan any such bracketed sequence with its closing brackets removed from the left, so that each time an opening bracket is encountered we know that we must recognise two complete subsequences and then insert the corresponding closing bracket. The following algorithm will achieve the replacement of closing brackets.

1. Initialise by setting the input and output string pointers to the beginning of their respective strings

and the stack pointer to the bottom of the stack

2. Get the next character from the input and send it to the output string. If this character is an asterisk then goto step 3. Otherwise, place a zero on the top of the stack and repeat step 2.

3. If the top of the stack is a zero then change it to a one and return to step 2. Otherwise remove the top element from the stack and send a closing bracket to the output string.

4. If the stack is now empty then quit, otherwise return to step 3.

Let us now replace each opening bracket by a 'one', and each asterisk by a 'zero'. The representation of the binary tree (and therefore of the rooted tree) is now in the form of a sequence of zeroes and ones called a terminated binary sequence (t.b.s.). This is a sequence of 2n-1 bits and is therefore very economical in computer storage space.

These concepts will now be illustrated using an example. Consider the tree in fig. 2. but regard it now as an unlabelled ordered rooted tree. It is now labelled using the positive ordering, and the result is shown in fig. 4. The weight representation and the height representation of the tree are now shown in the table below.

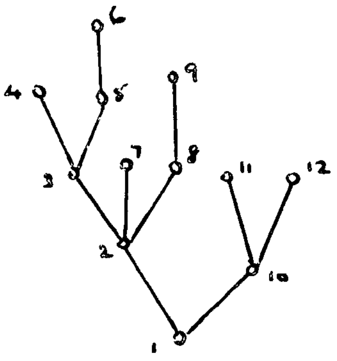
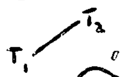
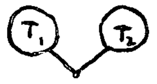


Fig. 4.

Node Label	1	2	3	4	5	6	7	8	9	10	11	12
Weight Rep.	12	8	4	1	2	1	1	2	1	3	1	1
Height Rep.	0	1	2	3	3	4	2	2	3	1	2	2

We will now describe the steps of the mapping of the ordered rooted tree into the corresponding binary tree and then to the t.b.s.. Let us denote the binary tree corresponding to a tree T by $\textcircled{T}$. Thus if T may be decomposed into 

, then we denote the corresponding decomposition of $\textcircled{T}$ by 

. As an example, the steps in the decomposition of the tree in fig. 5(i) are given in fig. 5(i)...(viii).

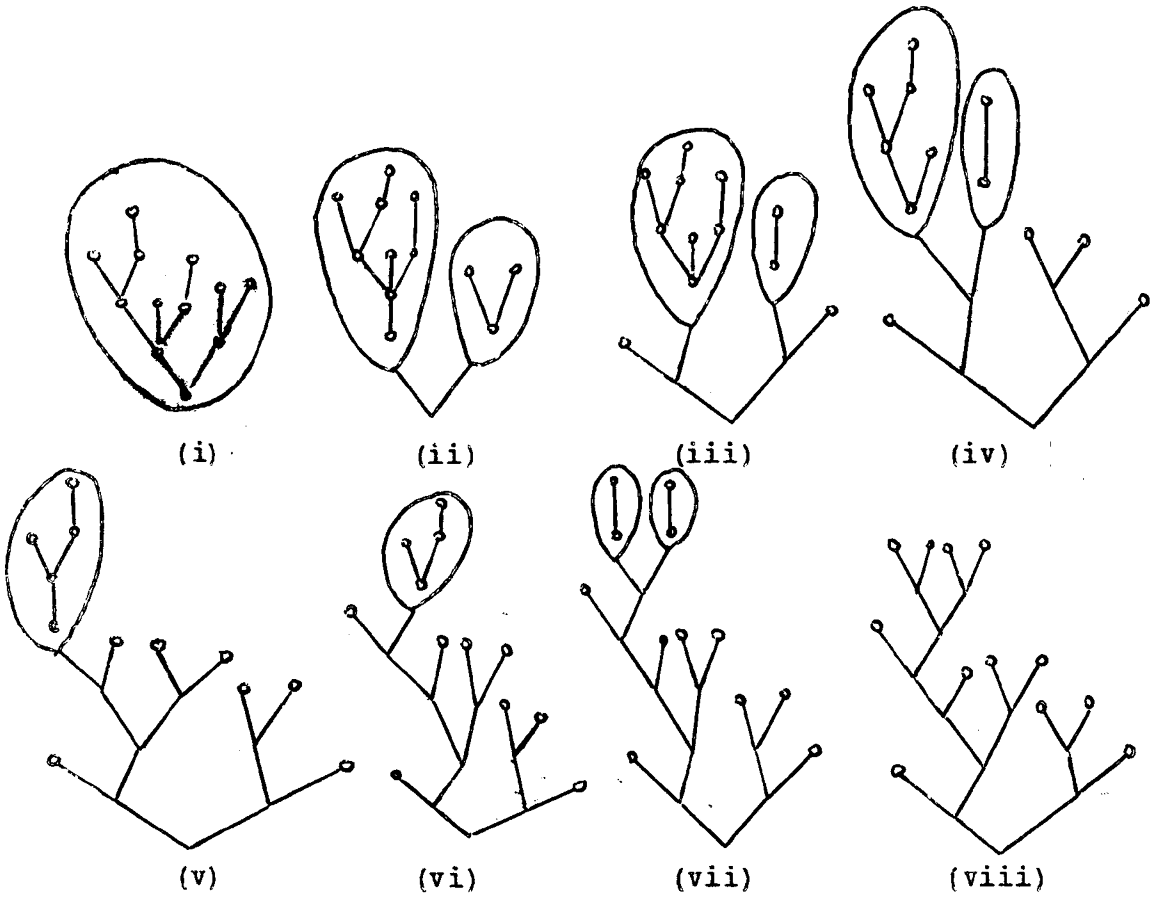


Fig. 5.

In the transition from (ii) to (iii), the left subtree decomposes into a single node on the left, while the right subtree has a single node as its right part. The growth of the binary tree therefore stops along these branches. The tree in (viii) does in fact possess 12 terminals and 11 non-terminals. This tree is represented by the following sequence of brackets and asterisks:

((*(((*((**) (**)))*) (**))) ((**)*))

which contracts to the t.b.s.:

1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 1 0 0 1 1 0 0 0

Two comments may be made about this sequence. The first is that the t.b.s. Contains one more 'zero' than it does 'ones', and in fact if we start at the beginning of the sequence and proceed along counting the numbers of zeroes and ones, the sequence terminates when the number of zeroes exceeds the number of ones by one. Hence given any sequence of zeroes and ones, we may pick any member of the sequence as the starting point and extract the first subsequence which has this property, and this represents some ordered rooted tree. An arbitrary binary sequence can therefore be thought of as representing a forest of trees, provided that the sequence is terminated when a complete tree has just been found.

The second observation which may be made about the t.b.s. is that correspondences may be established between the zeroes of the sequence and the nodes of the tree, and between the ones of the sequence and the lines of the tree (and we notice that there are the correct number of each). If this is done we see that the zeroes appear in the sequence in the same order as their corresponding nodes appear in the positive labelling of the tree.

It is clear that some representations will be more useful in some contexts, where others would be easier to use in other contexts. The different lexicographic orderings of trees described in chapter III show that a representation which gives rise to one ordering is most inconvenient when dealing with some other ordering. In fact, while the t.b.s. representation is by far the most compact of those considered here, it also appears to be the least useful. Procedures have been written to convert from one representation to another for all the possible combinations which might be required. Some of these, together with a fuller discussion of the binary tree and the t.b.s. can be found in Snow (1966).

II.4 Rooted And Free Trees.

Having dealt in some detail with the representation of ordered rooted trees, we now describe the representation of unordered rooted trees and free trees. In fact, the only way of those described previously to represent unordered trees is the 'below' representation, and that can only be applied to labelled rooted trees. The techniques used to represent and manipulate unordered rooted trees have in fact been the same as those for ordered rooted trees, but steps were first taken to ensure that the ordered rooted tree was some kind of canonical form of the unordered rooted tree. The first method used was the representation of a rooted tree by the weight vector. The canonical form of this representation is obtained by sorting the subvectors of this vector, as was described in Snow (1966), and which will be discussed in greater detail in chapter III. The same sort of technique was used to reduce a tree in height vector representation to canonical form. Again this will be discussed in the following chapter.

The same points occur in the discussion of the representation of free trees, where, in addition to the necessity for finding a canonical form, we have to impose a root on the tree where it would not otherwise have one. Two methods of carrying out this operation will also be explained in the next chapter.

II.5 Linear Graphs.

The field of representation of trees appears to be considerably more fruitful than the representation of linear graphs. However, some of the methods of representing graphs are discussed here briefly, although in the later work, the adjacency matrix was used almost exclusively as the internal representation of a graph. A more convenient method was however used for input of the graph. Given a linear graph G with n nodes and m lines in which the nodes are labelled from 1 to n, and the lines are labelled from 1 to m, we may define two matrices. The first is the adjacency matrix

$A = (a_{ij})$ where $a_{ij} = 1$ if there is a line from node i to node j

$= 0$ otherwise

The graphs dealt with here will in general be undirected, and will not contain any line from a node to itself. In these cases the adjacency matrix will be symmetric, and all the elements on the principal diagonal will be zero. The matrix will be n x n. The second matrix is called the incidence matrix, and is defined as

$M = (m_{ij})$ where $m_{ij} = 1$ if the node i is an endpoint of the line j

$= 0$ otherwise

Clearly this matrix is n $\times$ m. Now in general a graph has more lines than it has points, i.e. m>n, particularly for large n, and thus the incidence matrix occupies more space than the adjacency matrix. Neither matrix is a particularly compact representation, since there is a considerable amount of reduncancy in both of these representations. The adjacency matrix is (for our purposes) symmetric, and so could be compressed into the upper triangle of an n $\times$ n array, while the incidence matrix contains only two non-zero elements per column (where the columns correspond to the line labels). Obruca (1966) describes a method of storing a graph in (m + n) storage locations. The method used for the input of a graph to our programs is a very small variation of Obruca's method.

Assume that the nodes are labelled from 1 to n. The graph may be represented in terms of a vector of length (m + n - 1) (for undirected graphs; (m + n) locations are required for a directed graph). For each node i, the vector consists of all the nodes j to which node i is joined, and for which j > i. For the nodes j such that j is joined to i and j < i, an entry appears in the section of the vector pertaining to the node j. The list for the node i is separated from the list for the node i + 1 by some marker (such as a zero). Thus the line from i to j is represented by the appearance of the number j in the vector between the (i-1)-th zero and the i-th zero (assuming that i < j). Thus the graph in fig. 6. Would be represented as:

2, 4, 0, 3, 5, 6, 0, 4, 5, 7, 0, 0, 6, 0, 8, 0, 8, 0,

a vector of length 18,

there being 8 nodes and

11 lines in the graph. The order of the elements within each list is immaterial.

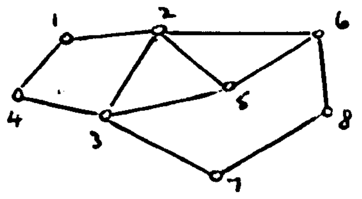


Fig. 6.

It is also interesting to notice that given n, the value of m may be deduced, since we can look along the sequence until the (n − 1)-th zero is reached, and then we know that the end of the vector has been found, and m is the number of non-zero elements passed. There is never a list for the node labelled n since there there are no nodes j for which $n < j$, so that all the lines incident to the node n appear earlier in the sequence. Also, two adjacent zeroes indicate that all the lines incident to the corresponding node have been listed previously, as for example the node 4 in fig. 5.

The adjacency matrix of a graph has a further property. Let $a_{ij}^{(k)}$ be the (i,j)-th element of the k-th power of the adjacency matrix $A = (a_{ij})$. Then $a_{ij}^{(k)}$ is the number of distinct paths of length k from node i to node j. This can be shown using an inductive argument, suppose $a_{ij}^{(k-1)}$ does represent the number of distinct paths of length k-1 from the node i to the node j. Then $a_{ij}^{(k)}$ is obtained by forming the inner product of the vector $(a_{i1}^{(k-1)}, a_{i2}^{(k-1)}, \ldots, a_{in}^{(k-1)})$ and the vector $(a_{1j}, a_{2j}, \ldots, a_{nj})$. This inner product gives us:

$$a_{ij}^{(k)} = \sum_{p=1}^{n} a_{ip}^{(k-1)} a_{pj} = \sum_{\substack{\text{all nodes } p \\ \text{adjacent to } j}} a_{ip}^{(k-1)}$$

Hence we take all the paths of length k − 1 which if increased by one line would reach the node j. Later in the work, we shall require another matrix S:

$S = (s_{ij})$ where s_{ij} is the length of the shortest path in the graph between ~~in the graph~~ nodes i and j, and this could be found by determining the smallest k such that $a_{ij}^{(k)}$ is non-zero. However, even allowing for the fact that some sophisticated methods for multiplying matrices could be found for the simple case when one (at least) of the matrices is purely binary, it is clear that there are more efficient methods for deriving the elements of S available.

The whole subject of finding the shortest path between two nodes of a graph has been studied extensively (see Pohl 1969) and various algorithms have been developed (e.g. Dijkstra (1959) and Nicholson (1966)). For finding the elements of the matrix S, however, Warshall's algorithm (Warshall 1962) is probably the best, since we are concerned with finding the shortest distance between every pair of nodes. For the case where we are required to find the shortest distance to every node from some fixed node r, some method involving the growing of a spanning tree from r is likely to be the most efficient (see chapter VI).

The adjacency matrix representation of a graph proved so useful for other manipulations in the programs that no other representation was used, except that the list representation described earlier was more useful for the initial input of the graph. A procedure was written which accepted the graph in the list representation and output the corresponding adjacency matrix for use by the rest of the program. Since no space problems were encountered during the work (the limiting factor was almost invariably computer time!) it was thought to be unnecessary to 'pack' the binary adjacency matrix into less than one matrix element per computer word, but clearly this could have been done with the consequent reduction in efficiency due to having to 'unpack' the matrix to inspect an individual item.

III Tree Indexing.

III.1 Introduction.

In this chapter an attempt is made to interpret in a meaningful way the statement $\tau_1 \prec \tau_2$ where τ_1 and τ_2 are trees, for each type of tree so far considered. This is done by listing the trees of a particular type, and perhaps of a particular size, in some order. The objective is initially to construct a mapping I from the set of trees of various classes to the set of positive integers in such a way that we may define the relation $\prec$ as

$$\tau_1 \prec \tau_2 \iff I(\tau_1) < I(\tau_2)$$

(and also $\tau_1 = \tau_2 \iff I(\tau_1) = I(\tau_2)$)

and secondly to construct a straightforward algorithm to find this mapping I and its inverse for trees of a variety of 'sizes' and types. Since the mapping I is intended to be an isomorphism between the set of trees and the set of integers between 1 and k where k is the number of trees in the set, and since the relation < is a total ordering over the integers, the relation $\prec$ is also to be a total ordering over the set of trees, i.e. for any two distinct trees τ_1 and τ_2 belonging to the set, we have either $\tau_1 \prec \tau_2$ or $\tau_2 \prec \tau_1$

Other desirable properties of the ordering relation < over the integers are also carried over by the isomorphism. Page (1971) describes methods of carrying out this process of constructing a mapping for a wide class of objects, showing, using mainly permutations of various types as illustrations, how the recurrence relations which are used to count these objects may be used to generate these same objects in some order, and also to give them a unique index together with an easy method of mapping from an object to its index and back again. In the case of trees, the counting methods are more complicated functional expressions, whose recurrence relations are deeply buried, and consequently Page's approach must be considerably extended to cope with these more complex situations. The author believes, however, that since we have a method of counting trees of various types, and linear graphs for that matter, it should in principle be possible to create an indexing scheme for all these objects, by consideration of the method used to count the objects. We will show that using different representations, however, the statement $\tau_1 \prec \tau_2$ can be made meaningful, although the ordering of the trees is greatly dependent on the representation employed.

III.2 Ordered Rooted Trees.

There are several ways of indexing ordered rooted trees of which the most obvious is perhaps the numerical ordering of the corresponding t.b.s. when considering each t.b.s. as a binary number. This method is not particularly useful since the t.b.s. is a rather specialised type of binary sequence, and the numbers produced from the t.b.s.'s do not form a particularly sensible sequence of numbers. However, as a first step , it does allow us to attach a meaning to the statement $\tau_1 \prec \tau_2$ when τ_1 and τ_2 are rooted trees.

A more natural ordering stems from the recursive way the trees are counted. Consider a typical ordered rooted tree such as that shown in fig. 1. If this tree has n nodes, then the tree τ_1 has i nodes, and τ_2 has n-i nodes for some i, 1 <= i <= n-1. Then the number of trees with n nodes is the number of trees τ_1 with i nodes × the number of trees τ_2 with n-i nodes, summed over all possible values of i. Thus, if $\mathcal{T}_n$ is the number of trees with n nodes:

$$\mathcal{T}_n = \mathcal{T}_1 \times \mathcal{T}_{n-1} + \mathcal{T}_2 \times \mathcal{T}_{n-2} + \cdots + \mathcal{T}_{n-1}\mathcal{T}_1 \quad \text{for } n >= 2.$$

From this equation, by multiplying both sides by x^n, and summing over all n from 2 to ∞, we arrive at the generating function form of the counting series for ordered rooted trees:

$$\mathcal{T}(x) = x + \{ \mathcal{T}(x) \}^2$$

$$\text{where } \mathcal{T}(x) = \sum_{n \geq 1}^{\infty} \mathcal{T}_n x^n$$

This formula may be considered as a representation of the fact that an ordered rooted tree either consists of a single node, or it is the combination of two ordered rooted trees. Thus $\mathcal{T}(x)$, which may be taken as representing the set of all ordered rooted trees, is constructed by taking the single tree with one node only (represented by the term x) or by taking elements from the product set (represented by $\mathcal{T}(x) \times \mathcal{T}(x)$ or $\{ \mathcal{T}(x) \}^2$).

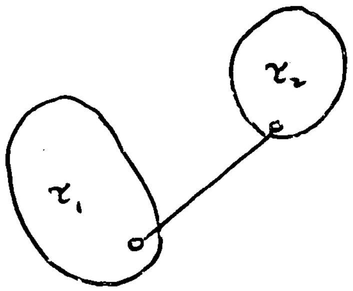


Fig. 1.

Another approach, leading to the same result, is favoured by Harary. Here we consider an ordered rooted tree to be a root node with zero or more ordered rooted trees as its principal subtrees. Thus, using the term $\mathcal{T}(x)$ once again to represent the set of ordered rooted trees, we have

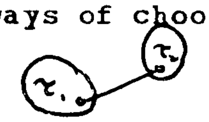
$$\mathcal{T}(x) = x\ (1 + \mathcal{T}(x) + \{\mathcal{T}(x)\}^2 + \{\mathcal{T}(x)\}^3 + \ldots\ )$$

which gives

$$\mathcal{T}(x) = x/(1 - \mathcal{T}(x))$$

Now in the natural ordering of the ordered rooted trees of n nodes, the first tree is the "join" of the tree with one node ($\mathcal{T}_1$ = 1) and the first tree with n - 1 nodes. The first tree with n - 1 nodes may be found by the same method, i.e. it is the "join" of the tree with 1 node and the first tree with n - 2 nodes. The method then proceeds recursively until the first tree of n nodes is found explicitly. In the general case, to find the k-th tree of n nodes, we examine the numbers s_i, where $s_i = \sum_{j=1}^{i} \mathcal{T}_j \mathcal{T}_{n-j}$, until we find that i such that

$$s_{i-1} < k <= s_i$$

We then know that the k-th tree decomposes into two parts τ_1 and τ_2 where τ_1 contains i nodes and τ_2 contains n - i nodes. Now there are $\mathcal{T}_i$ ways of choosing τ_1, and $\mathcal{T}_{n-i}$ ways of choosing τ_2, giving $\mathcal{T}_i \mathcal{T}_{n-i}$ ways of constructing the tree 

and we are looking for the k'-th member of this set, where

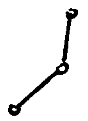
$$k' = k - s_{i-1}$$

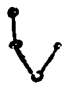
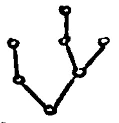
Let k be of the form $(a-1)\mathcal{T}_{n-i} + b$, where $0 < b <= \mathcal{T}_{n-i}$, and thus we now want to look for the a-th tree in the set of trees with i nodes and the b-th tree in the set of trees with n - i nodes. We may now deduce these by recursive application of the above method.

Let us clarify this method by reference to an example. Suppose we are required to find the 70th tree in the set of ordered rooted trees with 7 nodes. We have therefore $k = 70$, $n = 7$. From the table of the numbers $\mathcal{T}_n$, we can see that

$$s_3 = 66, \quad s_4 = 76$$

and so we know that our tree consists of a subtree of 3 nodes on the left, and one of 4 nodes on the right. In particular, since $k' = k - s_3 = 70 - 66 = 4$, we require the 4th such tree.

Now $\mathcal{T}_3 = 2$ and $\mathcal{T}_4 = 5$, and we may express k' as $(1-1)\mathcal{T}_4 + 4$, that is, the first tree with 3 nodes on the left, and the 4th tree with 4 nodes on the right. The first tree of 3 nodes is the composition of the first tree with one node and the first tree with two nodes. Thus the left part of the required tree is: 

according to the composition rule given in the previous chapter. Similarly, the 4th tree with 4 nodes is found to be the 1st tree with 3 nodes composed with with the first (and only) tree with 1 node, this tree is therefore , and the composition of these two gives the 70th tree with 7 nodes as: .

The alternative interpretation of the generating function equation would presumably give rise to a similar method of determining the k-th tree, but almost certainly to a different tree. In fact, since Harary's interpretation is an infinite sum of infinite sums, it is not possible to perform this mapping from numbers to trees without restricting the scope of the generating functions.

III.3 Rooted Trees.

The first approach to the problem of indexing a set of unordered rooted trees was made through the generation of ordered rooted trees. By the techniques developed in our earlier work (Snow 1966) we were able to determine whether two ordered rooted trees were isomorphic as unordered trees. Thus, one method of generating all rooted trees would be to generate all ordered trees in some sequence, and reject those which were isomorphic to trees already generated when considered as unordered trees. This was done by reducing each tree as it was generated to a canonical form as described in the aforementioned earlier work.

This method produced a vast quantity of extra work to be carried out by the program, since the number of ordered trees is approximately 2^{2n-7} for $n > 4$, whereas the number of unordered trees is only about $(2.6)^n$ for large n.

The number of rooted trees can be calculated exactly, and if we let T_n be the number of trees with n nodes, and let

$$T(x) = T_0 + T_1 x + T_2 x^2 + T_3 x^3 + \ldots,$$

By the application of Polya's theorem (Polya 1937), a classical theorem of enumerative combinatorial analysis which has been explained by a number of authors (de Bruin 1964, Liu 1968, Riordan 1958), we know that T(x) satisfies the functional equation

$$T(x) = x \exp \{ \sum_{r=1}^{\infty} T(x^r) / r \}$$

A more detailed discussion of Polya's theorem will be given later, together with a discussion of some applications. It was considered desirable in discussing an ordering relation over a set of trees, that the structure of the trees themselves should be reflected in the ordering in some way. Thus, if we have two trees τ_1 and τ_2, we wish to define index numbers $I(\tau_1)$ and $I(\tau_2)$ for τ_1 and τ_2 respectively, such that $I(\tau_1) < I(\tau_2)$ if and only if $\tau_1 \prec \tau_2$, where $\tau_1 \prec \tau_2$ also has an intuitively sensible interpretation with respect to the structure of the two trees. So, by analogy with the ordering of ordered rooted trees in the previous section, it was decided to decompose the tree τ as shown in fig. 2.

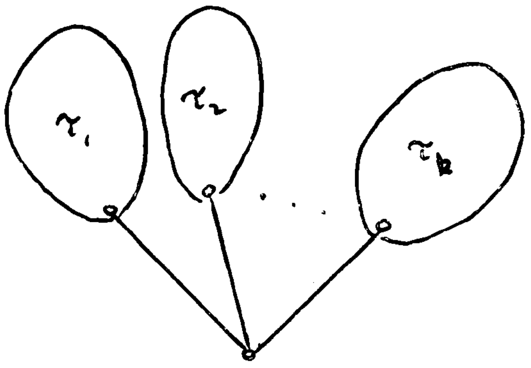


Fig. 2.

A canonical form for τ was used which was such that if τ is in canonical form, then $\tau_1 \succeq \tau_2 \succeq \ldots \succeq \tau_k$, and the τ_i are all in canonical form. Now if two trees τ and τ' are decomposed into $\tau_1, \tau_2, \ldots, \tau_k$ and $\tau_1', \tau_2', \ldots, \tau_{k'}'$ respectively, then an ordering on the set of trees might be:

$$\tau \prec \tau' \Leftrightarrow \text{either } \tau_j \prec \tau_j' \text{ for some } j <= \min(k, k') \text{ and } \tau_i = \tau_i' \text{ for } i = 1, \ldots, j-1, \text{ or } \tau_i = \tau_i' \text{ for } i = 1, \ldots, k \text{ and } k < k'.$$

Furthermore, since this ordering is given by a recursive definition, we need a starting point for the recursion, and we choose

$$\tau_\bullet = \circ \preceq \tau \text{ for all trees } \tau.$$

This ordering, which will be referred to as the natural ordering, does not depend in any way on the number of nodes in the trees, so that it is meaningful to compare two trees by this method even if the two trees have different numbers of nodes. In the examples shown later, the number of nodes in each tree is the same, although the comparison as defined is equally valid when comparing trees of different sizes.

III.3.1 Height Representation.

We may define an ordering on the height representation of a tree. This is simply the lexicographical ordering of the height vectors of the trees to be compared. In detail, this ordering is given by:

Given two trees τ and τ', with height vectors $\underline{h} = (h_1, h_2, \ldots, h_n)$ and $\underline{h}' = (h_1', h_2', \ldots, h_n')$, then $\tau \prec \tau'$ if and only if $\underline{h} < \underline{h}'$, where $\underline{h} < \underline{h}'$ is defined to mean

$$h_j < h_j' \text{ for some } 1 <= j <= n$$

$$\text{and} \quad h_i = h_i' \text{ for } i = 1, \ldots, j-1.$$

We demonstrate the connection between the natural ordering of trees and the height representation ordering of a pair of trees by means of the following theorem.

Theorem.

The ordering imposed on the set of rooted trees by their height representation ordering is the same as that given by the natural ordering.

Proof:

The proof is by induction on the height of the tree. The base point for the induction in the definition of the natural ordering is the trivial tree $\tau_0 = \cdot$. The height sequence for this tree is $\underline{h} = (0)$, whereas any other tree has at least two elements in its height vector, the first of which is always zero. This vector is therefore less than any other valid height vector, and this is the starting point for the recursion.

Now let us consider two trees τ and τ' which are represented by the height vectors $\underline{h} = (h_1,\ldots,h_n)$ and $\underline{h}' = (h'_1,\ldots,h'_n)$ respectively, and let us suppose that τ decomposes into principal subtrees $\tau_1,\ldots,\tau_k$ and τ' into $\tau'_1,\ldots,\tau'_{k'}$. Let us also assume that $\tau_j \neq \tau'_j$ but that $\tau_i = \tau'_i$ for $i = 1,\ldots,j-1$. We have constructed the height vector in such a way that $\tau = \tau'$ implies that the corresponding vectors are equal, and thus the sequence formed by concatenating the height vectors of the subtrees $\tau_1,\ldots,\tau_{j-1}$ is equal to that obtained by concatenating the height vectors of $\tau'_1,\ldots,\tau'_{j-1}$. But if each element of each of these sequences is increased by one, and the two sequences are each prefixed by a zero, then the two new sequences (which are still equal) are the partial height vectors corresponding to the root and the subtrees $\tau_1,\ldots,\tau_{j-1}$ and $\tau'_1,\ldots,\tau'_{j-1}$ respectively. Now the height sequences for τ and τ' are not equal, and by the inductive hypothesis, $\tau_j \prec \tau'_j$ if and only if $\underline{h}_{(j)} < \underline{h}'_{(j)}$. The partial height vectors $\underline{h}$ and $\underline{h}'$ can now be extended by concatenating the sequences $\underline{h}_{(j)}$ and $\underline{h}'_{(j)}$ in which each element has been increased by one. These new partial sequences $\underline{h}$ and $\underline{h}'$ are related by

$$\underline{h} < \underline{h}' \Leftrightarrow \underline{h}_{(j)} < \underline{h}'_{(j)}$$

since we know that all the elements of $\underline{h}$ and $\underline{h}'$ which precede the start of $\underline{h}_{(j)}$ and $\underline{h}'_{(j)}$ are equal. The remainder of the vectors $\underline{h}$ and $\underline{h}'$ are irrelevant since the result of the comparison is decided by the first point of difference between the vectors. Thus, by definition of the natural ordering of trees

$$\tau \prec \tau' \Leftrightarrow \tau_j \prec \tau_j'$$

By the inductive hypothesis

$$\tau_j \prec \tau_j' \Leftrightarrow \underline{h}_{(j)} < \underline{h}'_{(j)}$$

and by the lexicographical ordering of the height sequences

$$\underline{h}_{(j)} < \underline{h}'_{(j)} \Leftrightarrow \underline{h} < \underline{h}'$$

Q.E.D.

In fact the theory of the height representation, and in particular the above theorem, applies to ordered rooted trees. We introduce it here because this representation is more appropriate when considering the canonical form to which an unordered rooted tree is reduced.

We must also consider the meaning of the term 'canonical form' with respect to the height representation. The discussion of the height vector in the previous chapter mentioned that any subtree corresponded to some subvector of the height vector and gave the rule for finding such a subvector. In particular, the principal subtrees of a tree are given by the subvectors which begin with the value 1, and continue until immediately before the next element whose value is 1. We can therefore isolate the subvectors which represent the principal subtrees, and by the above theorem, we can order the subvectors lexicographically within the vector (first ensuring that each subvector is itself in canonical form) to obtain the canonical form for the height vector representation, which we see is the same as the canonical form used in the natural ordering.

This natural ordering (and hence the height sequence ordering) works very well for giving a representation to the intuitive idea of ordering the trees with n nodes, but apart from storing an ordered list of such trees together with their respective index numbers, the problem of associating an index number between 1 and T_n, where T_n is the number of rooted trees with n nodes, with each tree in the set still has no solution. However, we may now take a closer look at the way in which these trees are enumerated.

As noted earlier in section III.2, there are at least two ways to represent an ordered rooted tree and we showed the functional equations which indicated these representations. The second of these represented the view that an ordered rooted tree can be considered as a root with zero or more ordered rooted trees above it. Now, following Harary and Prins (1959), we may take the same point of view with regard to rooted trees, but with some modification. Suppose our ordered rooted tree has k subtrees above the root, then the combinations of subtrees may be taken from the full set of $\{ \mathcal{J}(x) \}^k$. In the case of rooted trees, some of these combinations will be equivalent, since certain permutations of this set of k subtrees will not change the tree. (The selection must also be made from the set

$\{ T(x) \}^k$ anyway). Thus we must take into account the permutations of the subtrees which leave the tree invariant. Polya's theorem (1937) shows how to enumerate the inequivalent members of this set. Without delving too deeply into the result discovered by Polya, we can state that if there are k subtrees above the root, then the number of inequivalent rooted trees is

$$Z(P_k, T(x))$$

where Z is the cycle index of the symmetric group P_k of permutations of k objects, and where T (x) is the counting series for rooted trees. Thus, since a rooted tree is a root with zero or more subtrees above it, we have the relation

$$T(x) = x \sum_{k>0}^{\infty} Z(P_k, T(x))$$

and it can be shown that

$$\sum_{k>0}^{\infty} Z(P_k, T(x)) = \exp\{ \sum_{r>1}^{\infty} T(x^r)/r \}$$

so that we have

$$T(x) = x \exp\{ \sum_{r=1}^{\infty} T(x^r)/r \}$$

We shall discuss Polya's theorem in greater detail, with particular reference to the counting of linear graphs, in chapter V.

In this method of counting trees, we decompose the tree into subtrees of height at most one less than the height of the original tree. This immediately suggests that there is a connection between this counting method and the height sequence representation. In fact Riordan (1960) used a very similar argument to the one of Harary and Prins given above to generate the trees with n nodes and height h. Here we denote by $T^{(h)}(x)$ the generating function for rooted trees of height at most h.

Riordan shows that the following equation holds:

$$T^{(h)}(x) = x \exp \{ \sum_{r \geq 1}^{\infty} T^{(h-1)}(x^r)/r \}$$

The generating function for trees of height exactly h is then given by:

$$T^{(h)}(x) - T^{(h-1)}(x)$$

By using the work of Riordan we may construct a table of values such as the one given below, in which we see the numbers of trees with n nodes and height h, and use it to eliminate the necessity for generating all the trees of a given size in order to find the k-th (in the height sequence ordering).

n / h	1	2	3	4	5	6	7	8
1	0	1	1	1	1	1	1	1
2			1	2	4	6	10	14
3				1	3	8	18	38
4					1	4	13	36
5						1	5	19
6							1	6
7								1

We may now merely dismiss all the trees whose height is less than the height we are interested in. For instance, suppose we require the 76th tree in the height ordering of the trees with 8 nodes. We see that there are 53 (= 1 + 14 + 38) trees of height less than or equal to 3, and so we are now looking for the 23rd tree (23 = 76 - 53) in the sequence of trees with 8 nodes and height 4 (of which there are 36).

Having established that the height of this tree is to be 4, we may make some remarks about the decomposition of this tree.

Since, by our definition of a canonical form for a tree the first subtree must be the tallest, we know that this subtree must have height = 3. Let us now consider the decomposition given by separating this subtree from the rest of the tree. (We note in passing that this is the mirror image of the decomposition we defined earlier for ordered rooted trees). Then if this first subtree τ_1 has $n(\tau_1)$ nodes, we deduce that the residue is a tree in canonical ordering with $8 - n(\tau_1)$ nodes and height <= 4. Looking at the above table, we see that $n(\tau_1)$ can take the values 4, 5, 6 or 7 in which case the number of possibilities for τ_1 is 1, 3, 8 or 18 respectively. The number of possibilities for the residue tree in these four cases is respectively 4 (= 1 + 2 + 1), 2 (= 1 + 1), 1 and 1. This, as expected, gives the total number of possibilities for this decomposition as 1.4 + 3.2 + 8.1 + 18.1 = 36. Unfortunately, however, the ordering of these trees which is implied by this method of counting the possibilities does not correspond with the lexicographical ordering of the height sequences. As an example, consider the two trees shown in fig. 3. Tree (a) has a first principal subtree which has 5 nodes, in which case it is included in the second term of the above expression, whereas tree (b) has 6 nodes in its first principal subtree corresponding to the third term of the expression. We would like therefore tree (a) to precede tree (b). However their corresponding height sequences are:

0 1 2 3 4 3 1 2

and

0 1 2 3 4 2 3 1

showing that by the natural ordering, tree (b) precedes tree (a).

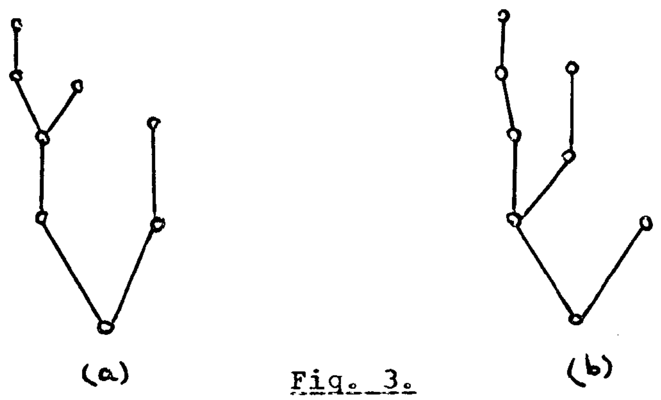


Fig. 3.

There are however methods available to generate a complete list of all the trees of n nodes in the height sequence ordering. The most straightforward method is as follows:

1. the first tree is represented by a sequence of one zero followed by n - 1 ones.
2. from any tree, we may generate the next tree in the sequence by increasing the last element in the vector by one, subject to the constraint that it may not exceed its predecessor in the sequence by more than one.
3. a check must then be made to ensure that this is a valid rooted tree (i.e. it is in canonical form). This is done recursively by comparing each subsequence with other subsequences at the same level in the same subtree.

This algorithm was programmed and it indeed generated all the rooted trees of a given size, but still trees were being generated and then rejected on the grounds that they were not in canonical form. In the case of the algorithm given above, the number of invalid trees generated was not nearly as high as it had been when all ordered rooted trees were generated and duplicates were then rejected, but a method was still sought which would generate all the rooted trees without generating duplicates. Scoins (1968) produced a recursive algorithm to solve this problem, which uses the implicit stack created by the recursion to maintain back pointers to the previous subtree, which give information about the maximum value that can be taken by each element in the height vector. This procedure then generates all the trees without duplicates. Although the height vector representation appears to tie in closely with the generating functions which we have inspected, the relationship will never be entirely satisfactory while the generating functions themselves contain implicit references to the number of nodes in the tree. By this we mean that the generating function T(x) is defined to be:

$$T(x) = T_1 x + T_2 x^2 + T_3 x^3 + \ldots$$

where the T_i are the numbers of trees with i nodes. We might hope that the relationship with the generating function methods of counting trees would be more closely related to a representation in which more account is taken of the number of nodes of each subtree, such as the weight representation to be described in the next section.

III.3.2 Weight Representation.

Consider again the functional equation whose solution is the generating function T(x).

$$T(x) = x \exp \{ \sum_{r=1}^{\infty} T(x^r)/r \}$$

thus

$$\log \{ T(x) / x \} = \sum_{r=1}^{\infty} T(x^r)/r$$

by expanding the power series and reversing the order of summation, we have

$$\log \{ T(x) / x \} = \sum_{r=1}^{\infty} -T_r \log (1 - x^r)$$
$$= \log \{ \prod_{r=1}^{\infty} (1 - x^r)^{-T_r} \}$$
$$T(x) = x \prod_{r=1}^{\infty} (1 - x^r)^{-T_r} \quad \ldots\ldots\ldots\ldots\ldots\ldots (1)$$

This result was however known to Cayley (1889), who derived it in a more empirical fashion. Cayley reasoned as follows:

Any tree of n nodes can be considered as a root, together with either one tree with n - 1 nodes above it, or two trees, one with p nodes, and one with n - 1 - p nodes above the root, or three trees with p, q, and n - 1 - p - q nodes respectively above the root, and so on. Thus:

$$T_n = T_{n-1} + \sum_{p+q=n-1} T_p T_q + \sum_{p+q+r=n-1} T_p T_q T_r + \ldots \quad \ldots\ldots\ldots\ldots\ldots (2)$$

Now each term in $\sum_{p+q=n-1} T_p T_q$ must have p >= q, and furthermore, if p = q (i.e. only if n - 1 is even) we have the term $T_p (T_p + 1) / 2$ instead of T_p^2. These restrictions are to avoid getting the same tree twice, but on one occasion with its branches reversed. Similarly the terms $\sum_{p+q+r=n-1} T_p T_q T_r$ must have p >= q >= r, and if p = q ≠ r or p ≠ q = r we have $T_p(T_p + 1)/2\ T_r$ or $T_p\ T_q\ (T_q + 1)/2$. Also, if p = q = r, then the appropriate term is $T_p (T_p + 1) (T_p + 2) / 6$. Similar restrictions must be placed on the terms containg four or more factors, where if there are k terms alike, the term T_p^k must be replaced by the number of k-combinations with repetition. Cayley shows that this method of counting trees leads directly to the generating function which satisfies equation (1).

Examination of this generating function suggests that partitions are connected with this method of counting trees, and indeed, each term in the equation (2) may be regarded as being derived from a partition of n - 1, and thus we can consider the possibility of ordering the trees according to an ordering of the partitions of n - 1. Clearly, if the i-th part of a particular partition is p_i, then the ordering of subtrees within a tree is tied to the ordering of the parts p_i.

To illustrate Cayley's method of counting trees, suppose we know the values of T_n for n = 1,...,6:

$T_1 = 1$, $T_2 = 1$, $T_3 = 2$, $T_4 = 4$, $T_5 = 9$, $T_6 = 20$,

we may find the value of T_7 by writing down all the partitions of 6:

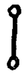
1 1 1 1 1 1	$\frac{1}{6!} T_1 (T_1+1)(T_1+2)(T_1+3)(T_1+4)(T_1+5)$	= 1
2 1 1 1 1	$T_2 \frac{1}{4!} T_1 (T_1+1)(T_1+2)(T_1+3)$	= 1
2 2 1 1	$\frac{1}{2!} T_2 (T_2+1) \frac{1}{2!} T_2 (T_2+1)$	= 1
2 2 2	$\frac{1}{3!} T_2 (T_2+1)(T_2+2)$	= 1
3 1 1 1	$T_3 \frac{1}{3!} T_1 (T_1+1)(T_1+2)$	= 2
3 2 1	$T_3 T_2 T_1$	= 2
3 3	$\frac{1}{2!} T_3 (T_3+1)$	= 3
4 1 1	$T_4 \frac{1}{2!} T_1 (T_1+1)$	= 4
4 2	$T_4 T_2$	= 4
5 1	$T_5 T_1$	= 9
6	T_6	= 20
		48

Hence T_7 = 48.

We may now use this method of counting trees as the basis for an ordering of all the trees of n nodes. Let us denote the set of rooted trees with n nodes by set(T_n). The braces underneath the sets indicate that the trees included in any brace correspond to a single part of the partition.

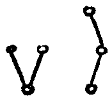
Set (T_1) = { }

T_2 -> partitions of 1 -> 1

Hence set (T_2) = {  }

T_3 -> partitions of 2 -> 1 1

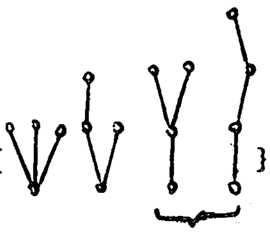
2

set (T_3) {  }

T_4 -> 1 1 1

2 1

3

set (T_4) = {  }

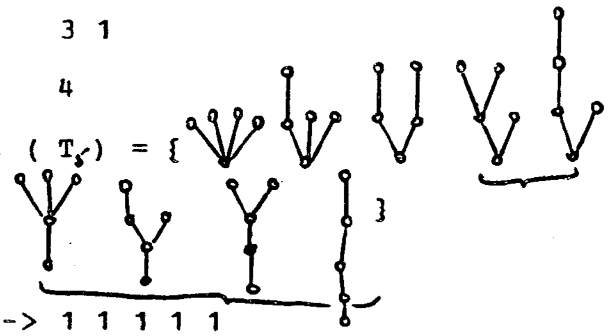
T_5 -> 1 1 1 1

2 1 1

2 2

3 1

4

set (T_5) = {  }

T_6 -> 1 1 1 1 1

2 1 1 1

2 2 1

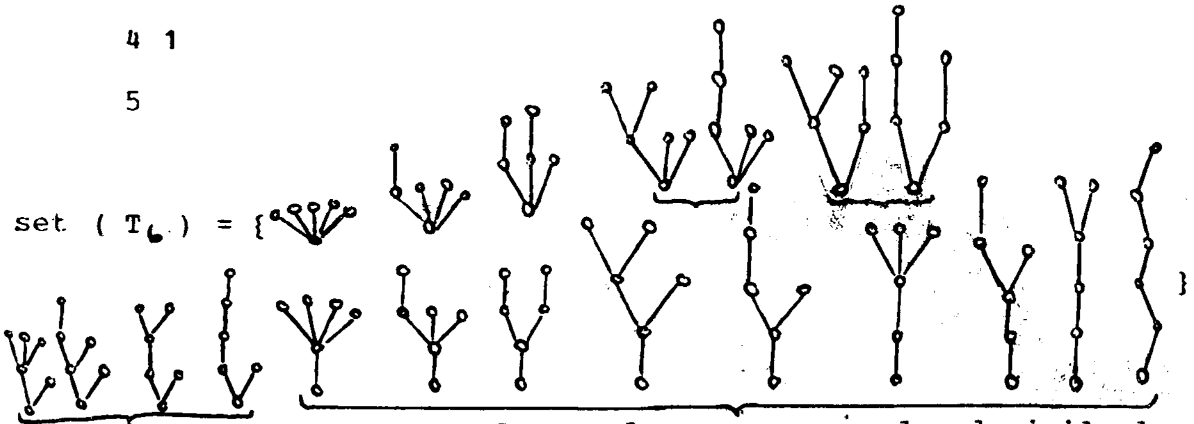
3 1 1

3 2

4 1

5

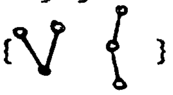
set (T_6) = {



}

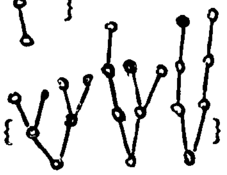
The trees of 7 nodes and more are ordered similarly. However it is instructive to see how the partition 3 3 of 6 is dealt with. The contribution $T_3 T_3$ to the sum T_7 is in fact $\frac{1}{2} T_3 (T_3 + 1)$ and not T_3^2. Similarly, the corresponding operation in the tree ordering gives:

if T -> {



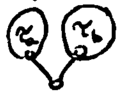
}

$\frac{1}{2} T_3 (T_3 + 1)$ -> {



}

i.e. we have



and we ensure that $\tau_a >= \tau_b$. This extends naturally to the cases where there are k equal parts in any partition.

Using the Cayley ordering for the trees with n nodes, we are able to associate with each number in the set { $1, \ldots, T_n$ } a unique tree, and this tree may be found without reference to the other trees in the set, but only to the numbers $T_1, \ldots, T_n$. We do however require an ordering of the partitions of $2, \ldots, n$, and also an ordering of the combinations with repetition for many of these numbers. If all these numbers are available then we can find the k-th tree in this ordering, whenever $1 <= k <= T_n$.

Define a tree to be of type P, where P is the partition of the number n - 1, $p_1, p_2, \ldots, p_r$, when the tree is of the form shown in fig. 4. In the figure, the tree τ_i contains p_i nodes, for i = 1,...,r.

Now the number of trees of type P may be calculated. Let this number be T(P). Thus, given an ordering for all partitions of n - 1, and a number k, the type of the tree number k in the set of all trees of n nodes may be found. This is done by subtracting T(P) from k for each P in order, until the value of k becomes negative. When this happens, we deduce that the k-th tree is of type P' where P' is the current partition in the ordering. Let k' be the last positive value of k in the subtraction process. Then the k-th tree in the set of n node trees is the k'-th tree of type P'.

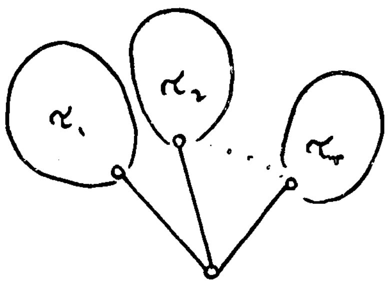


Fig. 4.

Now to find the k'-th tree of type P', assume that P' is the partition $(p_1', p_2', \ldots, p_r')$. If we take the case when all the p_i' are distinct, we allow the last value in this list to vary most quickly. In other words, we take the first tree in each of the sets of trees with p_i' nodes for i = 1,...,r-1, and take each tree with p_r' nodes in turn, and when all these trees have been counted, we take the second tree with p_{r-1}' nodes and again take all the trees with p'_r nodes. If some of the p'_i are equal, we are concerned with ordering the set of combinations with repetition. The obvious ordering for these is simply as follows:

Suppose we require k-combinations with repetition of the elements $1,\ldots,n$. Then these combinations are to be ordered by taking the first element as 1 up to n in order, and for each value of the first element (let it be i), we take the (k-1)-combinations of the elements $1,\ldots,i$. We shall give two examples of the way a particular tree can be found.

Let us first attempt to find the 27th tree with 8 nodes. By generating the partitions of 7 and counting the numbers of trees for each partition we have:

1 1 1 1 1 1 1	1 tree
2 1 1 1 1 1	1 tree
2 2 1 1 1	1 tree
2 2 2 1	1 tree
3 1 1 1 1	2 trees
3 2 1 1	2 trees
3 2 2	2 trees
3 3 1	3 trees
4 1 1 1	4 trees
4 2 1	4 trees
4 3	8 trees

As these partitions are generated, the corresponding number of trees are subtracted from the value 27, and before the last partition (4 3) is generated, the value has been reduced to 6, and thus would go negative on subtraction of the number (8) of trees with this partition. Hence, we now are looking for the 6th tree with partition 4 3. Already part of the weight vector for the resulting tree can be set up:

$$\underline{w} = (8,4,-,-,-,3,-,-)$$

where the dashes represent values which have yet to be determined. Let the trees with 4 nodes be denoted by the numbers 1,2,3,4, and the trees with 3 nodes be denoted by the letters a,b, then the combinations which we can have are (in order) 1a, 1b, 2a, 2b, 3a, 3b, 4a, 4b. Now the 6th tree in this sequence is the tree denoted by 3b, i.e. the 3rd tree in the set of trees with 4 nodes, and the second tree with 3 nodes. In the same way as before, we may find the find the 3rd tree with 4 nodes by enumerating the partitions of 3:

1 1 1	1 tree
2 1	1 tree
3	2 trees

The third tree in this sequence is then the first tree with partition 3, which itself corresponds to the first partition of 2, i.e. 1 1. Our weight vector now becomes:

$$\underline{w} = (8,4,3,1,1,3,-,-)$$

To fill in the last two dashes in this vector, we now inspect the second tree with 3 nodes, i.e. the tree b above. This corresponds to the second tree partition of 2 (since the first partition of 2 represents only one tree), and this in turn makes reference to the first partition of 1. Thus, the weight vector has the final form:

$$\underline{w} = (8,4,3,1,1,3,2,1).$$

The other example to be considered is to find a tree from a larger set of trees. Let us consider the determination of the 121st tree in the set of tree with 12 nodes. By the same counting process of partitions as before, it is found that the 121st tree with 12 nodes is the 16th tree with partition 4 4 3. The set of trees with this partition consists of 20 trees, and we again denote the trees of 4 nodes by 1, 2, 3 and 4, and the trees of 3 nodes by a and b. The 2-combinations of 1, 2, 3, 4, in order, are:

1 1, 2 1, 2 2, 3 1, 3 2, 3 3, 4 1, 4 2, 4 3, 4 4.

But each of these combinations has to be used in conjunction with the two trees a and b. The final result of picking out the 16th tree in this set gives that the required tree is made up of the trees 4 2 b, i.e. the fourth tree with 4 nodes, the second tree with four nodes and the second tree with 3 nodes. This gives rise to the weight vector:

$$\underline{w} = (12,4,3,2,1,4,2,1,1,3,2,1).$$

The two trees generated by this method in the two examples given are shown in fig. 5.

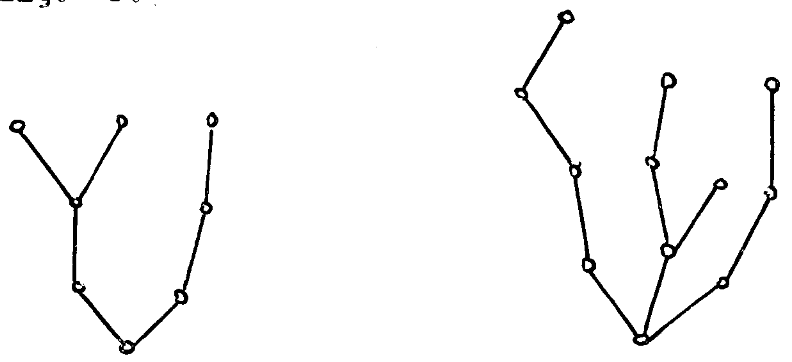


Fig. 5.

The weight vector referred to here is of course the weight representation which has already been described in chapter II. Although the weight representation is the most natural to use when discussing Cayley's ordering of trees, this ordering unfortunately does not quite correspond exactly with the lexicographical ordering of the weight vectors. This is because we have considered Cayley's ordering as fixing a partition at level 1 of the tree and allowing the partitions at level 2 to vary through all their possible values. However, by the strict lexicographical ordering of the weight vectors a partition at level 2 may have to be fixed while the later parts of the first level partition may have to be changed. We illustrate this with an example. Consider the two trees shown in fig. 6.

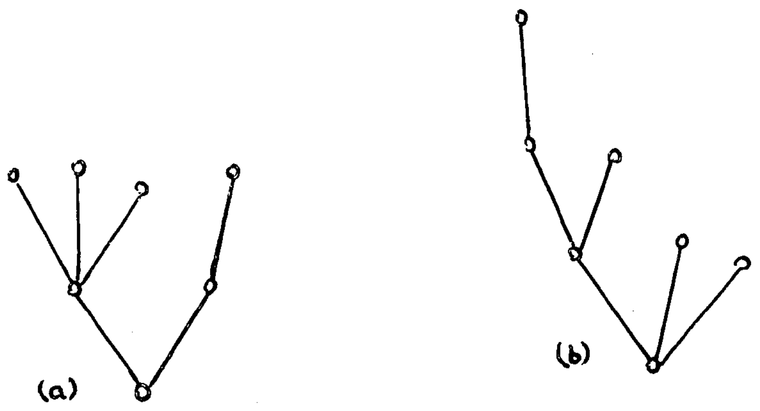


Fig. 6.

Tree (a) clearly comes before tree (b) in the ordering by weight vectors, since their respective weight representations are:

(a) 7 4 1 1 1 2 1

and:

(b) 7 4 2 1 1 1 1

but the first level partition for tree (a) is 4 2 while that for tree (b) is 4 1 1. The weight representation ordering can of course be modified by making the comparison of elements of the weight vectors in a slightly different order. For instance, if we compare the weight vectors of the two trees in fig. 6. in a term by term fashion, we find that $w_3^{(a)} < w_3^{(b)}$ and hence we deduce that tree (a) comes before (b) in the ordering. If however we first compare $w_1^{(a)}$, $w_1^{(b)}$ and then $w_2^{(a)}$, $w_2^{(b)}$ and on finding both of these pairs equal we then compare $w_{2+w_2}^{(a)} = w_{2+4}^{(a)} = w_6^{(a)}$ with $w_6^{(b)}$ we find that tree (b) is now less than tree (a). The full algorithm for scanning the weight vector in this order is:

1. compare the first element of each vector, and if these are different, the result of the comparison between the two trees is the result of this comparison; otherwise,
2. Set $i = 2$, and $k = w_1$.
3. Compare $w_i^{(a)}$ and $w_i^{(b)}$ and if these differ then the result is the result of this comparison; otherwise,
4. Set $i = i + w_i^{(a)}$.

   If $i <= k$ then return to step 3;
5. Set i = smallest value such that $w_i^{(a)}$ has not yet been considered and set $k = w_{i-1}^{(a)}$; if there is such an i, return to step 3; otherwise the algorithm terminates with the result that the two trees are equal.

Using this algorithm, the trees to be compared are examined in the same way as they are generated in the Cayley ordering, by looking at the whole partition at one level before moving further up in the tree structure. The same result could be achieved by defining the weight vector in such a way that the nodes are labelled in a different order, viz. by labelling all the nodes at one level of the tree before labelling the nodes at any higher level. If the weight vector were defined in this way, the lexicographic ordering of the weight vectors would correspond exactly with the Cayley ordering of the trees. The use of the word level is quite consistent since the nodes corresponding to values within a single partition all have the same height value.

III.4 Free Trees.

In the two previous sections, we have been able to order the corresponding types of tree according to some representation both of the trees themselves, and of the numbers $\mathfrak{J}_n$ and T_n. The reason why this was possible was that we were able to express the numbers $\mathfrak{J}_n$ and T_n in terms of a sum of products of earlier numbers in the same sequence. That is, we have been able to show that trees of n nodes could be described in terms of tree with n - 1 or less nodes. In the following discussion, we have to modify our method slightly since we have to take account of subtraction operations which appear in the counting formulae for free trees.

III.4.1 Weight Representation.

The most easily described formula for the generating function for free trees is due to Otter (1948) and is also referred to by Riordan (1958). If

$$t(x) = t_1x + t_2x^2 + t_3x^3 + t_4x^4 + \ldots$$

where t_i is the number of free trees with i nodes, then it can be shown that t(x) satisfies the equation

$$t(x) = T(x) - \tfrac{1}{2}\{ T^2(x) - T(x^2) \}$$

Riordan, in his description of the derivation of this equation, defines the centroid of a tree, and this is the same as the centre of number as defined by Cayley (1889).

Given a free tree, at any point in the tree we have a number of lines incident to it. Each such line determines a subtree of the tree, and the number of nodes in each of these subtrees is known as the weight of that subtree. Let the weights of the subtrees at some node x be a, b, c, etc. where a >= b >= c >= ... Then a + b + c + ... = n - 1 where the tree contains n nodes. Now if a > $\frac{1}{2}$n, the subtree which corresponds to the weight a (let us call it B(a)) is said to be predominant. If a = $\frac{1}{2}$n (which implies that n is even), then B(a) is said to be merely dominant, and if a < $\frac{1}{2}$n then all the subtrees B(a), B(b), B(c), ... are said to be subequal.

Theorem.

Given any tree with n nodes, if n is odd then there is one and only one node for which the subtrees are subequal. If n is even, then either there is one and only one node for which the subtrees are subequal, or else there is no such point, but there are two adjacent points both of which have a merely dominant subtree, and all the other nodes have a predominant subtree.

Proof: (cayley)

Clearly, if n is odd there can be no merely dominant subtree, since this implies that a = $\frac{1}{2}$n for some node, but a must be an integer. Thus it remains to show that there is one and only one node with all its subtrees subequal.

Starting at any node, the values a, b, c, ... may be calculated for that node. If $a < \frac{1}{2}n$, then we have a centre of number (at least one). Otherwise, move to the adjacent point in the subtree B(a). For this node we may now calculate the quantities a', b', c', ... and one of these quantities is equal to 1 + b + c + ... Thus $a' <= a - 1$. So we can move about the tree, each time moving into the predominant subtree, until we reach a point for which the subtrees are subequal. We now have to show that there is only one such node. Suppose there are two such nodes x_1 and x_2. Then x_1 has subtrees $B(a_1)$, $B(b_1)$, $B(c_1)$, ... and x_2 has subtrees $B(a_2)$, $B(b_2)$, $B(c_2)$, ... Now $a_1 < \frac{1}{2}n$, and therefore $b_1 + c_1 + \ldots > \frac{1}{2}n-1$. Now the node x_2 must be in one of these subtrees, and hence a_2 cannot be less than $1 + b_1 + c_1 + \ldots > \frac{1}{2}n$, contradicting the assumption that all the subtrees at x_2 are subequal. Thus, for n odd, there exists a unique point whose subtrees are subequal

For the case where n is even, the argument proceeds exactly as before, except that there may be a situation when $a = \frac{1}{2}n$, for some node. Carry out the process as described above, until a point is found whose subtrees are subequal. If, however, a node is found such that $a = \frac{1}{2}n$ (and $b + c + \ldots = \frac{1}{2}n - 1$) we now proceed to the adjacent point in the subtree B(a). Now we find a', b', c', ... where $a' = 1 + b + c + \ldots = \frac{1}{2}n$, i.e. we have found a pair of adjacent nodes both having merely dominant subtrees. The proof that these are the only such points proceeds as before. Q.E.D.

Cayley defines the point with all its subtrees subequal to be the _centre of number_, or in the case where there are two adjacent nodes with merely dominant subtrees these two nodes together with the line joining them are defined as the _bi-centre of number_. These two nodes are sometimes referred to separately as the _half-centres of number_ of the tree. Riordan (1966) uses the terms _centroid_ and _bi-centroid_ for the centre and bi-centre of number respectively.

In attempting to generate all free trees, we may generate all rooted trees and reject all those for which the root does not coincide with the centroid, or with one of the half-centres of number. In this latter case, we are faced with the problem of deciding which of the two ways in which a free tree may be represented as a rooted tree we should reject. However, by the theorem, we know that there can be no such trouble as long as the tree has an odd number of nodes.

Returning to Otter's formula for the number of free trees, we can expand the formula to obtain an explicit expression for the number t_n, the number of free trees with n nodes.

$$t_n = T_n - \{ \sum_{p+q=n} T_p T_q - T_{n/2} \} / 2 \qquad (T_{n/2} = 0 \text{ if n is odd})$$

$$= T_n - \sum_{p>q} T_p T_q \qquad \text{if n is odd}$$

or

$$= T_n - \sum_{p\geq q} T_p T_q - \tfrac{1}{2} T_{n/2}^2 + \tfrac{1}{2} T_{n/2}$$

$$= T - \sum_{p\geq q} T_p T_q - \tfrac{1}{2} T_{n/2}(T_{n/2} - 1) \qquad \text{if n is even.}$$

This method of counting trees is based on the definition of the centroid (or centre of number). Suppose now that each tree is to be considered as two parts, the left-most subtree above the root, and the remainder of the tree, as shown in fig. 7. Assume that the tree is in canonical form for rooted trees as defined by the Cayley ordering. Then if the number of nodes in the left-most subtree is p, and $p < n/2$, then the root of the tree is the centroid. (This follows from the definition of centroid). Now consider the case when n is odd.

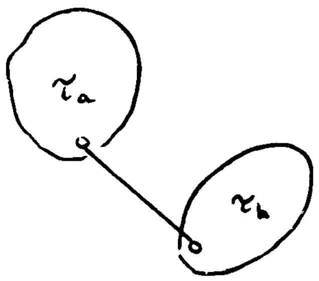


Fig. 7.

Then if p is the number of nodes in the left subtree, the remainder of the tree contains q (= n - p) nodes. Now for the root of this tree to be the centroid we must have $p < n/2$ and $q > n/2$, i.e. $p + q = n$ and $p < q$. Hence if $p > q$ this tree is excluded from the list of free trees, and so the term $-\sum_{p>q} T_p T_q$ appears in the counting expression for t_n. In the case where n is even a further term comes in to the reckoning. In this case it is possible for p to be equal to n/2. For this value of p, the tree has no centroid, but it has a bi-centroid, and a correction term is required in the expression for t_n to ensure that the same free tree does not get counted twice - when the root is at each end of the bi-centroid. As we saw when examining the counting methods for rooted trees, if two parts of the tree were of equal weight, the corresponding term in the counting expression was $\frac{1}{2}T_{n/2}(T_{n/2} + 1)$ and not $T^2_{n/2}$. So in this case, all the trees which decompose into subtrees such that $p = q = n/2$ are a set of $T^2_{n/2}$ trees, of which only $\frac{1}{2}T_{n/2}(T_{n/2} + 1)$ are required, i.e. from the set of all T_n rooted trees with n nodes we must omit (in addition to those for which $p > n/2$) $\frac{1}{2}T_p(T_p - 1)$ of the trees where $p = n/2$. This then accounts for the appearance of this term in the expression for t_n when n is even.

This study of the counting expressions for the t_n leads directly on to a method of generating the free trees with n nodes in order. If n is odd, we generate all the free trees from the set of rooted trees without having to reject any, in the case where n is even, some of the generated trees will have to be rejected, but this is only for the trees which have a bi-centroid, and there is reason to believe that the set of trees with bi-centroids is relatively small, and less than half of these have to be discarded. In n is odd then, we know from Cayley's theorem that each tree has a centroid. Hence the free trees may be represented by those partitions of $n-1$ for which the largest part $p_1 < \frac{1}{2}n$. Thus, if only these partitions are generated, we generate only the free trees. In the case where n is even, we may again generate the partitions for which $p_1 < n/2$, but we must now go on to look at the partitions which give $p_1 = n/2$, and be must be somewhat more careful when looking at the trees to which they correspond. These are the trees with bi-centroids, and here they must all be generated and any duplicates must be rejected. Cayley also gives the numbers of trees with bi-centroids and centroids and from this set of numbers we see that the set of trees possessing a bi-centroid is relatively small. (These numbers will be given later in the Appendix). A second reason why it is relatively inexpensive to discard some of the trees with bi-centroids is that the method of deciding whether to throw a tree away is trivial.

If a tree has been generated for which $p_1 = n/2$, we know that

$$\sum_{i=2}^{k} p_i = n/2 - 1$$

since this tree is represented by a partition of $n - 1$. Now the second level partition in the first subtree may be compared very easily with the remainder of the whole tree. Again we arbitrarily decide that if we have a tree with a bi-centroid we reject this tree if the first subtree is less (in the sense of our ordering) than the remainder of the tree.

Two examples will again help to clarify the points made here.

By looking at the sequence of trees of 7 nodes in the Cayley ordering, we are interested in the trees whose weight vector $\underline{w}$ has the value 3 or less for w_2, and we see that there are 11 of them, exactly the same number as the number of free trees of 7 nodes. Explicitly these trees are:

7 1 1 1 1 1 1

7 2 1 1 1 1 1

7 2 1 2 1 1 1

7 2 1 2 1 2 1

7 3 1 1 1 1 1

7 3 2 1 1 1 1

7 3 1 1 2 1 1

7 3 2 1 2 1 1

7 3 1 1 3 1 1

7 3 2 1 3 1 1

7 3 2 1 3 2 1

These trees are drawn out in fig. 8.

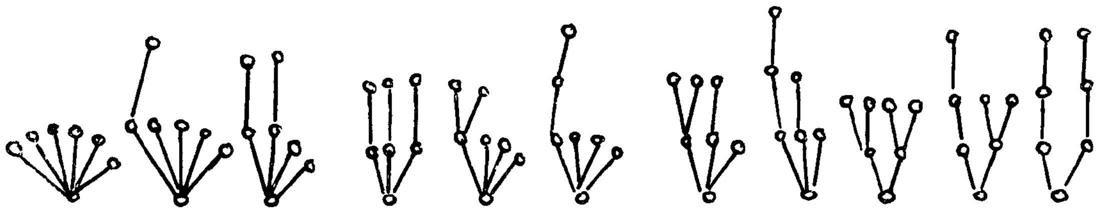


Fig. 8.

We can also see the relationship between the other trees in the set of rooted trees with 7 nodes, and the terms in the expression for t_7.

$$t_7 = T_7 - T_6 T_1 - T_5 T_2 - T_4 T_3$$
$$= 48 - 20 \times 1 - 9 \times 1 - 4 \times 2 = 11$$

and we can see in the full set of rooted trees the trees whose first level has $p_1 = 6$, 5 and 4 respectively.

The other example illustrates what happens in the case when the trees have bi-centroids. Here we take the trees with 8 nodes. The trees for which the first level partition has largest part p equal to 3 are generated as in the previous example, but we need to look more closely at those trees with the value 4 as the second element of the weight vectors.

The total number of such trees is $16 = (4)^2 = T_4^2$, and they are:

```
8 4 1 1 1 1 1 1 *
8 4 2 1 1 1 1 1 &
8 4 3 1 1 1 1 1 &
8 4 3 2 1 1 1 1 &
8 4 1 1 1 2 1 1
```

8 4 2 1 1 2 1 1 *

8 4 3 1 1 2 1 1 &

8 4 3 2 1 2 1 1 &

8 4 1 1 1 3 1 1

8 4 1 1 1 3 2 1

8 4 2 1 1 3 1 1

8 4 2 1 1 3 2 1

8 4 3 1 1 3 1 1 *

8 4 3 1 1 3 2 1

8 4 3 2 1 3 1 1 &

8 4 3 2 1 3 2 1 *

Of these, those which are marked with an asterisk have the two parts to be compared equal to each other, and thus they must be included in the set of free trees. Those that are marked with an ampersand are those whose first subtree and remainder parts τ_1 and τ_2 respectively are related by $\tau_1 \succ^{*} \tau_2$. These also have to be included in the set of free trees. It will be seen that each tree in the set of those marked with an ampersand has a dual in the set of those unmarked. Fig. 9. shows these trees.

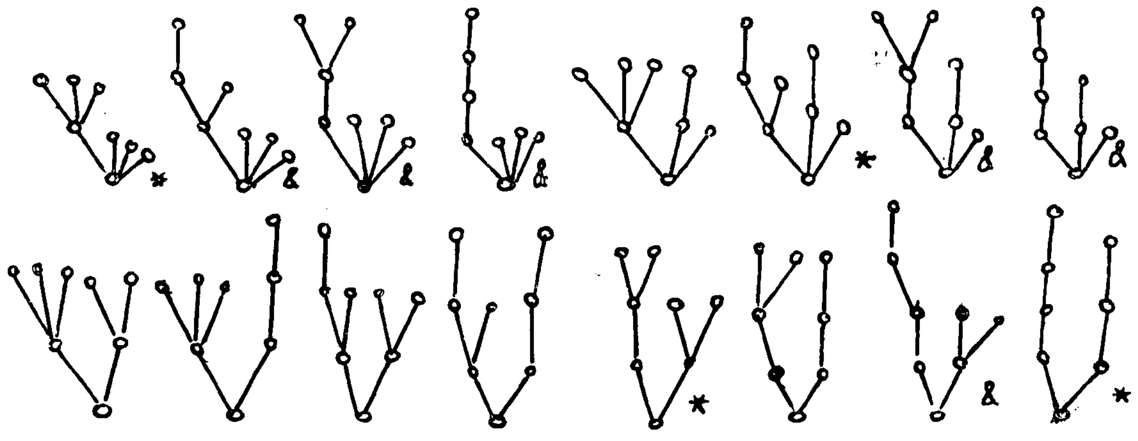


Fig. 9.

III.4.2 Height Representation.

There is another special point in a free tree which can be identified. This point is the centre of the tree. From each node in the tree, there is one and only one path to every other point in the tree (by the definition of a tree). Let the distance between any two nodes be the length of the path joining those two nodes. The diameter of a tree is defined to be the greatest distance between any two nodes in the tree. If the diameter is even, then there is a node at the mid-point of the diameter, and this point is called the centre of the tree. If the diameter of the tree is odd, then there is a middle line in the path, and this line together with its two end-points is called the bi-centre.

Theorem.

The centre or bi-centre of a tree is unique.

Proof:

Suppose there are two centres, i.e. there are two distinct paths each equal in length to the diameter which have distinct mid-points x_1 and x_2. Let y_1 and y_2 each be an extremity of the path with mid-points x_1 and x_2 respectively, such that the diameters do not intersect between x_1 and y_1 or between x_2 and y_2, and let the length of each diameter be $2r$. Then the distance from x_1 to y_1, and from x_2 to y_2 is r in each case. But since the tree is connected and x_1 and x_2 are distinct, there must be a path of length d (> 0) from x_1 to x_2, and hence the path from y_1 to y_2 must pass through x_1 and x_2 and its length is equal to $r + d + r > 2r$, contradicting the statement that the original 'diameters' were of maximum length.

The proof for the uniqueness of the bi-centre is almost identical.

Q.E.D.

One algorithm for finding the centres or bi-centre of a tree is to remove all the terminal nodes and the lines to which they are joined to give a reduced tree. This 'stripping' process may now be repeated on the reduced tree, and so on, until either one or two nodes remain. In these cases we have found the centre or bi-centre of the tree respectively. It is clear that this algorithm will terminate with the correct result since each iteration removes one line from each end of every diameter, and that the definition of diameter implies that this process will take at least as many iterations to remove all the lines from a diameter as are required to remove all the lines from any other path in the tree. Since a line is removed from each end of a diameter, the algorithm terminates when the middle of the diameter is reached.

It is possible to determine whether the root of a rooted tree coincides with its centre, or with one end of the bi-centre. This operation again becomes trivial if the tree is held in its height representation. If, however, free trees are being generated by finding such trees in the set of rooted trees, some care must again be taken to ensure that duplicates are not accepted due to the root being at one end of the bi-centre. In the canonical form of a tree according to the height representation, the principal subtrees planted at the root are ordered by their height. Thus the first two subtrees are the highest. If in any rooted tree the height of the first two subtrees are equal, then the root of this tree is at its centre. If the height of the first two subtrees differ by one (i.e. the first subtree has height one greater than the second subtree) then the root is at one end of the bi-centre of the tree. Let us assume that the tree has the form shown in fig. 10. In this case the root is at one end of the bi-centre, and we know that the height of the first subtree is one greater than the height of the second subtree, i.e. h(τ_a) = h(τ_b), where h(τ) is the **maximum** height of the tree τ. Now if $\tau_a = \tau_b$, then the same free tree rooted at the other end of its bi-centre would be identical as a rooted tree. However, if $\tau_a \neq \tau_b$, then rooting the tree at the opposite end of its bi-centre would give two distinct rooted trees. Thus, precautions must be taken to ensure that when a tree is found whose root is coincident with one end of the bi-centre, a previous tree is not being duplicated. This can be prevented by checking that $\tau_a >= \tau_b$.

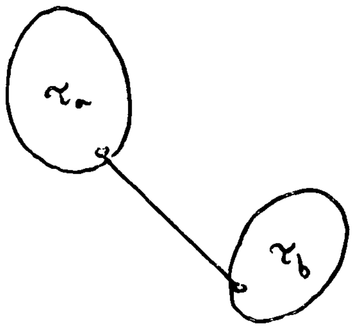


Fig. 10.

Thus, as in the case of the centroid, it is very easy to decide whether a given rooted tree is also a valid representation of a free tree, provided that the appropriate representation of the rooted tree is being used.

We are not aware of the existence of any counting methods for trees with centres and bi-centres analogous to Cayley's work on the counting of trees with centroids and bi-centroids, but by observation it appears that the number of trees with bi-centres is in general much greater than the number of trees with bi-centroids for free trees with a given number of nodes. This is obviously true for trees with an odd number of nodes since we have already seen that there are no trees with bi-centroids, whereas we can always construct at least one tree with a bi-centre - the tree where the longest path is 3, and all the nodes are joined to the same node x except one, which is at a distance 2 from x, such as for example the tree in fig. 11.

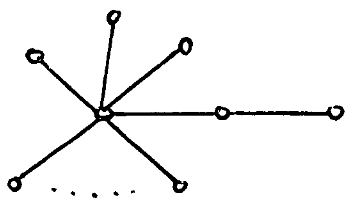


Fig. 11.

We may count the trees which possess a bi-centre by considering such trees in the following:

Let the tree be decomposable into two trees τ_a and τ_b as illustrated in fig. 12. It has already been pointed out that $h(\tau_a) = h(\tau_b)$, and that $\tau_a \succ e \, \tau_b$. Now consider the tree shown in fig. 12. This tree has one more node than the tree shown in fig. 10., but we may still impose the same conditions on the two component trees τ_a and τ_b. Thus, these trees may be counted using the formula:

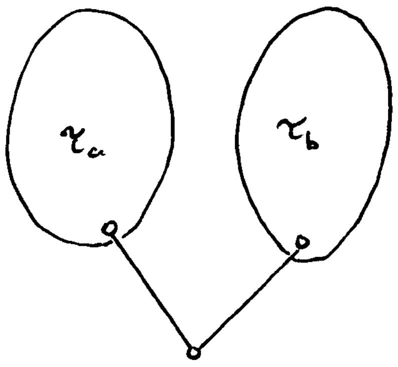


Fig. 12.

$$\sum_{h>0}^{n/2} \sum_{\substack{i>j \\ i+j=n}} T_{ih} T_{jh}$$

Where T_{nh} is taken to mean the number of trees with n nodes and height h. Clearly, the only tree with height 0 is the trivial tree with one node, and so $T_{n0} = 0$ for all $n > 1$. Similarly there are no trees for which $h > n-1$, and so $T_{nh} = 0$ for $h >= n$. This formula is not entirely satisfactory, since the condition that the left subtree of the root must be greater than the right subtree should strictly refer to the height representation ordering, whereas the $i > j$ which appears in the inner summation actually refers to the number of nodes in the two subtrees. However, this formula does give us the number of trees with n nodes which possess a bi-centre.

We made reference earlier to the fact that the number of trees with a bi-centroid was given by the formula:

$$\frac{1}{2} T_{n/2}(T_{n/2} + 1)$$

and so we are able to draw up the following table:

n	4	6	8	10	12	14	16
no. of trees with bi-centre	1	3	11	51	274	1541	9497
no. of trees with a bi-centroid	1	3	10	45	210	1176	3670

From this we see that for n >= 8 the number of trees with a bi-centre is greater than the number of trees with a bi-centroid, which together with the fact that there are no trees with an odd number of nodes which have a bi-centroid, leads us to the conclusion that the method of generating free trees using the weight representation is the more efficient to use, since the number of extra comparisons required to ascertain whether a bi-central tree has been previously generated is clearly greater than the number of comparisons of bi-centroidal trees.

IV The Graph Isomorphism Problem.

IV.1 The Problem.

This chapter describes the graph isomorphism problem, and reviews some of the approaches to its solution. The main approach which was taken in this work turned out to be very similar to the work of Corneil, and consequently many of the results quoted here are to be found in Corneil (1968) and Corneil and Gotlieb (1970).

The main problem in finding whether two graphs are isomorphic is concerned with distinguishing between nodes of an unlabelled graph which appear to be similar, or to establish that the two nodes are in fact completely indistinguishable.

Given two graphs G and G', with sets of nodes V and V' respectively, a one-one mapping σ from V onto V' (i.e. σ is a bijection) is called an <u>isomorphism</u> if for any pair of nodes x_1 and x_2 in G, (x_1, x_2) is a line in G if and only if $(\sigma x_1, \sigma x_2)$ is a line in G'. If an isomorphism exists between G and G' then the two graphs are said to be <u>isomorphic</u>.

IV.2 Unger's And Sussenguth's Methods.

Most graph isomorphism testing algorithms attempt to set up correspondences between the nodes of the two graphs in terms of sets. Two sets of nodes are considered to be equivalent at some level, and then more graph attributes are applied in an attempt to reduce the size of the corresponding sets. A solution is found when each set contains just one node, and this corresponds to a set containing a single node of the other graph. These methods are described by Unger (1964) and also by Sussenguth (1965) in which the problem is made slightly simpler by the introduction of attributes taken from the application which is being described by the graph, rather than using purely graph attributes. The nodes of the graph are divided into subsets according to some property which must be invariant under isomorphism. More and more properties are then introduced to try to reduce the size of the sets. The final objective is that each set should contain only one node, and an isomorphism is determined between the graphs by the set correspondences. Because Sussenguth's method is taken from a chemical problem, one of the properties which can be made use of is the fact that the nodes correspond to chemical elements, and thus the nodes were initially labelled (or at least coloured).

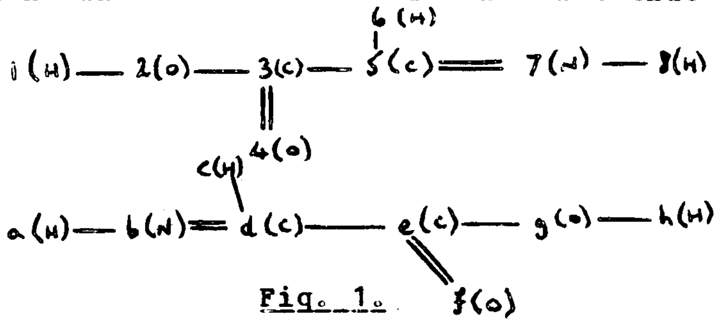


Fig. 1.

To describe this method in greater detail, the steps will be explained with reference to an example. Given the two chemical structures shown in fig. 1., is there a mapping from one to the other which preserves adjacency? In this case, we can see by inspection that the two graphs are in fact isomorphic, but we will illustrate that the action of Sussenguth's algorithm gives an isomrphism. First of all, we know that atoms of a certain type must be carried into atoms of the same type, i.e. we have a partial labelling of the nodes. In fig. 1., the node label is given first for each node, followed in brackets by the chemical element which is held at the node, that is, the 'colouring' of the node.

Thus we have the set correspondences:

{1,6,8} <--> {a,c,h}

{2,4} <--> {g,f}

{3,5} <--> {d,e}

{7} <--> {b}

We know that any isomorphism carries the elements of one set into elements of the corresponding set. We also know that since some nodes are the endpoints of double links and some of single links, thus making use of another chemical property (which may be considered as a colouring of the lines), we can set up some more correspondences:

{1,2,3,5,6,7,8} <--> {a,b,c,d,e,g,h}

{3,4,5,7} <--> {b,d,e,f}

Now combining these two groups of correspondences we have:

{1,6,8} <--> {a,c,h}

{2} <--> {g}

{4} <--> {f}

{7} <--> {b}

{3,5} <--> {d,e}

Now whenever a correspondence is set up where there is only one node in each set, we may introduce the sets of nodes which are joined to that node. Thus from {2} <--> {g} we have:

{1,3} <--> {e,h}

and from {4} <--> {f} we have

{3} <--> {e}

and from {7} <---> {b} we have

{5,8} <---> {a,d}

We may now deduce a full set of correspondences:

{1} <---> {h}	{5} <---> {d}
{2} <---> {g}	{6} <---> {c}
{3} <---> {e}	{7} <---> {b}
{4} <---> {f}	{8} <---> {a}

This example shows how the algorithm operates, but the two graphs are particularly simple in structure and the algorithm is able to form a full set of correspondences. We also notice that these graphs are trees, and we have shown (Snow 1966) that we are able to demonstrate isomorphisms in the general case between trees.

In a more complicated example, particularly without the assistance of the chemical attributes which Sussenguth employs, there may come a time during the operation of the algorithm that the partitioning of the nodes into subsets ceases without setting up the correspondences between single nodes. In this situation we can take one of two courses of action. The first is to find a further criterion by which the nodes may be partitioned into sets. The other is to try an assignment. This is done by taking one node from each of two corresponding sets. We now assume that these two nodes do correspond, i.e. that we can find an isomorphism in which these two nodes are mapped onto one another, and attempt to make the algorithm match up the rest of the nodes. If this succeeds, then an isomorphism has been determined, but if it fails then this assignment is rejected, and another tried. The criterion for the non-existence of an isomorphism between the two graphs is that the cardinality of any pair of corresponding sets is not the same, and if this should happen, we have first to see if we have made any assignments, and if so break them, and if no assignments have been made, or if there are no more assignments which can be made, we conclude that the graphs are non-isomorphic. Sussenguth goes on to show how the same method can be used to solve the subgraph isomorphism problem, i.e., to determine whether a given graph is a subgraph of another given graph. This problem is inherently more difficult since the correspondences between sets are defined by set inclusion rather than by set equality.

Corneil (1968) in his introductory chapter gives a comprehensive list of the sorts of criteria which might be applied to a graph to partition the nodes into sets, and which are invariant under isomrphism. Thus using the method of Sussenguth, we can probably show fairly quickly that two graphs are non-isomorphic, if that is true, but we may have to apply more and more criteria in order to determine an isomrphism if one exists. It is felt that using the assignment technique it could be possible to perform a lot of operations on a lot of different assignments before an isomorphism is found.

IV.3 The Classification And Refinement Method.

Later in the same work, Corneil tries to develop a single algorithm which when applied in a number of ways will partition the nodes into sets, and seeks to prove that if this algorithm fails to discriminate between two nodes then these nodes are entirely equivalent to each other. Hence if we are forming set correspondences between nodes of two graphs, and Corneil's algorithm produces corresponding sets which have more than one node each, then any assignment made between elements of these corresponding sets automatically must succeed. With the aid of a single conjecture, Corneil is able to show that his algorithm does in fact find what are defined as transitive subgraphs.

The approach made here is to use algorithms similar to those described by Corneil, but with the objective of reducing a single graph to a canonical form. This canonical form is intended to be independent of the original labelling of the graph, and hence is invariant under isomorphism. Thus, to show that two graphs are either isomorphic or non-isomorphic, we compare the canonical forms of the two graphs, and if they are the same, then we know that the graphs are isomorphic, otherwise they are non-isomorphic. On the other hand, Corneil, like Unger and Sussenguth, operates on the two graphs under test simultaneously, and if at any stage they are found to have different characteristics, the algorithm terminates immediately with the result that the graphs are not isomorphic.

IV.3.1 The Refinement Algorithm.

The basic technique used both by Corneil and ourselves is similar to a node classification algorithm given by Read and Parris (1966). This algorithm begins with the assumption that the nodes of a graph can be classified in some way (we shall return to this later). The algorithm then proceeds:

1. Place all the nodes with the same classification in the same class. Suppose this places the nodes in k 'equivalence' classes $V_1, V_2, \ldots, V_k$ where an 'equivalence' class is intended to imply that we have so far found no labelling independent criterion by which we can distinguish between nodes within the class.
2. For each node x, form a list of numbers $(a_1, a_2, \ldots, a_k)$, where a_i is the number of nodes in the class V_i which are adjacent to x.
3. Within each class V_i, sort the nodes into order according to their corresponding list. If for some pair of nodes x and x', we have lists $(a_1, a_2, \ldots, a_k)$ and $(a'_1, a'_2, \ldots, a'_k)$, we say that $(a_1, a_2, \ldots, a_k) < (a'_1, a'_2, \ldots, a'_k)$ if

   $a_i < a'_i$ for some i, and

   $a_j = a'_j$ for $j = 1, \ldots, i-1$.
4. Now refine the classification as follows:

   If two nodes were previously in the same class, and have the same list, then they remain in the same class, otherwise the nodes are put into different classes.
5. Repeat steps 2 to 4 until either each class contains exactly one node, or until an iteration of the algorithm fails to increase the number of classes.

It is clear that the algorithm will always terminate since either the n nodes of the graph will be completely classified into n classes, or else the classes will remain exactly the same through a whole cycle of the algorithm. Because the sorting of lists takes place only within classes, if two nodes are in different classes at any time, they can never subsequently be placed in the same class by the algorithm.

This procedure, however, depends on the assumption that we are able to give an initial classification to the set of nodes. This does not present a great deal of difficulty however, since if we begin by placing all the nodes in the same class, during the first iteration the nodes would become classified by their degrees. (This is because the list for each node x would consist of one element only, a_1, where:

a_1 = the number of nodes in the set V_1 to which x is joined. In this case V_1 is of course the set of all the nodes in the graph, so that a_1 is precisely the degree of x).

Consider the algorithm at work on the graph in fig. 2. The initial classification places all the nodes in the same class V_1. The lists for each node consist of one element only:

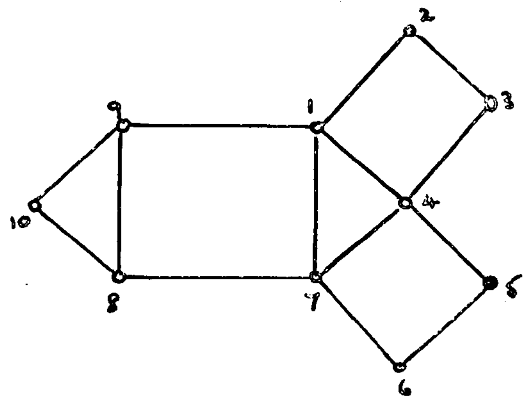


Fig. 2.

1 --> (4)

2 --> (2)

3 --> (2)

4 --> (4)

5 --> (2)

6 --> (2)

7 --> (4)

8 --> (3)

9 --> (3)

10 --> (2)

We may now re-classify the nodes according to these lists, and we obtain:

class I	-	{1,4,7}
class II	-	{8,9}
class III	-	{2,3,5,6,10}

The second iteration of the algorithm now refines this classification by the following lists:

class	node	list
I	1	(2,1,1)
	4	(2,0,2)
	7	(2,1,1)
II	8	(1,1,1)
	9	(1,1,1)
III	2	(1,0,1)
	3	(1,0,1)
	5	(1,0,1)
	6	(1,0,1)
	10	(0,2,0)

Re-ordering the nodes according to these lists, re-partitioning the set, and forming the new lists, we have:

I	1	(1,1,1,1,0)
	7	(1,1,1,1,0)
II	4	(2,0,0,2,0)

III	8	(1,0,1,0,1)
	9	(1,0,1,0,1)
IV	2	(1,0,0,1,0)
	3	(0,1,0,1,0)
	5	(0,1,0,1,0)
	6	(1,0,0,1,0)
V	10	(0,0,2,0,0)

One more iteration gives:

I	1	(1,1,1,1,0,0)
	7	(1,1,1,1,0,0)
II	4	(2,0,0,0,2,0)
III	8	(1,0,1,0,0,1)
	9	(1,0,1,0,0,1)
IV	2	(1,0,0,0,1,0)
	6	(1,0,0,0,1,0)
V	3	(0,1,0,1,0,0)
	5	(0,1,0,1,0,0)
VI	10	(0,0,2,0,0,0)

Since within each class, each node has the same list, the algorithm terminates. The partitioning of the nodes which has now been formed will be known as the <u>final partitioning</u>. Corneil's term for this algorithm is the Terminal Connection Partitioning Algorithm, and Read calls it the Refinement Algorithm. We prefer to reserve the word "terminal" for certain nodes in trees, and thus we shall refer to the final partitioning, and use Read's term of refinement algorithm. The partitioning given by the algorithm in this example displays the symmetries in the graph which are apparent by inspection of the graph as shown in fig. 2., that is, the graph is symmetric about a line passing through nodes 4 and 10.

The graph shown in fig. 3. demonstrates that this algorithm does not however classify the nodes completely into sets of entirely equivalent nodes.

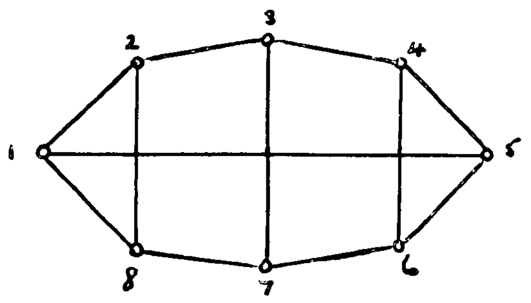


Fig. 3.

The algorithm begins by placing the nodes in the same class, and after one iteration, the nodes are classified by their degrees. However the degree of each node is equal to 3, and so the algorithm would terminate immediately after the completion of the second iteration. It is clear that there are differences between the nodes, since while nodes 1 and 5 have cycles of length 3 passing through them, the nodes 3 and 7 do not. Thus we can illustrate that the algorithm as it stands cannot always distinguish between non-equivalent nodes.

IV.4 The Automorphism Partitioning.

We shall now introduce some more definitions which will be required in the following discussions. A one-one mapping σ from the set V of nodes of a graph G onto itself is said to be an <u>automorphism</u> if $x_1, x_2 \in V$ and (x_1, x_2) is a line in G if and only if $(\sigma x_1, \sigma x_2)$ is also a line in G, i.e. σ is an isomorphism of G onto itself. In fact, an automorphism is simply a relabelling of the graph in such a way that the graph remains unchanged under the relabelling. A transitive graph is a graph G such that for any pair of nodes x and y in G there exists an automorphism σ such that $\sigma x = y$. A transitive subgraph H of a graph G is a subgraph of G such that for any pair of nodes x and y in H, there exists an automorphism σ of G such that $\sigma x = y$.

It is not difficult to show that all the automorphisms of a graph form an algebraic group, and this will be referred to as the automorphism group of the graph. We may make the remark that in a large number of cases (especially for large graphs) the automorphism group will consist of the identity automorphism only.

We know by definition that if there exists a mapping in the automorphism group which maps a node x onto a node y, then x and y are in the same transitive subgraph. We can show then that the automorphism group divides the set of vertices of G into a number of equivalence classes, or in other words, defines a partition of the nodes of the graph into subsets. The partitioning induced by the automorphism group is known as the automorphism partitioning. The objective of this section is therefore to describe algorithms which find the automorphism partitioning of a graph.

IV.4.1 The Cycle Vector.

Now by the final partitioning as given by the refinement algorithm, we might hope that the graph given in fig. 3. is a transitive graph, but as has been explained previously, this is unfortunately not so. In explaining why this graph is not transitive, e.g. why the nodes 1 and 3 are different, the next step is immediately suggested. This is the notion of the <u>cycle vector</u> for each node. For each node x, the length of the cycles passing through x may be calculated. Let $\{y_i\}$ be the set of all nodes in the graph which are adjacent to x. We then take each pair of nodes y_i, y_j $(i \neq j)$ from this set and calculate the length of the shortest path from y_i to y_j which does not pass through x. If this distance is p, then the length of the cycle through x is p+2. If the degree of x is d(x), then there are $\frac{1}{2}$ $d(x) \times (d(x) - 1)$ such pairs of nodes (y_i, y_j), and the same number of distances p. The lengths of the cycles (the values $p + 2$) are put into the vector $\underline{c} = (c_1, c_2, \ldots, c_k)$ where $k = \frac{1}{2} d(x)$ $(d(x) - 1)$, and where it is arranged that $c_i \le c_{i+1}$ for $i = 1, \ldots, k-1$. We must also agree that if there are two nodes y_i, such that there is no path joining them which does not pass through x, i.e. if x is an articulation point, then the length of the corresponding cycle is arbitrarily large. The vector $\underline{c}$ is the cycle vector for the node x. Now this vector may be used as a further discriminator between nodes - two nodes being in the same class if and only if they have the same cycle vector.

The operation of finding the shortest distance between two nodes is not a trivial one, and this must be done several times for each node in the graph. Thus the computation involved in finding the cycle vector for each node is quite lengthy. Having found this vector for each node, the classification refinement algorithm may be used to try to re-classify the nodes into smaller classes.

IV.4.2 The Vertex Quotient Graph.

A computationally more convenient method of discriminating between the nodes of a regular graph is to use the vertex quotient graph as defined by Corneil. We must however define a quotient graph first.

Given a graph, we may apply the node classification algorithm until the final partitioning is found. The quotient graph for this partitioning is then defined as:

Each class of the partitioning is represented by a node in the quotient graph. In the original graph, if a node in class i is joined to k nodes in class j, then we have a directed arc of weight k from node i to node j in the quotient graph. Since nodes of the graph may be joined to other nodes in the same class, loops (i.e. arcs from a node to itself) are permitted to appear in the quotient graph. We define the adjacency matrix of a weighted directed graph to be:

$$A = (a_{ij})$$

where $a_{ij} = 0$ if node i is not joined to node j, and otherwise it is the weight of the arc which goes from i to j. Thus it is clear that if the classification algorithm is allowed to form its lists in the iteration after the final classification is found, then these lists form the rows of the adjacency matrix for the quotient graph.

In the example given in fig. 2., the corresponding quotient graph has 6 nodes, representing the classes I, II,...., VI. Node II (node 4 in the original graph) has two connections to each of the nodes I (nodes 1 and 7) and V (nodes 3 and 5). Thus the quotient graph has a directed line of weight 2 from II to I and from II to V. If we represent the weight of an arc in a particular direction by that number of arrow-heads on the line in that direction, the lines starting at node II are shown in fig. 4. The process may then be continued to find the complete quotient graph, and this is shown in fig. 5.

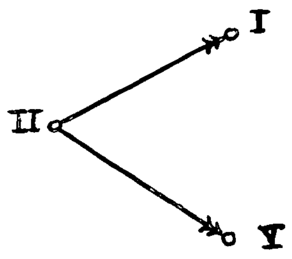


Fig. 4.

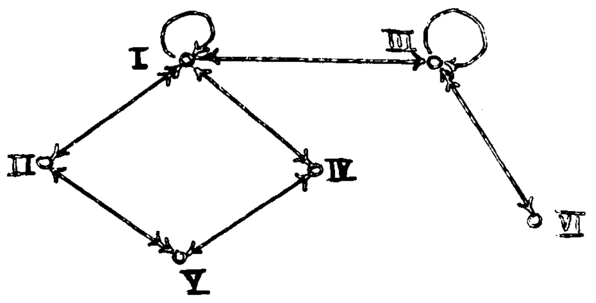


Fig. 5.

Since we have assumed that the quotient graph is formed from the final partitioning, we know that the lists corresponding to each node in any class are the same, and also we know that these lists are of length k, where k is the number of classes, and thus we may take the lists as the rows of a square matrix, and this matrix can be seen to be the adjacency matrix for the quotient graph. In the example given above, we obtain the matrix:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 0 & 2 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 \end{pmatrix}$$

For any node x of a graph, the nodes of the graph may be partitioned into two classes, {x} and V - {x}, where V is the set of nodes in the graph. The final connection partitioning algorithm may now be applied to this partitioning, and the final partitioning which the algorithm gives is called the final vertex partitioning of the graph with respect to the node x. The final vertex partitioning with respect to x may now be used to determine the vertex quotient graph with respect to x.

We will now illustrate the formation of the vertex quotient graph for the node 1 of the graph shown in fig. 3. It will be recalled that this graph is regular of degree 3, and that the connection partitioning algorithm was unable to re-classify the nodes because of its regularity. We now show the formation of the vertex quotient graph for node 1 of this graph, and quote the corresponding results for the other nodes.

The initial partitioning into classes {x} and V - {x} gives:

I	-	{1}
II	-	{2,3,4,5,6,7,8}

First iteration gives:

I	1	(0,3)

II	2	(1,2)
	3	(0,3)
	4	(0,3)
	5	(1,2)
	6	(0,3)
	7	(0,3)
	8	(1,2)

Second iteration:

I	1	(0,3,0)
II	2	(1,1,1)
	5	(1,0,2)
	8	(1,1,1)
III	3	(0,1,2)
	4	(0,1,2)
	6	(0,1,2)
	7	(0,1,2)

Third iteration:

I	1	(0,2,1,0)
II	2	(1,1,0,1)
	8	(1,1,0,1)
III	5	(1,0,0,2)
IV	3	(0,1,0,2)
	4	(0,0,1,2)
	6	(0,0,1,2)
	7	(0,1,0,2)

Fourth and final iteration:

I	1	(0,2,1,0,0)
II	2	(1,1,0,1,0)

	8	(1,1,0,1,0)
III	5	(1,0,0,0,2)
IV	3	(0,1,0,1,1)
	7	(0,1,0,1,1)
V	4	(0,0,1,1,1)
	6	(0,0,1,1,1)

This is the final vertex partitioning with respect to the node 1, and the corresponding vertex quotient graph is shown in fig. 6., for which the adjacency matrix is:

$$\begin{pmatrix} 0 & 2 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 2 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

This graph turns out to be the vertex quotient graph of node 5 as well as for node 1. By applying the vertex partitioning algorithm to each node in turn we discover that the vertex quotient graph for the nodes 3 and 7 is as shown in fig. 7., and the graph for nodes 2, 4, 6 and 8 is as shown in fig. 8.

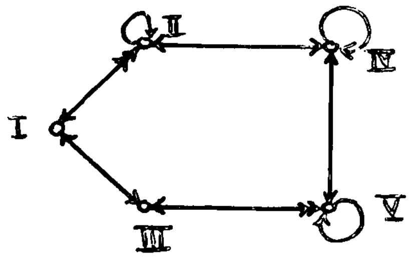


Fig. 6.

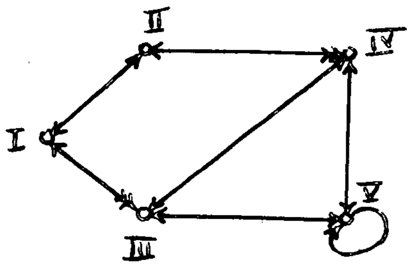


Fig. 7.

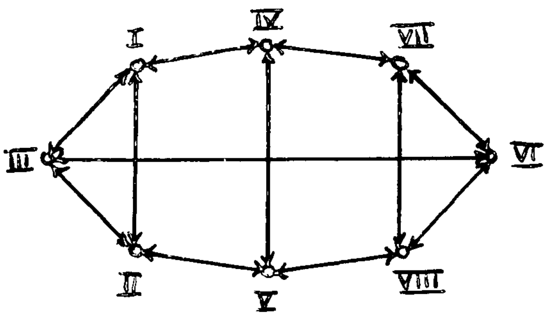


Fig. 8.

The adjacency matrices for these two graphs are respectively:

$$\begin{pmatrix} 0 & 1 & 2 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

and:

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

In the case of the last four nodes, the number of classes in the final partitioning is equal to the number of nodes in the original graph, and the resultant vertex quotient graph is simply a permutation of the nodes of the original graph.

A further interpretation of the vertex quotient graph is the consideration that if two nodes are mapped into the same node of the vertex quotient graph with respect to some node x, then these two nodes are not only symmetric with respect to the whole graph, but also are symmetric with respect to the node x within the graph.

We may now make use of the vertex quotient graphs to impose a further partitioning on the nodes of the original graph. Thus two nodes are put into the same class if and only if they give rise to the same vertex quotient graph. In effect, we are 'colouring' the nodes according to their vertex quotient graphs. We may compare the adjacency matrices of any two such vertex quotient graphs on an element by element basis, comparing the members of the first row before proceeding to compare elements in the second row, and so on, until the two matrices differ, and the result of the element by element comparison is then taken as being the result of the comparison of the matrices.

Returning to the example, we may now partition the nodes of the graph as:

I	-	{1,5}
II	-	{3,7}
III	-	{2,4,6,8}

A further attempt is made to refine this classification, which in this case has no further effect on the partitioning.

We now consider the construction of the cycle vector for the nodes of the same graph. For the node 1, the possible pairs of nodes from the set of nodes adjacent to node 1 are:

(2,8), (2,5) and (5,8)

We then remove node 1 from the graph and look for the shortest distance between each pair of nodes in the reduced graph. Clearly these distances are:

$d(2,8) = 1$

$d(2,5) = 3$

$d(5,8) = 3$

and so the cycle vector for the node 1 is $(3,5,5)$. The cycle vector for node 5 is also $(3,5,5)$ and the nodes 3 and 7 have cycle vector $(4,4,5)$. The cycle vector for the remaining nodes, 2, 4, 6 and 8 is $(3,4,5)$. Hence, using the cycle vector as a discriminator, we obtain the classification:

I	-	{2,4,6,8}
II	-	{1,5}
III	-	{3,7}

We see from this that apart from the order of the classes, the partitioning given by the cycle vectors of the nodes is the same as that given by the vertex quotient graphs. In all the examples that have been considered, this has been found to be true, although we have so far been unable to show that it is generally true. The partitioning of the nodes by their vertex quotient graphs is then used as the initial classification of the nodes prior to the refinement of this classification to find a final partitioning.

We may therefore attempt to partition the nodes of a regular graph using either the vertex quotient graph method or the method of the cycle vectors. As was pointed out previously, the calculation of the cycle vector for any particular node is a complex procedure, especially when the shortest path between a number of pairs of nodes is required. Thus it was decided to use the method of the vertex quotient graphs as described by Corneil, and the conjecture has been made that if the nodes of a graph are partitioned according to their vertex quotient graphs, then the nodes of each class form a transitive subgraph provided the graph does not have the property of 2-strong regularity, where 2-strong regularity is an extension of the concept of regularity. Corneil shows that it is possible to extend the notion of regularity indefinitely, and suggests that if a graph is h-strongly regular, then an "h-th order" extension of the vertex quotient graph can be used to distinguish between nodes which would appear to be identical using any lower order vertex quotient graph. At the present time the smallest known non-transitive 2-strongly regular graph has 26 nodes. A full definition of a 2-strongly regular graph is given in Appendix II.

An extension of the final vertex partitioning for a given node will now be considered. We assume that some partitioning for the nodes of the graph G has already been found. For each class defined by this partitioning, we would like to test the corresponding subgraph for transitivity. In the class under consideration, we select one node and set this in a class by itself, still preserving the remaining classification. Thus if the final partitioning has k classes, the new classification formed has k + 1 classes. The final partitioning algorithm may now be applied to this new classification to form a new final partitioning and its associated quotient graph. This may be thought of as the vertex quotient graph of the node singled out with respect to the original partitioning. This process takes place for each node in the class and this set of nodes may now be partitioned according to the quotient graphs thus formed.

Any class which has more than one node in it should have this test for transitivity applied to it, and if any class is found not to be transitive the whole process must be repeated in case the new partitioning formed by sub-dividing an intransitive subgraph causes a previously tested subgraph to reveal another subgraph which is not transitive. When a pass has been made through all the classes of the graph and no refinement of the classification has been made, then (subject to the conjecture given above) we know that the classes represent transitive subgraphs, i.e. each node in the class may be considered entirely equivalent to each other node in the same class. This final partitioning also gives rise to the final quotient graph for the graph G.

Throughout this discussion on the formation of quotient graphs, we may be sure that the labelling of the nodes of the quotient graph is completely independent of the labelling of the original graph, since the order of the nodes is determined only by the lexicographic ordering of certain vectors, and the elements of these vectors are also independent of the original labelling of the graph. Thus we are quite justified in considering the identity of quotient graphs as being synonymous with the isomorphism of quotient graphs.

IV.4.3 The Representative And Re-ordered Graphs.

For each class which is represented by a single node of the final quotient graph, every node within the class has the same vertex quotient graph with respect to the final partitioning. This is clearly true, otherwise a further refinement would be possible. We may now present a graph, which is defined by Corneil to be the <u>representative graph</u> G_R of the graph G.

The representative graph G_R of the graph G is defined as the final quotient graph of G, in which each node is labelled by the vertex quotient graph which is common to all the nodes of G which are in the class corresponding to that node of G_R. It is clear, but proved formally by Corneil, that if two graphs G_1 and G_2 have representative graphs $G_R^{(1)}$ and $G_R^{(2)}$ respectively, then:

$$G_1 \cong G_2 \Rightarrow G_R^{(1)} \equiv G_R^{(2)}$$

This implication is perhaps more useful in its negated converse form:

$$G_R^{(1)} \not\equiv G_R^{(2)} \Rightarrow G_1 \not\cong G_2$$

Thus, in the full graph isomorphism algorithm, if two graphs are found to have different representative graphs, then they cannot possibly be isomorphic. It follows from the conjecture made earlier that if this conjecture is true, then the converse of the above result also holds, i.e.:

$$G_R^{(1)} \equiv G_R^{(2)} \Rightarrow G_1 \cong G_2$$

If this conjecture could be proved, then the graph isomorphism problem would be solved, but since this is subject to conjecture, a further step must be taken. The final partitioning which is used to construct the final quotient graph is taken and the first class which contains more than one node is subdivided by setting one of the nodes in a class by itself.

It is immaterial which node is chosen for this, since they are all considered to be equivalent. The refinement algorithm is then applied to this new partitioning. If the result of partitioning this set as much as possible is that the classification now has n classes (where n is the number of nodes in the original graph) then the algorithm terminates, otherwise the first class which contains more than one node is again sub-divided, and the procedure repeated. When the classification does finally consist of just n classes, we form a quotient graph based on this partitioning, and we know that in this graph, each node represents a single node of the original graph. This graph is therefore simply a permutation graph of the original graph. This graph, which is called the <u>re-ordered graph</u> G_r of the graph G, is clearly isomorphic to G. Thus, if two graphs G_1 and G_2 produce the re-ordered graphs $G_r^{(1)}$ and $G_r^{(2)}$ respectively, we have:

$$G_r^{(1)} \equiv G_r^{(2)} \Rightarrow G_1 \cong G_2$$

Once again, Corneil conjectures that the converse is true, namely:

$$G_1 \cong G_2 \Rightarrow G_r^{(1)} \equiv G_r^{(2)}$$

Corneil uses these results to check the graphs for isomorphism and throughout his algorithm he is able to compare the progress of the graphs being checked, so that if at any stage the two graphs are seen to behave differently, the deduction can immediately be made that the two graphs are not isomorphic. In a later piece of work, Corneil (1971) uses essentially the same algorithm to compute the automorphism partitioning for a graph, this time operating on only one graph to find its automorphism partitioning. In this work, we require a canonical form for the graph, and we have re-programmed the algorithms described here, taking the conjectures on trust, and using the permutation induced by the re-ordered graph as being the permutation which produces the canonical labelling for the graph.

We conclude this chapter by demonstrating the algorithm at work on the graph shown in fig. 3.

We have already seen that the graph is regular of degree 3. The vertex quotient graph for each node is then computed, and these have already been illustrated in figs., 6, 7, and 8. We begin then with the initial partitioning:

I	-	{1,5}
II	-	{3,7}
III	-	{2,4,6,8}

We now illustrate how the vertex quotient graph is formed for node 1 with respect to this partitioning. Firstly, the node 1 is separated from the rest of its class, giving the partitioning:

I	-	{1}
II	-	{5}
III	-	{3,7}
IV	-	{2,4,6,8}

Refinement of this partitioning then gives the classification:

I	-	{1}
II	-	{5}
III	-	{3,7}

IV	-	{2,8}
V	-	{4,6}

and this turns out to be the final partitioning. The corresponding vertex quotient graph is shown in fig. 9. The other node in the class with node 1, node 5, also has the same vertex quotient graph. The nodes 3 and 7 have the same vertex quotient graph as each other, and this is shown in fig. 10. The 8 node graph in fig. 11. is the vertex quotient graph for each of the nodes 2, 4, 6, and 8. Since none of these classes in the original partitioning can be further refined by consideration of these vertex quotient graphs we conclude that the three classes each represent a transitive subgraph. Hence, the graphs shown in fig. 9, 10 and 11 are the labels of nodes I, II and III respectively of the representative graph which is shown in fig. 12.

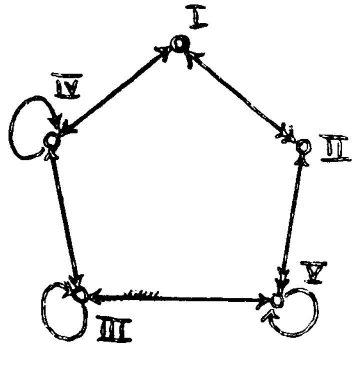


Fig. 9.

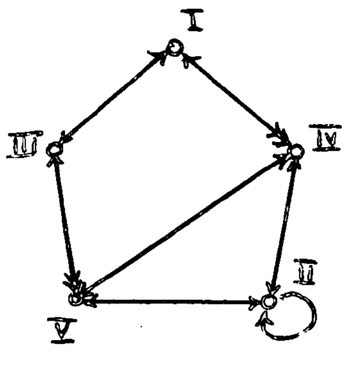


Fig. 10.

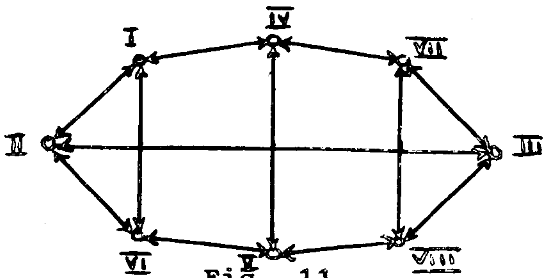


Fig. 11.

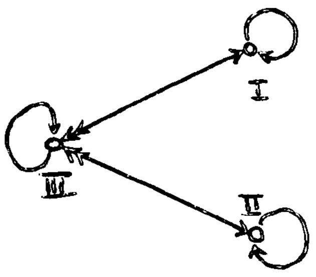


Fig. 12.

We now proceed to the formation of the re-ordered graph. The first class which possesses more than one node is class I, and so this class is sub-divided into the two classes {1} and {5}. The refinement algorithm then produces the following partitioning:

I	-	{1}
II	-	{5}
III	-	{3,7}
IV	-	{2,8}
V	-	{4,6}

Class III now is the first class which may be sub-divided and if this is done and the final partitioning is again found, we have:

I	-	{1}
II	-	{5}
III	-	{3}
IV	-	{7}
V	-	{2}
VI	-	{8}
VII	-	{4}
VIII	-	{6}

Now we have essentially defined a permutation of the nodes of the original graph, and by the conjectures put forward previously we believe that this labelling is invariant under isomorphism. In chapter V we go on to make use of the canonical labelling defined by these techniques.

IV.5 Two Problems.

Following on from the definitions made in this section, a number of interesting questions are raised. Since it is conjectured that two identical representative graphs imply that the graphs they represent are isomorphic to one another, it must certainly be possible to reconstruct the original graph (with a possibly different labelling of the nodes) from the representative graph. Actually, this is done when the re-ordered graph is formed, but the refinements which take place during the formation of this graph implicitly use the adjacency relationships of the original graph. The question of finding an algorithm to build a graph given only its representative graph is then raised.

In the spirit of Ulam's conjecture (1960) that a graph G is uniquely reconstructable from the set of n subgraphs G - {x} for each node x in G, we pose two further problems:

(a) is it possible to reconstruct a graph G of n nodes uniquely given only its n vertex quotient graphs, and:

(b) is it possible to reconstruct a graph G of n nodes uniquely from its n cycle vectors.

If we were able to prove the statement made earlier that partitioning the nodes by cycle vector or by vertex quotient graph gives the same partitioning, then the problems (a) and (b) are equivalent. However, for the moment, we must leave them as open questions.

V Graphs.

V.1 Introduction.

We set out in this chapter to construct a correspondence between the set of non-isomorphic graphs of n nodes, and the integers $1,\ldots,g_n$, where g_n is the number of such graphs. One method for counting the non-isomorphic graphs is using Polya's theorem, and it was hoped that this study would lead to a method whereby any objects which can be enumerated by using Polya's theorem may also be systematically generated using this technique.

However we were only able to reach a partial solution to this problem, in which, as we shall show, we are able to set up a systematically ordered list of the graphs of order n, but not the 1-1 correspondence with their counting numbers at which we were aiming. That is, we were unable to construct a mapping which would take a graph into its index number and vice versa. In fact a number of different lists were formed.

V.2 Relationships Between Node Labels And Line Labels.

We now consider a graph to be a collection of n points, and we may label them in any one of n! ways. For this set of n points, we have $n(n-1)/2$ positions where a line may be placed, i.e. there are $n(n-1)/2$ possible ways of choosing two points (an unordered pair) from a set of n. Now we may label these line positions according to some rule (we shall discuss some ways of labelling the line positions later), and clearly any permutation of the nodes gives rise to exactly one permutation of the line positions. Thus we may define an isomorphism between the symmetric group of order n (i.e. the group of all permutations of n objects) and a certain subgroup of the permutations of order n(n-1)/2.

Consider now a labelled graph. As we have already noted, we may define a labelling of the line positions. A bitstring may now be used to specify this labelled graph, in which a 'one' indicates the presence of a line in the corresponding line position, and a 'zero' bit indicates the absence of a line. Now clearly application of one of the permutations in this subgroup of order n(n-1)/2 (which we will denote by $L(P_n)$) will permute the bits within the bitstring, forming a new bitstring. If one bitstring α may be permuted into another bitstring β using one of these permutations, then we say that the bitstrings α and β are equivalent. It is clear that if two bitstrings are equivalent, then the two graphs which they represent are also isomorphic, since the permutation of the bits which carries bitstring α into bitstring β (i.e. a permutation of the line positions) is induced by a permutation of the points, and this is precisely the definition of isomorphism between graphs.

The use of the word "equivalent" between bitstrings is justified since it is easy to show that the relation

$\alpha \cong \beta \iff$ there exists a permutation in $L(P_n)$ mapping α into β

is an equivalence relation bearing in mind that $L(P_n)$ is a group. Thus the set of all bitstrings is partitioned by this equivalence relation into equivalence classes, each class being a set of labelled graphs which are isomorphic to one another when considered as unlabelled graphs.

We now describe the mappings from the point pairs to the lines. Two such mappings were considered, and in fact it did not seem to make much difference which one was used.

The first mapping was simply to take the point pairs in lexicographic order, and then number the elements of the list thus obtained. Hence, for n = 4, we have:

line	point-pair
1	1 - 2
2	1 - 3
3	1 - 4
4	2 - 3
5	2 - 4
6	3 - 4

This mapping has the virtue of being very simple to implement and to understand.

Let us now consider the effect on the line positions of a graph when the nodes are permuted by the cyclic permutation. Under this permutation of the nodes, we discover that the lines are also permuted cyclically, though not in a single cycle, but in a number of cycles. To give an example of this, consider the complete graph of 5 nodes shown in fig. 1. If we now let the nodes 1,...,5 be permuted cyclically, we see that the pair (1,2) is mapped into the pair (2,3), (2,3) into (3,4), and so on, and also that (1,3) is mapped into (2,4) etc. In fact, as we would expect, if we apply this cyclic permutation 5 times, the graph returns to its original labelling, and in doing so the point pairs (or lines) have gone through a complete cycle of length 5. However there are 10 lines, which means that the lines must be such that they can be partitioned into two distinct sets each of which contains 5 lines. These two sets consist of the set of lines which form the periphery of the diagram, and the set of lines on the interior of the figure.

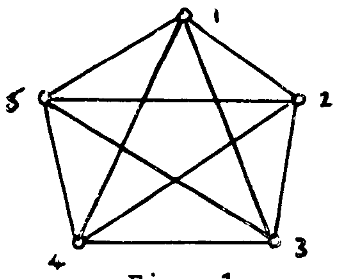


Fig. 1.

In general, for odd values of n, the lines partition themselves into (n-1)/2 sets (or cycles) each of length n. For even values of n, however, this does not happen, since the line joining opposite points of the graph will, after only n/2 applications of the cyclic permutation, finds itself in its original position (though in the reverse sense). By opposite points we mean points i, j such that |i-j| = n/2, and which therefore appear diametrically opposite one another if the graph is drawn with the nodes numbered consecutively around the periphery of the diagram. The same action as for the case where n is odd takes place for every other line of the graph. Thus the lines of a graph with an even number of nodes fall into n/2-1 cycles of length n and one cycle of length n/2. The second mapping of point pairs into an index number is by this partitioning, and within each cycle, by the value of the first node. Thus for n = 4 and n = 5 we have:

n = 4

line	point-pair
1	1 - 2
2	2 - 3
3	3 - 4
4	4 - 1
5	1 - 3
6	2 - 4

n = 5

line	point-pair
1	1 - 2
2	2 - 3
3	3 - 4
4	4 - 5
5	5 - 1
6	1 - 3
7	2 - 4
8	3 - 5
9	4 - 1
10	5 - 2

These two mappings may be described in terms of fairly simple functions:

Mapping 1.

line(i, j) = if $i>j$ then line(j, i)

else $(i-1)(2n-i)/2 + j - i$.

Mapping 2.

line(i, j) = if $i-j>n/2 \pmod n$ then line(j, i)

else $(j-i-1)n + i$. (mod n)

where in each case 'line(i, j)' is the line label for the line joining the points i and j.

In this discussion we have considered the effect of the cyclic permutation of the nodes upon the lines of the complete graph. It would, however, have been equally valid to consider a general labelled graph of n nodes, and to observe the effect on the line positions under a cyclic permutation of the nodes.

V.3 The Equivalence Graph.

The problem of finding a representative of each equivalence class in the set of all graphs may now be couched in terms of a graphical model.

Consider the set of all bitstrings of length $m = n(n-1)/2$, and let each such bitstring be represented by a node in the graph model. Then a directed line is inserted in this graph from the node α to the node β if and only if bitstring β is obtainable from bitstring α under the operation of permuting the bits in α by one of the permutations in $L(P_n)$. Let us define this graph to be the equivalence graph for the graphs of order n.

Now since for any pair of equivalent bitstrings there is a permutation mapping one into the other, the resulting equivalence graph is a collection of connected components, each of which is a complete directed graph. We may also remark that the graph may be considered to be undirected, since whenever a line from node α to node β exists, the inverse permutation is represented by a line from node β to node α. If we now take one node only from each of these connected components, this set of nodes will represent the set of inequivalent bitstrings, and hence we have obtained the set of non-isomorphic graphs.

In order to reduce the number of permutations which have to be examined, we require a set of permutations which generate the whole group. In terms of the equivalence graph model, this is the same as removing lines from the graph but without disconnecting any of the connected components. We know however, from Ledermann (1961) that the two permutations:

(2 3 ... n 1) and (2 1 3 4 ... n)

will generate the whole of the group of all permutations of order n. This means that the graph does not lose any connectivity by removing all the lines except those representing these two permutations.

Let us now consider the action of these two permutations in a slightly different light. Suppose we have a set S of n objects, and that the objects of S are to be permuted by the two permutations given, the cyclic permutation, and the transposition of the first two elements. Now consider a graph in which the nodes represent the n! ways of ordering the elements of S, and in which we have two types or 'colours' of line. A 'blue' line from node α to node β if ordering β is obtained from ordering α by application of the first of our two permutations (the cyclic permutation), and a 'red' line from γ to δ if ordering γ is mapped into ordering δ on applying the transposing permutation. The graph thus formed is highly symmetric, and it contains (n-1)! cycles of length n containing only 'blue' lines. These cycles are not connected to one another, each one corresponding to n applications of the cyclic permutation. The graph also contains a number of cycles containing alternately 'red' and 'blue' lines. These cycles are of length 2(n-1), which may be seen by considering the action of applying the cyclic permutation followed immediately by the transposition. This results in the first element remaining in its original position, with all the other elements being permuted one place cyclically. Thus, n-1 such pairs of permutations will form a cycle consisting of n-1 'blue' lines and n-1 'red' lines. Fig. 2. shows this graph for the case when n = 4. The 'blue' lines in fig. 2. are shown as solid lines and the 'red' lines are dotted.

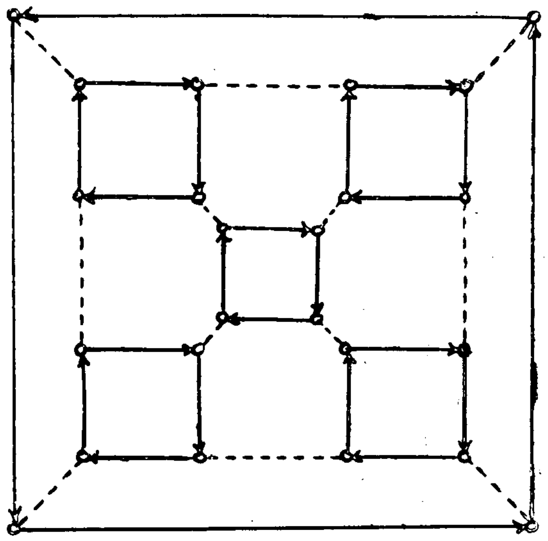


Fig. 2.

The general problem of scanning all possible results of applying any permutation to some object or set of objects is then a problem of finding a Hamiltonian path through a graph such as the one in fig. 2. It is not difficult to find a Hamiltonian path through this graph when the graph is considered as undirected, but we must bear in mind that although the 'red' lines are essentially non-directed, the cycles of 'blue' lines are directed so that the cycle is traversed in either the clockwise or the anti clockwise sense. In the diagram in fig. 2., this is seen to be true except that the outside cycle is in the opposite sense from all the other cycles. This however appears quite consistent when we consider projecting this diagram onto the surface of a sphere, and then observe the diagram from outside (or inside) the sphere. The cycles then are all in the same sense. It appears not to be possible to traverse this diagram, taking due account of the directions on the lines, so that each point is passed once and once only. It is possible however to find a Hamiltonian cycle if we allow a movement along a line in the 'wrong' direction just twice. This is shown in fig. 3.

A graph such as the one in fig. 2. shows the interconnection of the objects when subject to the permutations which we have been considering. However, if we consider a graph of n nodes as a bitstring as described earlier, we may apply the permutations to this bitstring and we do not necessarily obtain a different bitstring. Thus, for any given bitstring, we may construct a graph such as the permutation graph shown, but in which some of the nodes coincide with each other. This is because n! is an upper bound on the number of distinct labelled, but isomorphic, graphs of n nodes, whereas in general there would be less than this number. Let us show this final point with respect to an example. Referring forward to list III which shows the equivalence relations between the graphs of order 4, consider those graphs in the class containing graph 1. Corresponding to each of these graphs is a node in the 'reduced' equivalence graph, and from each such node there are two lines, a 'red' one and a 'blue' one, and some of these lines may be loops. The result of this is the graph shown in fig. 4. We also observe that a surjective mapping is defined from the set of nodes in the permutation graph shown in fig. 2., onto the set of nodes in the equivalence graph shown in fig. 4., and that this graph is one of the components of the whole 'pruned' equivalence graph. This component corresponds to the graphs of order 4 with just one line.

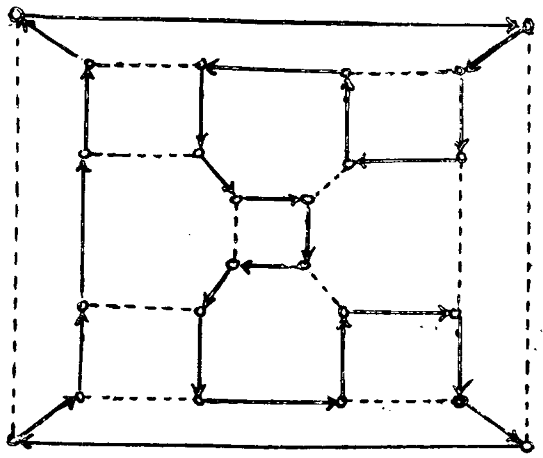


Fig. 3.

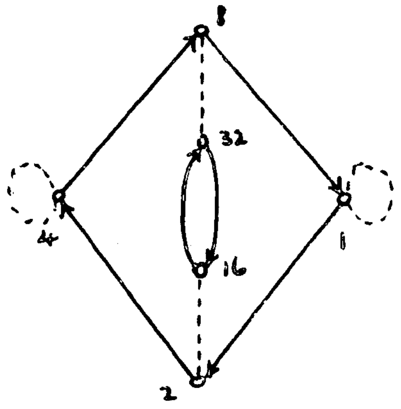


Fig. 4.

Thus we may consider attempting to find all the non-isomorphic graphs by examining the connected components of this graph. A common method of finding the connected components of a graph is by finding a spanning forest for the graph. If such a spanning forest can be found, then the roots of the trees in the forest can be taken as the representatives of the various connnected components.

A method was devised to carry out essentially these operations which discovered all the non-isomorphic graphs of n nodes. The method involved, however, inspecting each of the 2^m possible bitstrings (where $m = n(n-1)/2$), but it was only necessary to look at two permutations, and not the full symmetric group. In common with a number of connected component finding algorithms, we begin by assuming that each node is in a component by itself, and joining components together as soon as a link is discovered between them. Let us denote the two permutations

(2 3 4 ... n 1) and (2 1 3 4 ... n)

by A and B respectively. Then we observe that permutation A returns any bitstring to itself after at most n applications. Similarly B can only be applied once, since two successive applications will clearly give the original bistring once more. In terms of the graph, there are cycles of length at most n, representing successive applications of permutation A, and lines representing the permutation B which have a line in the opposite direction joining the same nodes, representing a second application of the permutation B. It may be that for some nodes, the original bitstring will re-appear after less than n applications of permutation A, in which case it is not necessary to apply this permutation any further to this node.

The following algorithm finds the subsets of equivalent bitstrings for a given n, and the non-equivalent bitstrings are found by taking one representative of each subset. Each bitstring belongs dynamically to a subset (represented by a tree) which includes its known equivalences. The algorithm gradually merges the subsets leaving finally a collection of trees each of which represents a subset of equivalent bitstrings, such that two bitstrings appear in the same tree if and only if they are equivalent. The roots of these trees may then be taken as the non-equivalent bitstrings.

1. Make each node in the equivalence graph the root of a (trivial) spanning tree by setting below(i) = i for i = $0,\ldots,2^m-1$. Thus each bitstring is in its own subset.
2. For each node i, taken in order, which is currently the root of a spanning tree, apply the permutation A n-1 times or until the bitstring i reappears whichever is sooner. For each bitstring j (except i) so

constructed, set below(j) = i. On completion of this step, the distinct trees include those nodes which can be transformed into each other using only (possibly repeated applications of) the permutation A. Each tree now contains at most n nodes.

3. For each bitstring i in order, apply permutation B once to obtain bitstring j. If i = j, then the permutation B has had no effect so ignore this case. If i > j, we may again disregard the case, since we should previously have found j such that an application of B gave the bitstring i. If i < j, then we have discovered a possible link between two of the trees established in step 2. We must first ensure, however, that i and j are in distinct trees, and this may be achieved by inspecting the roots of the trees containing i and j. (The root of the tree containing node i may be obtained by applying the function root(i) which is defined by

   root(i) = if below(i) = i then i else root(below(i))).

   If root(i) = root(j), then the nodes i and j are already in the same tree, and so we disregard this case also. If root(i) ≠ root(j), then we need to link together the (distinct) trees containg i and j, and this may be achieved by setting below(root(j)) = i.

4. A representative of each subset of equivalent bitstrings is now found by taking the roots of the trees we have formed, and these bitstrings represent

the non-isomorphic graphs of n nodes.

We observe that the action of this algorithm does not give rise to real spanning trees for the equivalence graph, but we may be sure that two nodes are put into the same tree if and only if they are in the same component of the equivalence graph. The trees formed are "reachability trees" for the equivalence graph.

An example will help to clarify the action of this algorithm. Assume that n = 4. Then the number of possible line positions is 6, and there are $2^6 = 64$ possible bitstrings. We will denote these by the numbers 0,...,63. The two permutations are:

A - (2 3 4 1)

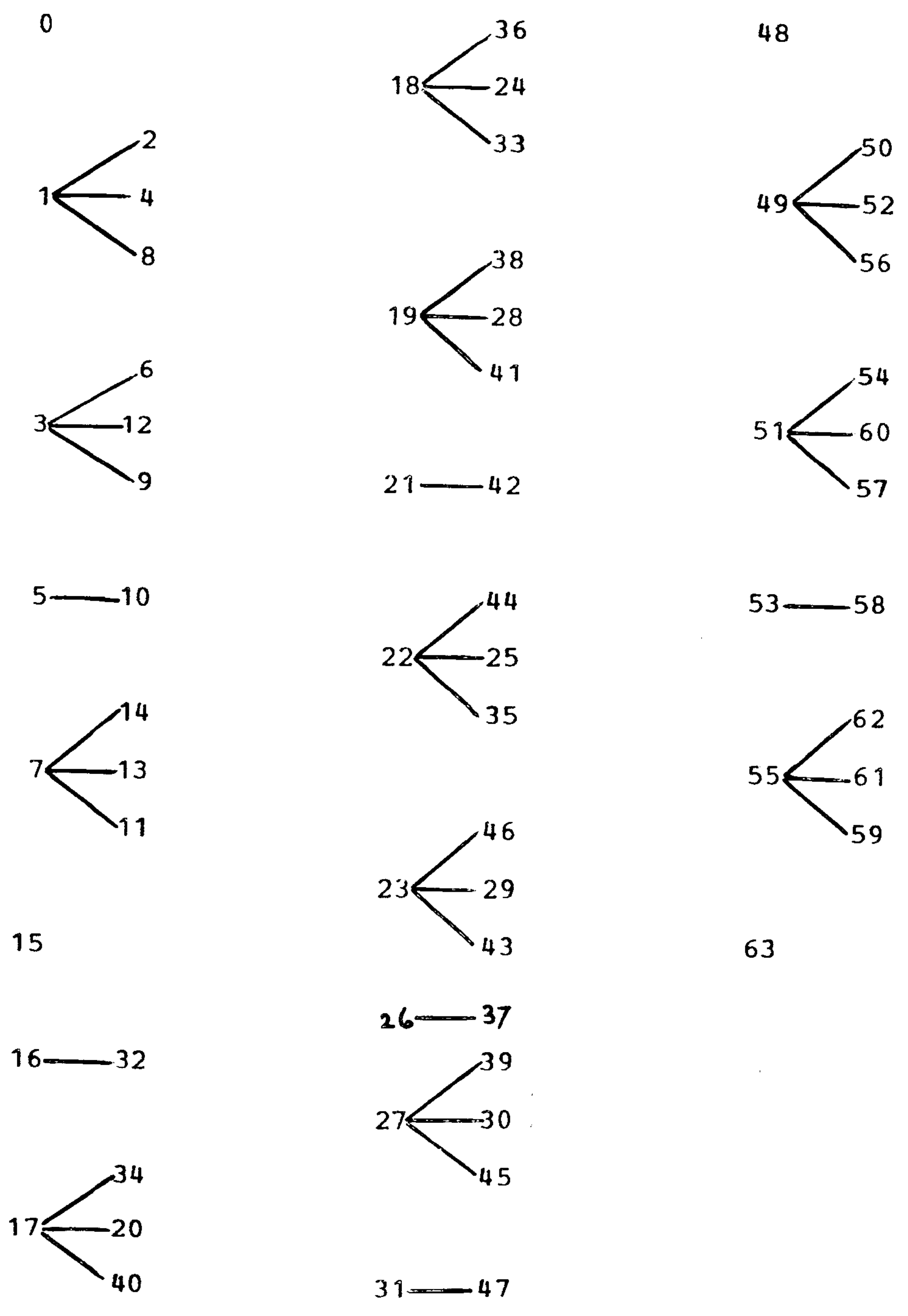
B - (2 1 3 4)

In this example we use the second mapping from points to lines and so the permutations A and B correspond to the permutations

(2 3 4 1 6 5) and (1 5 3 6 2 4)

of the line positions respectively.

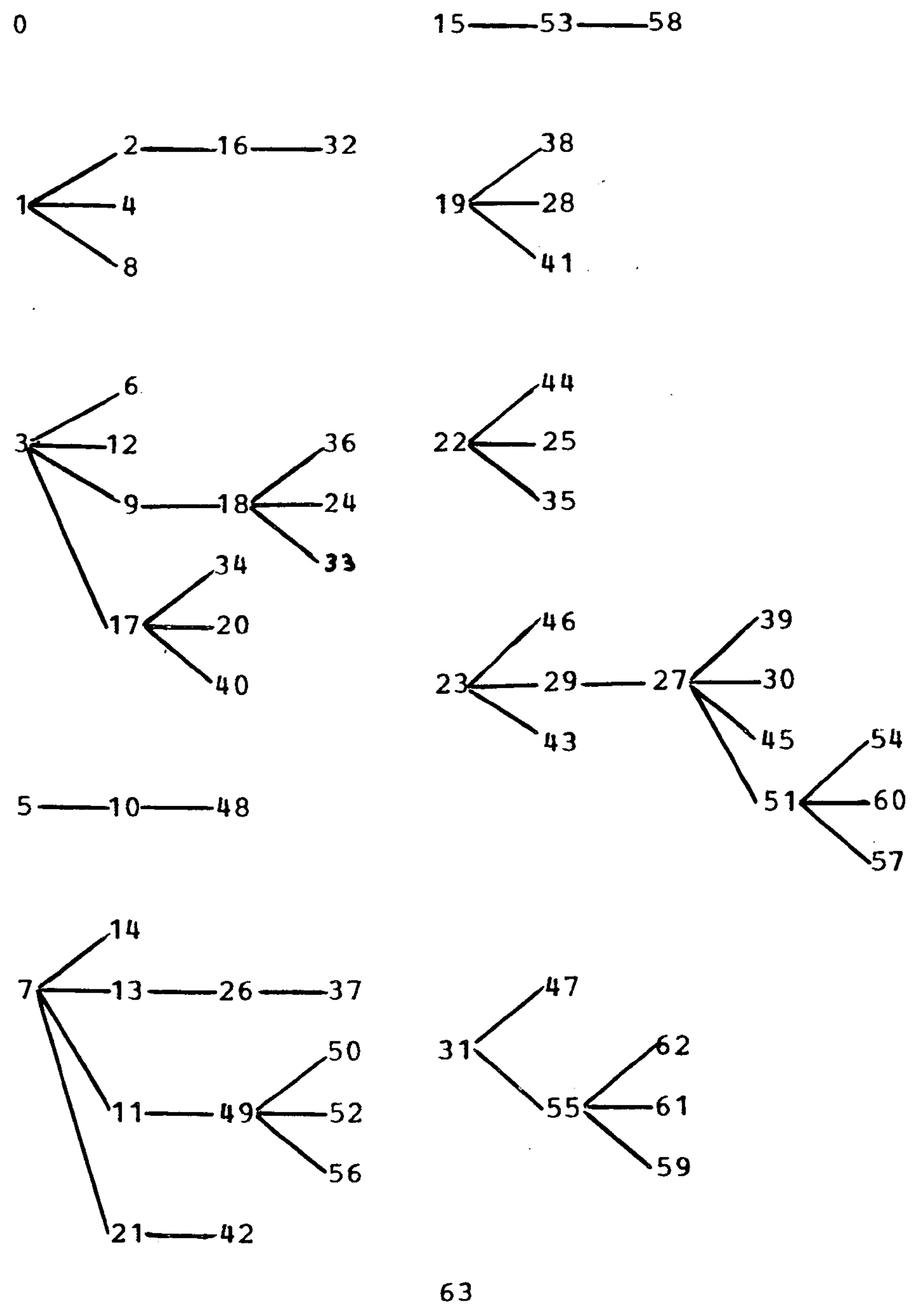
Step 2 of the algorithm, having set all the nodes to be in separate connected components, is to apply permutation A 3 times, or until the original bitstring re-appears. When this operation is complete the nodes are grouped as shown in list I. List II shows the links that are established by step 3, the isolated nodes representing the cases where the permutation B had no effect, and the lines showing the cases where B applied to i gives j and j > i. List III shows the trees as they have been transformed by the operation of setting below(root(j)) = i, where i and j are linked in list II and are at that moment in different trees. We note in passing that the trees (in list I) with roots 3 and 17 become joined by virtue of the link 3 - 17 in list II, and we subsequently discover that there is another link (from 6 to 20) in list II which would have connected these two trees, but by the time this link was discovered, the two trees had already been amalgamated. The roots of the trees in list III are then taken as the bitstring representations of the non-isomorphic graphs with 4 nodes, and these graphs are shown in fig. 5.


0
1 2 4 8
3 6 12 9
5 10
7 14 13 11
15
16 32
17 34 20 40
18 36 24 33
19 38 28 41
21 42
22 44 25 35
23 46 29 43
26 37
27 39 30 45
31 47
48
49 50 52 56
51 54 60 57
53 58
55 62 61 59
63

List I.

0	14——52	31——55
1	15——53	40
2——16	18	41
3——17	19	42——56
4	22	43——57
5	23	44
6——20	24——34	45
7——21	25——35	46——60
8——32	26——50	47——61
9——33	27——51	58
10——48	28——38	59
11——49	29——39	62
12——36	30——54	63
13——37		

<u>List II.</u>



<u>List III.</u>

The Isomorphic Subsets of all Graphs of 4 Nodes. The trees indicate the division of the 64 possibilities into 11 disjoint subsets.

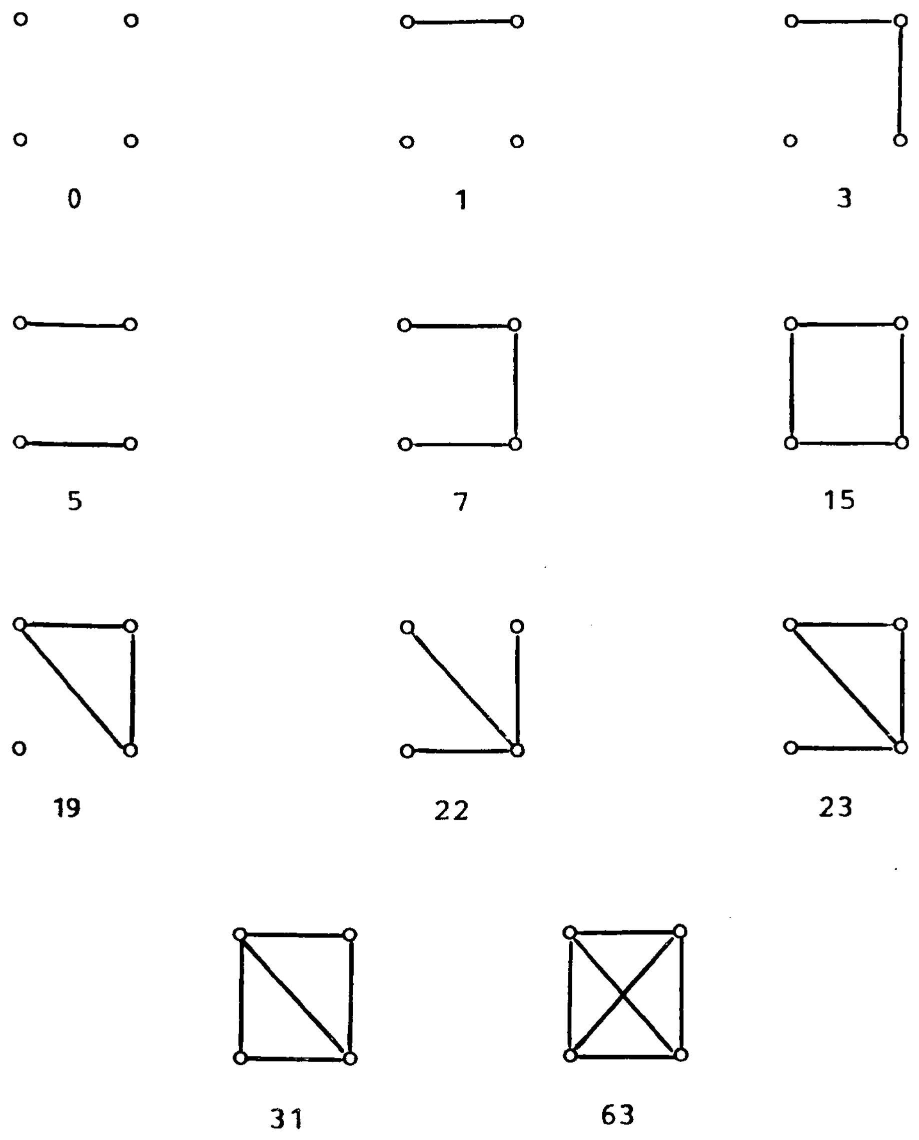


Fig. 5.

This algorithm represents a naive approach to the problem of generating all the graphs of a given number of nodes. In the remaining part of this chapter we attempt to improve on this method. Clearly this algorithm implies the inspection of most (if not all) of the bitstrings in the full set of 2^m (where $m = n(n-1)/2$). In what follows, we attempt to use certain combinatorial techniques to reduce the number of bitstrings which need to be examined.

V.4 Application Of Combinatorial Techniques.

Harary (1955) shows the method by which we may use Polya's theorem to enumerate the equivalence classes, and hence we can calculate the number of non-isomorphic graphs of n nodes. We will paraphrase the arguments used, and show how Polya's theorem operates in the counting of graphs. The explanation given by Liu (1968) of Polya's theory of counting is used extensively.

In fact, the important part of Polya's theory is the use of Burnside's theorem which is a general result for enumerating the number of equivalence classes into which a given set S is divided by a permutation group G of permutations operating on S. To reiterate, two members α and β of a set S are said to be equivalent if and only if there exists a permutation $g \in G$ such that $g(\alpha) = \beta$. Then Burnside's theorem states:

Theorem.

The number of equivalence classes into which S is divided by the group G is

$$\frac{1}{|G|} \sum_{g \in G} \psi(g)$$

where $\psi(g)$ is the number of elements of S which are invariant under the permutation g. (i.e. the number of elements $s \in S$ such that $g(s) = s$).

Proof:

Denote by $\eta(s)$ the number of permutations in G which leave the element s invariant. Then clearly

$$\sum_{s \in S} \eta(s) = \sum_{g \in G} \psi(g)$$

since these are both expressions for the total number of invariances over all permutations g and all elements s.

Now let $\{\alpha, \beta, \ldots, \kappa\}$ be the members of S which lie in one equivalence class. Then for any pair of elements α, β (say), the number of permutations carrying α into β is $\eta(\alpha)$. This can be shown by demonstrating that if g_* is a permutation taking α into β (and there must be at least one since α and β are in the same equivalence class), then

(a) any permutation taking α into β is of the form $g_* g_i$ where g_i is one of the permutations leaving α invariant,

and

(b) for two distinct g_i, g_j which leave α invariant, $g_* g_i$ and $g_* g_j$ are distinct permutations taking α into β.

Now any permutation in G must either leave α invariant, or else it must take α into one of the other elements in the equivalence class. Thus the permutations of G can be classified into those that map α into itself, those that map α into β, etc.

Since the number of permutations in each class is $\eta(\alpha)$, the total number of permutations in G is

G = $\eta(\alpha)$ x no. of elements in the equivalence class.

or

$$\eta(\alpha) = \frac{|G|}{\text{no. of elements in equiv. class}}$$

and since this is independent of α, this must also be the expression for $\eta(\beta)$, $\eta(\gamma)$, etc., and hence

$$\sum_{\substack{\text{all } s \text{ in} \\ \text{equiv. class}}} \eta(s) = |G|$$

Now summing both sides of this equation over all the equivalence classes, we have

$$\sum_{\substack{\text{all equ.} \\ \text{classes}}} \sum_{\substack{\text{all } s \text{ in} \\ \text{eq. class}}} \eta(s) = |G| \times \text{no. of equiv. classes}$$

and since the left hand side is simply a summation over all s S, we have

$$\text{no. of equiv. classes} = \frac{1}{|G|}\sum_{s\in S}\eta(s) = \frac{1}{|G|}\sum_{g\in G}\psi(g).$$

Q.E.D.

We have already described the way in which the relevant permutation group for graphs is constructed from the symmetric group of order n. It is convenient however to divide the set of graphs with n nodes into subsets of the graphs with n nodes and p lines, for $p = 0, 1, 2, \ldots, n(n-1)/2$. Thus, for the purposes of applying Burnside's theorem to this case we have S being the set of bitstrings of length $n(n-1)/2$ which contain exactly p 'one' bits. In addition we have $|G| = n!$.

Making use of the theorem of Burnside, and particularly of the proof, we have, for any given equivalence class consisting of the elements $\alpha, \beta, \ldots, \kappa$,

$$\eta(\alpha) = \eta(\beta) = \eta(\delta) = \ldots = \eta(\kappa)$$

and since $\eta(\alpha) + \eta(\beta) + \ldots + \eta(\kappa) = n!$ we know that n! must be divisible by $\eta(\alpha)$. Furthermore, if we can find the permutations which leave α invariant, and we can find one permutation mapping α into β, then we can find all the permutations mapping α into β by forming the product $g_x g_i$ for each $g_i \in H_\alpha$ (where H_α is the group of permutations leaving α invariant). Regarding this from the other point of view, we can take any graph, and provided we can find its automorphism group, we can find all the graphs to which it is isomorphic by taking each permutation in the set $G - H_\alpha$ and permuting the graph using this permutation. We can then find all the permutations which produce the same graph simply by multiplying this permutation by all the permutations in H_α. When this has been performed $n!/\eta(\alpha)$ times we know that all the graphs in the equivalence class containing the graph α have been accounted for, and so any graph which does not belong to this set is a new non-isomorphic graph.

V.4.1 The Sieve Algorithm.

A full set of graphs with n nodes and p lines is obviously obtainable by placing the p lines in all possible combinations in the $\binom{n}{2}$ line positions, and hence a complete list of all the combinations of p objects from $\binom{n}{2}$ can be taken as the set of all graphs with n nodes and p lines. The problem of finding the non-isomorphic members of this set can now be tackled. The general approach is to lay out the the combinations in a list, and by stages to delete those which are isomorphic to a combination already considered. Thus the method is similar in broad outline to the method of finding prime numbers known as the Sieve of Eratosthanes, in which all numbers are laid out in a list, and as soon as a prime is discovered, the list is passed over once deleting all those numbers which are multiples of that prime. When this can no longer be done, all those numbers in the list which have not been deleted are prime numbers.

We therefore present an algorithm for finding the non-isomorphic graphs with n nodes and p lines, which will be referred to as the Sieve algorithm.

1. Set up a bit map for the $\binom{\binom{n}{2}}{p}$ combinations.
2. Set up a bit map for the n! permutations.
3. Look for the first 'set' bit in the map for the combinations. If there are none, then we have found all the graphs. Otherwise store this graph (as graph 'α' say) and 'unset' its bit.
4. Find all the permutations under which graph α is invariant (i.e. its automorphism group - c.f. chapter IV) and 'unset' the bits in the permutation bit map corresponding to these permutations.
5. Find the first permutation whose bit is 'set'. If there are none, then go to step 2, otherwise let this permutation be g_x.
6. Apply the permutation g_x to the graph α and 'unset' the bit of the resulting graph in the graph bit map.
7. Form the permutation $g_x g_i$ for each $g_i \in H_\alpha$ and 'unset' the corresponding bits in the permutation bit map.
8. Return to step 5.

A few comments on the representations of the various objects used must now be given. Since both the graphs (which are mapped onto the combinations of p lines chosen from a set of $\binom{n}{2}$) and the permutations are to have bit maps associated with them, it is obviously necessary that both the combinations and the permutations should be readily representable as integers, and more precisely as consecutive integers. That is, for both the combinations and the permutations, it is important that the conversion from a combination or permutation to its corresponding integer in some ordering, and vice versa, can be achieved quickly. In the case of permutations, the most convenient method appeared to be a slight variation of the method of Lehmer (1964), and this method gives the permutations in lexicographic order.

In this method, it turns out that it is more convenient to consider the permutations of $0,1,\ldots,n-1$ rather than of $1,2,\ldots,n$ and that the permutations be numbered from 0 up to $n!-1$ instead of 1 to n! The method itself is based on the fact that any number in the range $0,\ldots,n!-1$ can be expressed uniquely in the form:

$$a_{n-1}(n-1)! + a_{n-2}(n-2)! + \ldots + a_1 1! + a_0 0!$$

where $0 \leq a_i \leq i$ for each i.

The mapping from the integer k to the k-th permutation in lexicographic order is then constructed by forming the a_i's for k. From these we can construct the permutation by noticing that the i-th position is occupied by the a_i-th element of those remaining at that stage. We can see this rather better by considering an example. Suppose we wish to find the 63rd permutation of the numbers 0,1,2,3,4, (remembering that (0 1 2 3 4) is the 0-th). The first step is to put 63 in the form

$$2.4! + 2.3! + 1.2! + 1.1! + 0.0!$$

i.e. $a_4 = 2$ $a_3 = 2$ $a_2 = 1$ $a_1 = 1$ $a_0 = 0$ (Notice that a_0 is always zero, by the condition on the a_i). The first position in the permutation is therefore occupied by the element 2, and the second position contains the element 2 in the set 0, 1, 3, 4 i.e. 3. Element 1 of of the set 0,1,4 occupies the next position, and element 1 of the set 0, 4 is the next member of the permutation. The last position is of course filled by the only remaining element, in this case 0, and this is a reflection of the fact that the value of a_0 is always zero. Thus the 63rd permutation of the numbers 0 to 4 is

$$(2\ 3\ 1\ 4\ 0)$$

The reverse mapping is obtained simply by reversing the process completely, reconstructing the a_i's from the permutation and then multiplying out the 'polynomial' to arrive at the integer k.

Similarly a mapping from the set of integers $1,\ldots,\binom{n}{k}$ to the set of lexicographically ordered combinations of k objects from a set of n can also be constructed.

With regard to the multiplication of the permutations, it is clear that the most efficient method of achieving this is to store the multiplication table of the permutation group, but this is obviously impractical since its space requirement is for $(n!)^2$ elements. Thus the multiplication must be carried out directly, but this can be done reasonably efficiently provided the mapping from permutation to integer and vice versa is efficient.

The remaining part of the algorithm which needs to be described in more detail is given as step 4 – finding the group of permutations under which the graph is invariant. We now describe an extension of Corneil's algorithm for finding the automorphism partitioning of a graph (Corneil 1972) to display the automorphism group. The algorithm for obtaining the automorphism partitioning has already been described in the previous chapter, and we present here the extension required to generate the automorphism group.

V.4.2 The Automorphism Group.

Suppose we have a graph whose automorphism partitioning has been found using the partitioning algorithm. Then we have the nodes grouped together in a number of disjoint sets. These sets were used to form the re-ordered graph of the previous chapter, by selecting one node from any set containing more than one node, and placing this node in a class by itself. This new partitioning was then refined using the partitioning algorithm. Now the conjecture is that in the automorphism partitioning, for any pair of nodes within a class, there exists an automorphism mapping one of these nodes into the other. Thus there are $|V_i|$ ways of choosing an element from V_i, assuming that V_i is the first set for which $|V_i| > 1$. The refinement then takes place, having made one such choice, and the process is repeated. The total number of automorphisms in the automorphism group is the product of the number of ways in which a choice may be made whenever a choice is to be made. Hence, if H is the automorphism group:

$$|H| = |V_{i_1}| \times |V_{i_2}| \times \ldots \times |V_{i_p}|$$

where V_{i_j} is the set of nodes, which after j-1 assignments and the corresponding refinements of the partitioning, has $|V_{i_j}| > 1$ but $|V_{kj}| = 1$ for $k < i$.

The actual permutations which make up the automorphism group can of course be found by actually making all the various assignments which are available at each stage, and then carrying out the refinement.

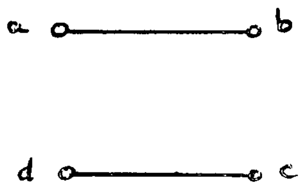


Fig. 6.

An example will show more clearly how this method works. Consider the graph of four nodes shown in fig. 6. The automorphism partitioning is:

I - {a,b,c,d}

Making the first assignment, we have a choice of four nodes, so we choose one of them, say a. After applying the final connection partitioning algorithm, the partitioning is:

I	-	{a}
II	-	{b}

III - {c, d}

It is apparent that if we now make an assigment to resolve the difficulty of set III having two nodes, we must make a choice between these two nodes. Since (by conjecture at least) the partitioning would have the same form whichever of the four nodes a, b, c, d were chosen for the first assignment, we know that there would always have been a further choice between two nodes to be made. Thus the total number of nodes in the automorphism group is 8 (= 4 x 2).

To continue with the example, the second assignment (together with what amounts to a trivial refinement) we have:

I	-	{a}
II	-	{b}
III	-	{c}
IV	-	{d}

This is the result of choosing c for the set III, but we could equally well have chosen d, in which case the partitioning which results would be:

I	-	{a}
II	-	{b}
III	-	{d}
IV	-	{c}

Thus we have formed two permutations under which the graph is invariant and these are:

$$\begin{pmatrix} a & b & c & d \\ a & b & c & d \end{pmatrix}$$

and:

$$\begin{pmatrix} a & b & c & d \\ a & b & d & c \end{pmatrix}$$

We may now write down the six remaining permutations which result when the other three choices are made at the first stage together with the two choices at the second stage in each case:

$$\begin{pmatrix} a & b & c & d \\ b & a & c & d \end{pmatrix}$$

$$\begin{pmatrix} a & b & c & d \\ b & a & d & c \end{pmatrix}$$

$$\begin{pmatrix} a & b & c & d \\ c & d & a & b \end{pmatrix}$$

$$\begin{pmatrix} a & b & c & d \\ c & d & b & a \end{pmatrix}$$

$$\begin{pmatrix} a & b & c & d \\ d & c & a & b \end{pmatrix}$$

$$\begin{pmatrix} a & b & c & d \\ d & c & b & a \end{pmatrix}$$

Changing the notation slightly, so that the nodes a, b, c and d are replaced by the nodes 1, 2, 3 and 4 respectively, and denoting a permutation of n numbers 1,2,....,n by a vector of length n

$(p_1, p_2, \ldots, p_n)$

where p_i is the value of the i-th element after the permutation has been applied, we may write dowm the permutations under which this graph is invariant as:

i	-	(1 2 3 4)
a	-	(1 2 4 3)
b	-	(2 1 3 4)
c	-	(2 1 4 3)
d	-	(3 4 1 2)
e	-	(3 4 2 1)
f	-	(4 3 1 2)
g	-	(4 3 2 1)

This collection of permutations can easily be shown to be a group with the following multiplication table:

	i	a	b	c	d	e	f	g
i	i	a	b	c	d	e	f	g
a	a	i	c	b	e	d	g	f
b	b	c	i	a	f	g	d	e
c	c	b	a	i	g	f	e	d
d	d	f	e	g	i	b	a	c
e	e	g	d	f	a	c	i	b
f	f	d	g	e	b	i	c	a
g	g	e	f	d	c	a	b	i

V.5 Partitions Of Graphs.

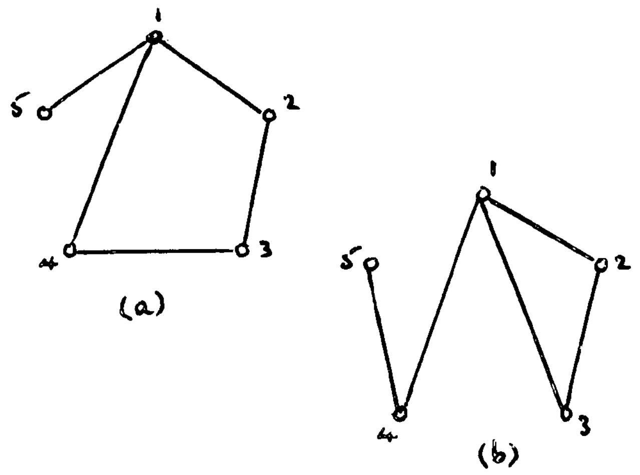
Throughout this chapter we have been concerned with making a selection of objects (in this case graphs) from a larger set of objects such that we have exactly one representative of each equivalence class in the set under some definition of equivalence. To achieve this we require that the set of objects from which the choice is made should contain at least one member of each equivalence class. Ideally, of course, the set from which we make the selection would consist of exactly the set of non-equivalent objects, in which case the selection mechanism required is the trivial one of selecting each object in the set.

Although this was our original aim, we have not been able to achieve this goal, and so we have taken as our starting set the set of all bitstrings of length $n(n-1)/2$. This corresponds to saying that we can take the set of all labelled graphs and deduce those that are non-isomorphic when considered as unlabelled graphs. We have implicitly recognised however that no graph with n_1 nodes can be isomorphic to a graph with n_2 nodes if $n_1 \neq n_2$. The length of the bitstring is then a partial specification on the set of graphs, knowing that all isomorphs must have the same partial specification.

We may now consider the possibility of carrying out a similar process on a set of graphs with a more precise partial specification. This stems from the fact that if two graphs are isomorphic, and σ is the mapping which takes the nodes of one graph onto the nodes of the other, $\sigma x = y$ implies that the degree of x is equal to the degree of y. This is true for all nodes x in the first graph, and so the degree sequences of two isomorphic graphs must be identical, where we define the <u>degree sequence</u> of a graph to be a vector

$$\underline{d} = (d_1, d_2, \ldots, d_n)$$

where d_i is the degree of the node i (in some labelling) and such that $d_i \geq d_{i+1}$ for i = 1,...,n-1. This is also known as the <u>partition</u> of the graph. Two graphs cannot be isomorphic if they have different degree sequences. However, it is not true that two graphs with the same partition are necessarily isomorphic, as can be seen by the example given in fig. 7. These two graphs both have 5 nodes, and both have degree sequence (3 2 2 2 1), but the two graphs are clearly not isomorphic, since for instance the node of degree 3 is adjacent to the node of degree 1 in graph (a), which is not true in graph (b).



<u>Fig. 7.</u>

In the earlier section concerning the Sieve algorithm, the total number of graphs which needed to be considered was $n! \times g_n$, where g_n is the number of non-isomorphic graphs with n nodes. In fact, this was done by finding a graph topologically distinct from any graph previously examined, and then inspecting each of the n! permutations of the labels. An alternative viewpoint is to consider generating all the labelled graphs, and then discarding all those which are not already in canonical form. By this means we would retain only the graphs in canonical form, so that if two graphs in the set are isomorphic, then they are also identical, in which case the problem of removing duplicates is made correspondingly easier.

If we could then generate only the set of graphs which are in canonical form already, the starting set of graphs would also be much smaller. As a first step towards this, we observe that the algorithm for reducing any graph to its canonical form begins by classifying the nodes of the graph by degree. Thus, in the canonical labelling, we know that if node i has degree d_i then $d_j \geq d_{j+1}$ for $j = 1,\ldots,n-1$. This is of course simply the condition on the degree sequence of a graph, so that we may consider attempting to generate the set of all non-isomorphic graphs by examining only those labelled graphs whose labelling gives rise to a valid partition for the graph. We also notice that only a restricted number of the n! possible labellings of a graph give rise to a valid degree sequence for the graph.

We could contemplate generating the set of all graphs by the following algorithm:

1. Generate all degree sequences.
2. For each degree sequence, generate a set of graphs which must necessarily contain at least one instance of each non-isomorphic graph with that partition.
3. Eliminate duplicates from this set.

Harary (1968) quotes two theorems which give necessary and sufficient conditions for a partition to be the degree sequence of some graph, and we will make use of both of these in the course of implementing this algorithm. The first theorem, due to Erdos and Gallai (1960) states:

Theorem

A partition $(d_1, d_2, \ldots, d_n)$ of a number 2m is the degree sequence of some graph if and only if:

$$\sum_{i=1}^{r} d_i \leq r(r-1) + \sum_{i=r+1}^{n} \min(r, d_i)$$

for each r, $1 \leq r \leq n$.

We will not give the proof for this theorem, since the statement of the theorem is sufficient to suggest an algorithm for generating all the degree sequences for graphs of n nodes.

We begin by observing that the partition $(n-1, n-1, \ldots, n-1)$ is valid (and corresponds to the complete graph). The algorithm then uses each degree sequence to generate the next one, the terminating condition being that the elements of the partition are all zero, which may be recognised by the appearance of a zero as the first element of the partition. For each iteration of the main loop, the steps are as follows:

1. Find the smallest j s.t. d_j can be reduced without destroying the ordered nature of the partition ($d_i \geq d_{i+1}$ for $i=1, \ldots, n-1$)
2. Reduce d_j by one.
3. Reconstruct the elements $d_{j-1}, d_{j-2}, \ldots, d_1$ making sure that the partition always satisfies the condition of the theorem.
4. check that the final sequence is a partition of an even number. If so, then this sequence is the required partition otherwise repeat the steps 1 to 4.

Step 4 is required since it is possible to construct a partition of an odd number such that all the conditions (except the evenness condition) are satisfied, but such a partition is not a valid degree sequence, since

$$\sum_{i=1}^{n} d_i = \text{twice the number of lines in the graph.}$$

(This is easily seen since each line in the graph makes a contribution of 1 to the degree of each of its endpoints, and hence a contribution of 2 to the total sum).

The second theorem which Harary quotes concerning the partition of a graph is due to Hakimi (1962):

<u>Theorem</u>

A partition

$$\underline{d} = (d_1, d_2, \ldots, d_n)$$

of an even number 2m is a valid degree sequence for some graph if and only if the partition

$$\underline{d}' = (d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$$

is also a valid degree sequence for some graph.

For this theorem we give the proof, as we shall require some of the techniques used in the proof later on.

<u>Proof:</u>

If $\underline{d}'$ is valid, then clearly so is $\underline{d}$ since from a graph with partition $\underline{d}'$ we may construct a new graph with partition $\underline{d}$ by adding a new node, and joining it to the first d_1 nodes of the original graph, where d_1 is the degree of the new node.

Now suppose that G is a graph with degree sequence $\underline{d}$. If there is a point in G of degree d_1 which is joined to nodes of degree $d_2,\ldots,d_{d_1+1}$, then the removal of this node will give a graph with partition $\underline{d}'$. If there is no such point, then we must show that from the graph G we may construct a graph G' with the same partition which does have such a point. Now in G, node i has degree d_i for $i = 1,\ldots,n$, and $d_i \geq d_{i+1}$ for $i = 1,\ldots,n-1$. In particular node 1 has the largest degree. Now in the graph we may find points i and j such that $d_i > d_j$ and such that line (1,j) is in G but line (1,i) is not. (If this were not so, then we would have the situation above, where node 1 is joined to i for $i = 2,\ldots,d_1+1$). But since $d_i > d_j$, there must a node k such that (i,k) is in G but (j,k) is not. If we now remove the lines (1,j) and (i,k) and replace them by the lines (1,i) and (j,k), the new graph has the same partition $\underline{d}$, but the sum of the degrees of the nodes adjacent to node 1 has increased by 1. It is now possible to repeat this process until eventually we acquire a graph with the desired property.

Q. E. D.

The proof of this theorem highlights two facts:

(a) We have an algorithm for generating one graph with a given partition

and

(b) Any graph having a particular degree sequence can be transformed into a graph with the same degree sequence, but with the property that the node 1 is joined to the nodes 2, $3,\ldots,d_1+1$, and that the degree sequence ordering is maintained.

Now fact (a) indicates the algorithm to be used to generate a graph with the given partition. The method is simply to join node 1 to each of the nodes $2, 3, \ldots, d_1 + 1$, and then construct a graph with partition $(d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, \ldots, d_n)$ on the nodes $2, \ldots, n$.

Fact (b) suggests that since it possible to transform any graph with this partition into this "standard" form, then it is also possible to generate any graph with this degree sequence from the "standard" graph by an appropriate sequence of operations of the type given in the proof of the theorem. Thus, in order to generate the set of all possible graphs with a given degree sequence, we must use the algorithm suggested above to find one such graph, and then define the set of all transformations which are the inverses of the transformations used in the proof of Hakimi's theorem. Naturally, in carrying out these transformations the graphs produced will not all be distinct, but we may either leave the problem of sifting out the duplicates to the third step of the generalised algorithm, or we may sift out the duplicates as we proceed.

The algorithm for finding one graph with the given partition simply joins node 1 to node i and subtracts one from d_i, for $i = 2, 3, \ldots, d_1 + 1$. The remaining nodes are then re-ordered so that the new partition is still ordered correctly. During this operation, the node labels, which were originally 1, $2, \ldots, n$ in that order, are carried with the new degree values in the re-ordering process. When this has been done, the algorithm is repeated on the nodes $2, \ldots, n$ (re-ordered) according to the revised degree sequence.

Let us now demonstrate this with an example in which the number n of nodes is 6, and the degree sequence

$\underline{d}$ = (4 4 3 3 2 2)

this graph will have 9 lines since the sum of the d_i is 18.

The first step is to join node 1 to each of the nodes 2, 3, 4 and 5. The degree sequence which results, after re-ordering, is (3 2 2 2 1) and the new ordering of the nodes is (2 3 4 6 5). If the procedure is repeated, the node 2 is joined to nodes 3, 4 and 6, leaving a partition of (1 1 1 1) and no further re-ordering is necessary. The next step joins nodes 3 and 4, and the final step joins nodes 5 and 6. The final graph is then given in fig. 8.

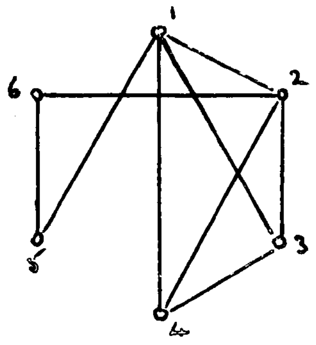


Fig. 8.

The second algorithm, that which constructs every graph with the given partition, is somewhat more complicated. Its action is to reverse the process which was described in the proof of Hakimi's theorem. In fact, it is more complex than Hakimi's construction, since in that case the problem was to move towards a form of the graph which was in some sense optimal. This constructive method moved through a series of steps to achieve its goal. Now for any graph obtained during this iterative process, there are possibly many graphs for which one step of the iteration would give this graph, and in reversing the process, we are faced with the problem of identifying all graphs for which this would be true.

We begin by defining a class of graphs, which will eventually contain all the graphs we are seeking. Initially, it contains only one graph, the graph constructed by the previous algorithm. The algorithm then applies a series of transformations to each graph in the class in turn, and each time a new graph is found, the class is enlarged to include this graph. It turns out also to be a convenient stage at which to check for duplications. A new graph is only inserted in the class if it is not already contained in the class. The way this is checked is to use Corneil's algorithm (see previous chapter) to reduce each graph to its canonical form (as defined previously) before attempting to place the graph in the class. We recall that we conjectured (following Corneil) that two graphs are isomorphic if and only if their respective canonical forms are identical. Thus each graph is put into its canonical form, compared with each graph already in the class, and if it is different from each one, then the class is enlarged to include this new graph, which itself now becomes eligible to be transformed.

The transformation algorithm is as follows:

1. Assume that the initial graph is in canonical form as defined in the previous chapter. In particular this implies that $d_i \geq d_j$ for $i < j$.
2. For each node i in the graph, find nodes j, k, l in the graph such that $i < j < k$, $i < l$ and such that the

lines (i, j) and (k, l) are in the graph, and the lines (i, k) and (j, l) are not.

3. Remove the lines (i, j) (k, l), replacing them with the lines (i, k) and (j, l). This step should be repeated for each 4-tuple satisfying the conditions in step 2.
4. Each graph produced in step 3 is put into canonical form, and compared with each graph in the class. If it is found to be there already, then it is discarded, otherwise it is added to the class.
5. If there are no more graphs in the class to be "developed", then the algorithm terminates; otherwise the next graph is chosen from the class and the algorithm is restarted at step 2.

The conditions on the nodes i, j, k and l as shown in step 2 are explained in the following way:

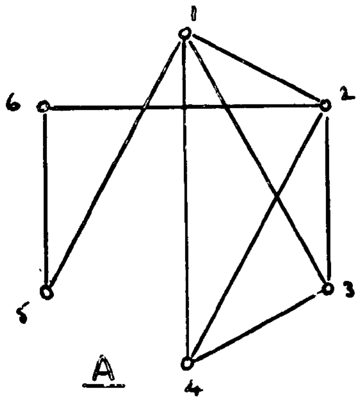
(a) We need to consider all nodes i in the graph, rather than just node 1 as required by Hakimi's proof, because this interchange of lines may need to be done at a different stage of the process of deciding whether a partition is a valid degree sequence. Hakimi's proof implicitly takes account of this by the fact that his argument is recursive.

(b) j, k and l are all greater than i, since for each i we are essentially concerned with the subgraph defined by the nodes i, i+1,...,n.

(c) The condition j < k stems from the fact that we know that $d_j \geq d_k$ when $j < k$, and we certainly want to ensure that this is so, since $d_j < d_k$ is the condition required for the transformation in Hakimi's theorem, and we wish to construct the reverse transformation.

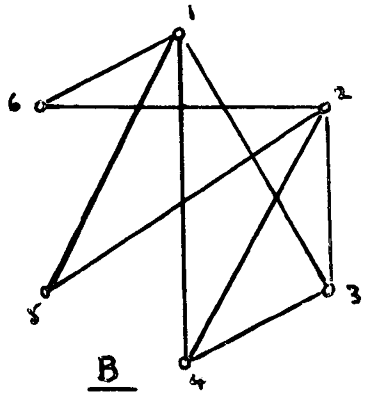
By examination of several examples, it is clear that this algorithm produces the same graphs a number of times over, and that the amount of work required in sifting out the duplicate graphs is more than we would wish. It is also regrettable that some partitions, for which there is only one graph, will cause this algorithm to look in vain for others. One solution would be to make use of some work of Parthasarathy (1968) which shows how to determine the number of non-isomorphic graphs with a given partition. If these numbers were also computed, then it would be necessary to invoke the above algorithm only when a partition is seen to have more than one graph associated with it. It would also be possible, and perhaps desirable, to know in advance the number of graphs which any partition will yield, so the algorithm can be stopped as soon as this number of distinct graphs have been generated.

We conclude this chapter with an example of this algorithm at work. In fig. 9., we show the graph of fig. 8. labelled as graph A, being a graph with partition (4 4 3 3 2 2).. Alongside we give 8 permissible 4-tuples, and the graphs which are obtained by the application of the transformation defined by each 4-tuple. This is done for each graph in the class, and each time a new graph is discovered, its canonical form is placed in the class. The graphs A, B, C, D and E are indeed the canonical forms of the 5 graphs with this partition (see Harary 1968 Appendix 1).

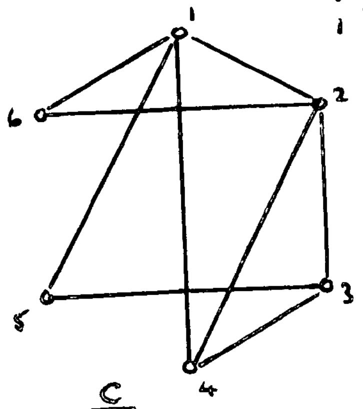


A

i	j	k	l	
1	2	6	5	→ B
1	3	6	5	→ C
1	4	6	5	→ C
1	5	6	2	→ A
2	3	5	6	→ C
2	4	5	6	→ C
3	4	5	6	→ D
3	4	6	5	→ D

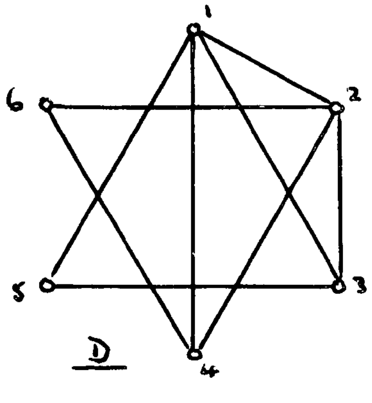


B



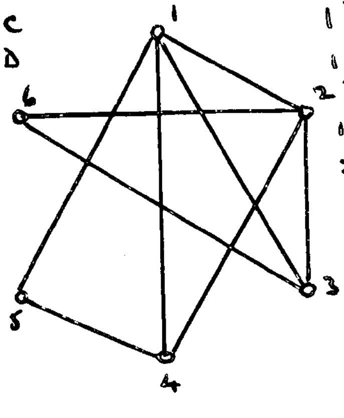
C

i	j	k	l	
1	2	3	5	→ B



D

i	j	k	l	
1	3	6	4	→ C
1	5	6	2	→ D
1	5	6	4	→ E
2	4	5	3	→ E
3	5	6	4	→ E



i	j	k	l	
1	4	6	3	→ C
1	5	6	2	→ D
1	5	6	3	→ E
2	3	5	4	→ C

Fig. 9.

VI Factorisation.

VI.1 General Discussion.

In chapter III, we dealt at some length with the problem of ordering and indexing trees of various kinds, and to this end, a number of ways were discussed for the decomposition of trees into smaller trees. In some sense this may be described as a factorisation of a tree. This decomposition or factorisation of a tree is a useful tool in the derivation of recurrence relations for counting trees. It was considered a natural extension of this to attempt to use the same technique, i.e. decomposition, to study the ordering of graphs. From the previous section, we see that we can in fact generate graphs in some way by use of the Polya theorem approach, with its associated study of permutation groups and similar combinatorial techniques.

However, factorisation of a graph is a subject of study in its own right and we now discuss some of the concepts involved in the factorisation of a linear graph.

A factor of a graph is defined to be any subset of the lines of a graph such that each node is incident to at least one line in the subset. Thus, any partial graph of a graph is a factor. An n-factor is a factor in which each node is incident to exactly n lines in the factor. Certain particular cases of factors are of interest. A 2-factor is a factor in which each node is incident to just two lines - i.e. it consists of a set of disjoint cycles where each node lies on one of the cycles.

If a 2-factor consists of just one cycle or component, it is said to be a Hamiltonian cycle. The problem of finding a cycle which passes through each node of a graph once and once only is known as Hamilton's problem, and it can be seen immediately that this is precisely the problem of finding a 2-factor with just one component. No efficient algorithm is yet known for finding a Hamiltonian cycle of a graph, or even of deciding whether one exists, although several theorems have been proved stating necessary conditions for the existence of such a cycle (see for instance, Harary 1968). Certain heuristic methods can however be used, but these essentially take heuristic 'short cuts' in an algorithm for solving the problem by exhaustive search. Berge (1958) describes a method due to Fortet (1959) for solving the problem for a directed graph, but similar techniques do not lend themselves easily to the solution of the undirected case.

VI.2 Matchings.

Another type of factorisation which is of interest is the 1-factor. This is closely related to the subject of matching in a general graph, and in fact, a perfect matching is a 1-factorisation of the graph.

Berge (1958) defines a matching M of a graph G to be any subset of the lines of G in which no two members of M are incident to the same node of G. A maximal matching is a matching M_m for which

$$|M_m| \geq |M| \text{ for any matching } M \text{ of } G,$$

where $|M|$ denotes the number of lines in the matching M. A matching M, of a graph G is said to be perfect if each node of G is an endpoint of one of the lines of M.

Clearly if G possesses a perfect matching, then n, the number of nodes in G, must be even, and if M is a perfect matching of G, then $|M| = n/2$. If a graph has a perfect matching, then we may attempt to find this matching which covers the graph by using a backtracking technique. This can in fact be done recursively by selecting any line of the graph, removing this line, its two endpoints and all the lines incident to these endpoints. We may then examine this reduced graph for a perfect matching. If it has one, the matching for the whole graph is the matching for the reduced graph together with the line which was removed initially from the whole graph. If the reduced graph has no perfect matching, then we replace the line which was removed initially and try to remove a different line from the original graph. In fact, since each node must be incident to one line of the matching for the graph to possess a perfect matching, it is only necessary to try those lines which are incident to one node. If none of these yield a solution, then we know that this node cannot be covered by a matching line and therefore no matching found by this method could be a perfect matching.

For a large graph, however, this backtracking method will be extremely inefficient, because of the amount of searching which may be required. Given a graph G with a matching M, a node x is said to be covered by the matching if a line belonging to the matching M is incident to x. The graph G is said to be covered by M if each node in G is covered by M. Any node not covered by a matching M is said to be <u>exposed</u>. An <u>alternating path</u> of a graph with respect to a matching M is a path $x_1, x_2, \ldots, x_k$ in G in which if line $x_{i-1}\, x_i$ is not in M, then the line $x_i x_{i+1}$ is in M. Thus an alternating path is a sequence of lines in the graph which are alternately in M and not in M. There is no requirement that this path should be of odd or even length, nor do we require that the first line should be in M or not. However, as can be seen from the following theorem due to Berge (1958), an alternating path of odd length joining two exposed nodes is of particular interest.

<u>Theorem</u>

A matching M of a graph G is maximal if and only if no two distinct exposed nodes are joined by an alternating path.

<u>Proof:</u>

This is not Berge's proof, but a slightly simpler proof due to Edmonds (1965).

Assume we have two exposed nodes x_1 and x_2 which are joined by an alternating path P. Since x_1 and x_2 are exposed, the first and last lines in the path must be non-matching lines (i.e. lines not in M). Thus P consists of an odd number of lines and contains one more non-matching line than it does matching lines. If a new matching is now constructed which consists of the non-matching lines of P together with the lines of M which are not in P, this is clearly a matching M' for which

$$|M'| = |M| + 1$$

Thus M is not maximal.

Now assume that M is not maximal, i.e. there exists a matching M' such that

$$|M'| \geq |M| + 1$$

We have to show that there exists an alternating path which joins two nodes which are exposed relative to M.

Consider a graph G' which has the same vertex set as G, and whose lines are the lines of G which are in either the matching M or the matching M' but not both. No node $x \in G'$ has degree greater than 2 since at most one line in M and one line in M' is incident to x. The connected components of G' are therefore:

(a) isolated nodes,

(b) alternating cycles relative to M (and to M')

or

(c) alternating paths.

(a) and (b) have either no lines or an equal number of lines from M and M'. Therefore at least one of the components of type (c) must contain more lines of M' than it does of M, by the assumption that $|M'| > |M|$. The first and last lines of such a path must be in M' and thus the endpoints of this path are exposed with respect to M.

Q.E.D.

This theorem gives the clue to a better method of finding a perfect matching of a graph. In fact, the backtrack procedure described earlier will find a perfect matching if there is one (albeit rather inefficiently), but says nothing about the matching that is produced if no perfect matching exists.

The method suggested by Berge's theorem is to search for an alternating path between any two exposed nodes. If no such path is found for any pair of exposed nodes then we know (by the theorem) that the matching we have is maximal (and therefore perfect if a perfect matching exists).

This idea has been modified and developed into a working algorithm by Edmonds (1965) and later Witzgall and Zahn (1965) added a further modification. The algorithm is basically a procedure for growing a certain type of tree from an exposed node in such a way that if at any stage a terminal node of the tree is adjacent to another exposed node, then we have an alternating path from that exposed node to the root of the tree.

Edmonds defines this particular type of tree which he calls an alternating tree. An alternating tree is a tree in which the nodes are divided into two classes called inner and outer nodes. Each line in an alternating tree joins an inner node to an outer node, and all the inner nodes have exactly two (outer) nodes adjacent to them. If there are m inner nodes in an alternating tree, then the tree has 2m lines in it. It follows therefore that the total number of nodes in the tree is 2m+1, and that the number of outer nodes is m+1. Thus there is one more outer node in an alternating tree than there are inner nodes.

Although we have defined an alternating tree in general terms, we now go on to describe how we are going to use the notion of an alternating tree to help to find alternating paths within a graph on which a matching is defined.

Given a graph G together with a matching M on the graph, we may consider an alternating tree of which one of the outer nodes is designated as the root of the tree. Now let us grow an alternating tree on the graph G so that the root of the tree is an exposed node of G with respect to M, and that every inner node is joined to one of its neighbours by a line in M. Since the root is an exposed node, i.e. it is incident to no lines of M, each inner node which is adjacent to the root is joined to its other neighbour by a matching line. This then defines how the matching lines are to be inserted into the tree. It is clear that in a tree of this sort, only the root can be an exposed node of G, since if one of the outer nodes is found to be an exposed node of G, we have found an alternating path to the root, and the matching M of G may be enlarged as described in the proof of the theorem. Edmonds describes such a tree as a <u>planted</u> tree, but in view of our previous use of this term, it was felt that the term <u>matching tree</u> would be more appropriate.

The growing of a matching tree within a graph G from an exposed node r with respect to a matching M is in fact a way of finding all the alternating paths from r, and as soon as one of these alternating paths meets another exposed node, we have what is known as an <u>augmenting path</u> for the matching M in G.

In a matching tree rooted at a point r, all the outer nodes correspond to nodes which may be reached in the tree from r by an alternating path of even length. Similarly, the inner nodes are those nodes for which no alternating path of even length from r exists, but only a path of odd length. We therefore prefer to rename the outer and inner nodes of the matching tree as _even_ and _odd_ nodes respectively.

Let us clarify these points using an example. Consider the graph shown in fig. 1. This graph has associated with it a matching and the lines in the matching are marked in the figure. Node 1 is an exposed node with respect to this matching and so we may look for an augmenting path from node 1 to another exposed node in the graph. We therefore grow an alternating tree from node 1 as shown in fig. 2. Now in this tree, nodes 1, 5, 11, 16, 3, 8, 14 and 9 are even nodes, and 2, 12, 15, 4, 6, 13 and 10 are odd nodes. Now node 8 is adjacent to node 7 which is also an exposed node. Thus we have found an augmenting path

(1,2,5,6,8,7)

and this in turn allows us to increase the matching by removing the lines (2,5) and (6,8) from the matching and replacing them with the lines (1,2), (5,6) and (7,8). We also observe that if this change is made to the matching, the new matching is now perfect.

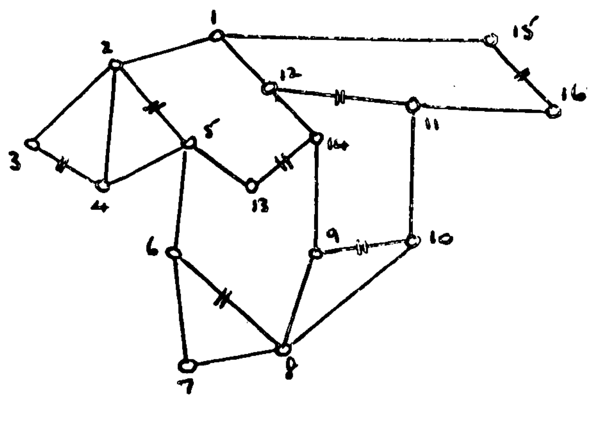


Fig. 1.

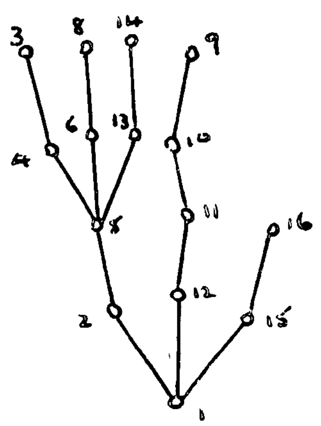


Fig. 2.

The object of Edmonds' algorithm is to find all those nodes x in G with the matching M such that there is an alternating path of even length from some exposed node r. From this we may either find an augmenting path and hence increase the matching, or establish that there is no such path, and hence that the matching is already maximal. A detailed description of Edmonds' algorithm follows.

Starting at the root, we search for any point x which is not exposed, and which is adjacent to the root r. (If there is a node y, adjacent to r, which is exposed, we may immediately put the line (y,r) into the matching). The node x is then put into the tree, and becomes an odd node. The node z such that $(x,z) \in M$ is also put into the tree and z is an even node. All the even nodes in the tree are then developed in this way. For a typical even node p being developed, we find a node x adjacent to p. We now have four cases to consider:

(i) x is an exposed node. In this case, an alternating path from x through p to r has been found, and the appropriate changes may be made to the matching.

(ii) x is not exposed, and has not previously been put into the tree. We now take the step described earlier, calling x an odd node, and z (where $(x,z) \in M$) an even node.

(iii) x is in the tree already and is an odd node. No action is taken in this case.

(iv) x is already in the tree and x is an even node. This is the most interesting case, since this defines a cycle of odd length in the graph. We know that p and x both have even length paths to r. Define the node b to be that node which is common to both the path from p to r and the path from x to r, and such that the path from b to r is of maximum length. This is the point at which the two paths from x to r and from p to r meet, and it may indeed be the root. Now b is an even node, since either it is the root itself, which is even by definition, or else it has three lines incident to it, the lines in the paths from b to the root, p and x respectively. Thus since odd nodes have exactly two lines incident to them we conclude that b must be an even node. Thus by discovering a line from p to x, we have established an odd cycle in the graph consisting of the two paths b to p and b to x, both of which are of odd length, and the line (p,x). Edmonds calls this odd cycle a _blossom_.

If a blossom has been found in a graph, we may consider any node c within the blossom. If c is an even node in the path from p to b, then c is even with respect to the root. If c is odd in the path from p to b, since we have an odd cycle passing through p, b, x and c, there is an even length path c to p to x to b. Thus c is again even with respect to the root by virtue of this path through x. Hence within a blossom, every node can be considered to be even. Thus the tree growing can be continued now considering the blossom as a single even vertex (called a _pseudo-vertex_ by Edmonds).

Witzgall and Zahn (1965) describe a very slight variation of Edmonds' algorithm in which the action taken on recognising a blossom is a little different. In this method, a tree (which may be called the <u>even path tree</u>) is built in which

$$\text{below}(i) = j$$

if and only if there exists a node k with the lines (i,k) and (k,j) in G, such that $(i,k) \in M$ and $(k,j) \notin M$. The step taken when an x is found not in the alternating tree is as before, and has the effect of adding the node z to the even path tree so that

$$\text{below}(z) = p$$

If an odd cycle is encountered in which p and x are the extremes of two even paths, and the line (x,p) is found to be in G, the algorithm of Witzgall and Zahn then chains down the path from p to the root, adding the odd vertices in this path to the even path tree as 'aboves' of x, and chains down the path from x to the root, adding the odd vertices in this path to the tree as 'aboves' of p. If in the chaining down process two adjacent even nodes are found, then this implies that an odd cycle involving these two nodes has previously been found, and so the chaining process stops. If the node b is the point at which the paths p to r and x to r meet as defined before, the chaining process also stops when b is reached, since any point below b is not in the odd cycle.

The algorithm stops if all the even nodes are found to be in state (iii), i.e. all their neighbours are odd nodes (except for members of a blossom). Edmonds calls an alternating tree with this property a Hungarian tree. This may occur before all the nodes of G have been examined. This is overcome by removing the Hungarian tree, together with all the lines incident to nodes in the Hungarian tree, from the graph G. The algorithm is then repeated on the reduced graph. Edmonds and Witzgall and Zahn present proofs that their respective algorithms end with maximal matchings.

The following example shows the operation of the algorithm. Consider the graph shown in fig. 3., in which the matching lines have been marked. We see that node I is an exposed node in the graph. In fact it is the only exposed node in the graph, but this does not affect the performance of the algorithm. Thus I becomes the root of the alternating tree. We first grow towards N. (The stages in the development of the tree are illustrated in fig. 4).

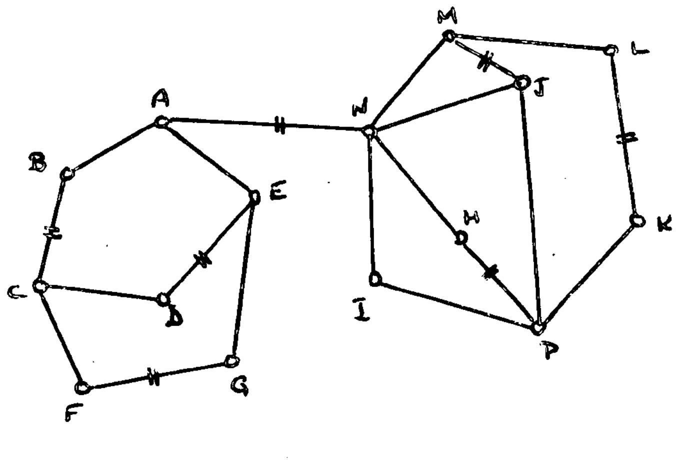


Fig. 3.

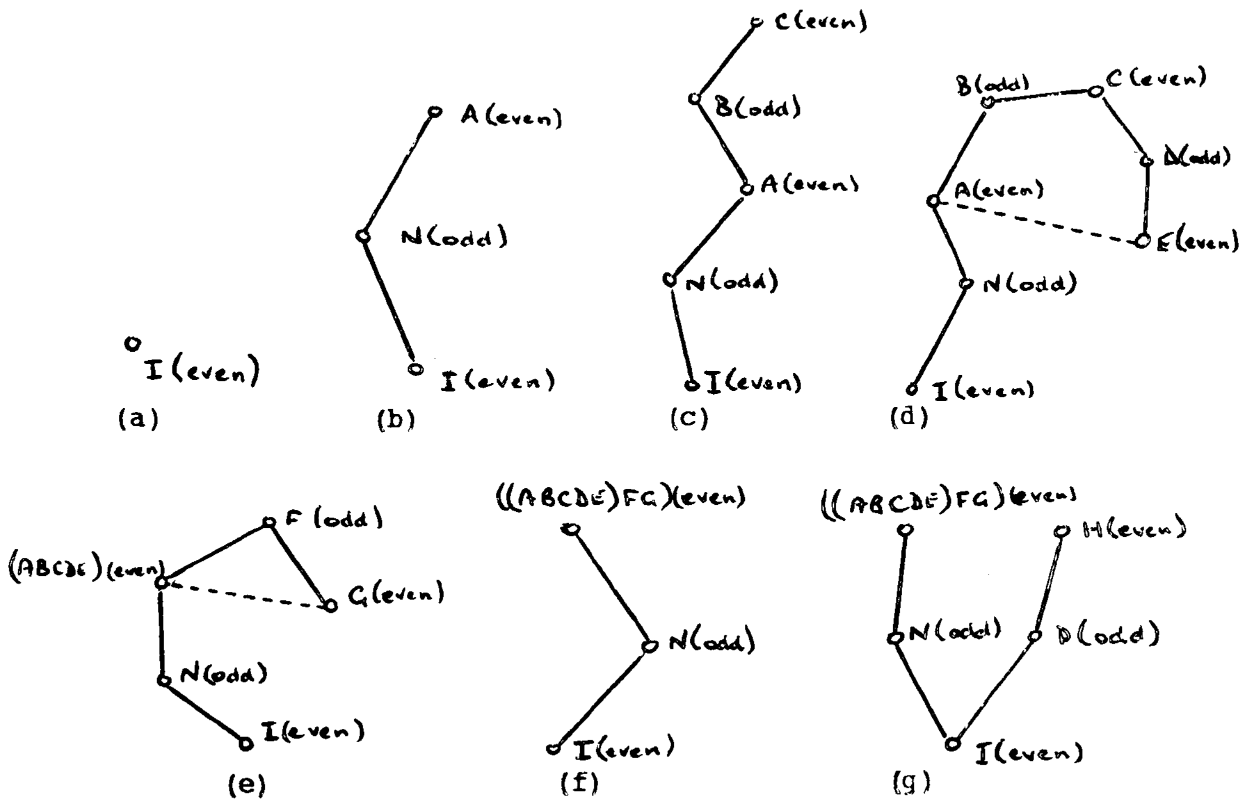


Fig. 4.

Fig. 4. (a) - (c) show how the normal growth of the tree takes place, and a further extension occurs in (d). Two even nodes (A and D) are now found to be adjacent. This blossom can now be contracted into the pseudo-vertex (ABCDE). In (e) another growth takes place and again two even nodes (G and the pseudo-vertex (ABCDE)) are found to be adjacent. The blossom again shrinks into a new pseudo-vertex ((ABCDE)FG). This pseudo-vertex cannot now be developed any further, and so (f) shows that further development of I is the only chance of growing the tree any more. In (g) one further step has been taken, and this tree is in fact Hungarian. If we now remove this tree from the original graph we obtain the subgraph which is shown in fig. 5. This subgraph possesses a perfect matching already, and hence we conclude that there is no alternating path between any pair of exposed nodes, and that the matching as shown in fig. 3. is maximal. This description only discusses the final stage of the algorithm; previous stages have already found augmenting paths and modified the matching accordingly.

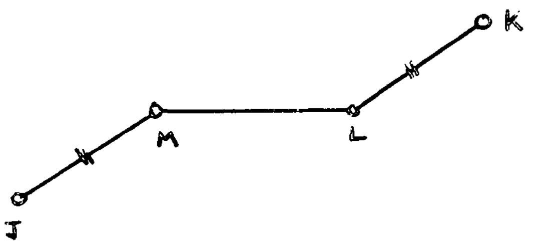


Fig. 5.

A slight extension to Edmonds' algorithm has also been considered. This is to grow alternating trees from each of the exposed nodes of the graph. The conditions under which the "join" of two trees occurs determines whether an alternating path between the roots of any two trees exists.

Briefly, we label all the exposed nodes as even nodes, and grow alternating trees for each one as described earlier. The action to be taken in each of the possible cases is as before, except in case (iv), when two even nodes are to be joined. (If we previously asked the question 'was a particular node in the tree?' we must now ask 'is the node in any tree?'). In case (iv) however we have to subdivide this occurrence into two further parts - when the two even nodes are in the same tree, and when they are in different trees. If the two even nodes are in the same tree, then they must be treated in the same way as before, namely, the process of reducing a blossom to a pseudo-vertex must be carried out. If the two even nodes are in different trees, we have then found an augmenting path between the roots of the two trees. This is of course the even length path from each root, together with the line between the two even nodes which are the growing points. This extension to the method of Edmonds is probably only a marginal improvement over the original algorithm, if any.

VI.3 Articulation Points.

The study of factorisation or decomposition of graphs would not be complete without a mention of articulation points. We make one remark about articulation points in connection with the study of matchings. A node x of the graph G is said to be an articulation point of G if the subgraph G − {x} is not connected, although G is connected. Berge (1958) states and proves the following theorem, which we give here without proof.

Theorem

A node x is an articulation point of a connected graph G, if and only if there exist two nodes p and q in G such that every path from p to q passes through x.

Let x be an articulation point of the graph G, and let $G_1, G_2, \ldots, G_p$ be the connected components of the graph G − {x}. We define p to be the order of the articulation point x (and we note that $p \geq 2$ by definition of an articulation point). The subgraphs $G_i \cup \{x\}$ for $i = 1, 2, \ldots, p$ are called blocks. It can be seen that G has a perfect matching if and only if $G_j \cup \{x\}$ has a perfect matching for some j, and G_i $(i \neq j)$ all have perfect matchings. A necessary condition for a graph G to possess a perfect matching is that $|G|$ is even, and therefore we have:

Theorem

If G is a connected graph, and x is an articulation point of order p which divides G into blocks $G_i \cup \{x\}$ for $i = 1,\ldots,p$, then a necessary condition for G to have a perfect matching is

$$|G_j| \text{ is odd for some } j,$$

and

$$|G_i| \text{ is even for all } i \neq j.$$

Thus, if a graph can be found to possess articulation points, then our backtracking algorithm for finding a perfect matching may be simplified by associating each articulation point with a particular block (according to the rule given in the above theorem), and then looking for perfect matchings in each of the resulting subgraphs separately. It is clear, however, that Edmonds' algorithm is much more efficient than the backtracking algorithm, and will cope with articulation points without further modification, as indeed the example shown earlier demonstrates.

Articulation points are useful in other cases of graph manipulation, and so it is perhaps worth discussing a method of finding the articulation points of a graph.

We stated, without proof, a theorem of Berge, which gave a necessary and sufficient condition that a point x should be an articulation point. Let a spanning tree for a graph G be a tree with the vertex set of G. In other words, a spanning tree is a tree which covers the nodes of G. In any tree, there is exactly one path between any two nodes of the tree, and if the tree is a spanning tree, then there is exactly one path in the tree between any two nodes of the graph. Now if a node x is an articulation point, then, by the theorem quoted above, there exist two nodes p and q such that every path from p to q passes through x, and in particular the path in the spanning tree passes through x. This implies that if a node y ∈ G is a terminal node in any spanning tree of G, then y cannot be an articulation point of G. In particular, if we are able to grow a planted spanning tree (planted tree as defined in chapter I -- not Edmonds' definition) from y which covers G, then y cannot be an articulation point of G. To find the articulation points of a graph G then we may attempt to grow a planted spanning tree from each point of G, and those nodes for which we do not succeed are taken to be the articulation points of the graph. It is also true that for any tree that is grown, all the terminal nodes as well as the root can be marked as non-articulation points, so it is not necessary to grow planted spanning trees from every node in the graph. It is necessary, though, to grow a tree from each node which subsequently turns out to be an articulation point.

It was felt that if some control were exercised over the way in which the tree was grown, it ought to be possible to reduce the number of points from which trees had to be grown. This could be done by attempting to grow a 'bushy' tree, i.e. a tree with as many terminals as possible. However, it turned out that if a tree-growing algorithm was used which tried to do this, nearly all the eliminatable nodes were eliminated in the first few trees, and all the subsequent trees had roughly the same set of nodes as their terminal nodes.

A slight extension to this algorithm can be used to find the articulation points of a graph, and at the same time find the order of the articulation point. (We may perhaps consider a non-articulation point as an articulation point with order 1). Suppose we are trying to find the order of a point x. We construct a spanning tree whose root is x, and in which all the nodes adjacent to node x are inserted at level 1. These nodes are then labelled from 1 to d(x), indicating that at this stage of growing the tree, the node x appears to be an articulation point of order d(x). As each node in the spanning tree is developed we have two cases to consider. Suppose we are developing node i, and we find that node i is adjacent to node j in the graph. Let the label of node i be l(i). Now either:

(i) j is not in the tree. In this case, place j in the tree (by setting below(j) = i or some other means) and set l(j) = l(i).

(ii) j is in the tree. This means that we have found a connection between the block containing i and the block containing j. If l(i) = l(j) then do nothing; otherwise set the labels of all the nodes whose labels are either l(i) or l(j) to be equal to the smaller of l(i) and l(j).

When all the nodes have been developed in this way, we have in fact inspected all the lines in the graph, and the number of distinct labels on the nodes adjacent to x is the order of the node x. If this number is greater than 1, then x is an articulation point of this order.

VI.4 Spanning Trees.

We have briefly touched on the subject of growing spanning trees of various sorts for a given graph. This section is devoted to an examination of the spanning trees of a graph, and various ways of finding a spanning tree.

Obruca (1966) makes extensive use of the spanning tree in the various techniques he develops, but in his case most of the graphs are cost associated, and the extraction of a particular spanning tree is correspondingly easier.

In chapter III, we described various methods of indexing a tree, and in each case we considered the tree as being a composition of two or more parts, usually smaller trees. Our first approaches to the problem of indexing linear graphs were based on attempts to decompose a graph in a similar fashion. To take an example of this, we defined the natural ordering for trees as $T \prec T'$, in terms of the ordering of the components T_1 and T_1', where the tree T is considered to be decomposable as shown in fig. 6. If a comparison between T_1 and T_1' failed to discriminate between the two trees, then the two subtrees T_2 and T_2' were examined, and so on.

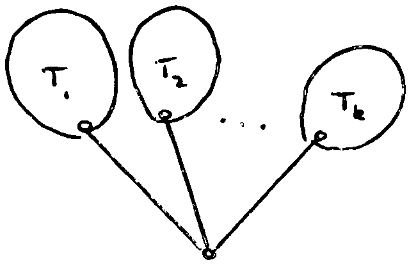


Fig. 6.

In Snow and Scoins (1969) we outlined a similar technique for indexing graphs. It was pointed out in that paper that a large part of the problem was the difficulty of decomposing the graph uniquely. The work of Corneil, as described in chapter IV, has now removed this difficulty (subject to the conjectures given in that chapter), and so the problem of deciding the truth or falsity of the statement $G \prec G'$ is now theoretically solved. For instance, the two graphs could be reduced to their respective canonical forms, and a straighforward comparison of these graphs such as the comparison of their adjacency matrices would give the required result. We have however developed a number of techniques for the practical decomposition of graphs.

The first problem which was mentioned in Snow and Scoins (1969) was that of finding the centre of a graph. However, those nodes which map onto node I of the representative graph may be considered to be the centres of the graph, and since we know that these nodes form a transitive subgraph, any one of these may be taken as the (unique) centre of the graph. The degree partitioning was used as the basis for all the partitionings in the formation of the representative graph, and therefore the use of the Cayley ordering is suggested for deciding which spanning tree is to be regarded as the 'best' spanning tree for a particular graph. Spanning trees of a graph are essentially free trees, and it is therefore necessary to take some point in the tree as the root. A better choice is to choose the root of the spanning tree by virtue of its status in the graph rather than by virtue of its status in the tree (although we would hope that they are connected when we come to create the tree). Since we already know that the representative graph and the re-ordered graph are invariant under isomorphism, any algorithm will suffice to produce a unique spanning tree for a graph provided it is based on the labelling of the re-ordered graph. We have, however, made an attempt to find an 'optimal' spanning tree for a graph, where optimal was taken to be a variety of properties, but mainly the property of 'coming earliest in the natural ordering of trees'. As we saw in chapter III, we can equate the ordering which we have called the natural ordering with the ordering defined by the lexicographical ordering of the height vectors, and it is with this in mind that we attempt to find the 'optimal' spanning tree for a graph.

Given any graph, we may compute from its adjacency matrix, another matrix, the shortest distance matrix

$$S = (d_{ij})$$

where d_{ij} is the length of the shortest path in the graph from node i to node j. Further, we define (following Obruca 1966) a <u>mushrooming spanning tree</u> as any spanning tree in which the length of the path from any node x to the root of the tree is the same as the length of the shortest path within the graph between the same two nodes. Now since the height vector of a tree is simply an ordered list of the distances from the root to each node of the tree, it is clear that the height representation of any mushrooming spanning tree will be simply a re-ordering of the elements of the row in the shortest distance matrix corresponding to the root. Although we have been unable to prove the statement, it seems intuitively reasonable that if the optimal spanning tree is defined in terms of the natural ordering of the trees, and therefore of the height representation, then the optimal spanning tree will be a mushrooming one. The trees which come first in the natural ordering are those with the smaller numbers in their corresponding height vectors, and the mushrooming tree is also an attempt to keep the height vector values as small as possible.

VI.4.1 Two Algorithms.

Two main algorithms were used in an attempt to discover the optimal spanning tree for a graph, and certain variations were also incorporated in certain cases. Unfortunately it was not generally possible to determine whether a tree was in fact the optimal spanning tree when either algorithm had discovered it. The two algorithms are slightly different in approach; the first being an iterative method, known as the Improvement Algorithm, and the second being a more direct method.

The Improvement Algorithm begins by taking any mushrooming tree of the graph. This may be constructed in a number of ways, of which the simplest is to grow from the root all the nodes at a distance 1 from the root. Then from each of these nodes are grown all the nodes at a distance 1 away which have not already been put into the tree. This process is repeated, building up the tree by 'levels', and a mushrooming tree certainly results. Clearly the tree which results is, however, highly dependent on the original labelling of the graph.

Given any spanning tree of a graph, we define the graph which results when all the spanning tree lines are removed to be the co-tree. It is obviously possible to create a list of the lines in the co-tree. The algorithm now proceeds as follows:

Consider any line in the co-tree which joins the nodes x and y. Since the spanning tree is mushrooming, we know that for any such nodes, $|h(x) - h(y)| \leq 1$. This is because if two nodes are joined in a graph, there exists a path from some point r to x which passes through y, and y is the last node on this path before reaching x, thus $d(r,y) \leq d(r,x) + 1$. Now if a spanning tree exists which does not contain the line (x,y), then both the path from r (the root of the tree) to x through y, and the path from r through x to y must exist. Thus $d(r,y) - 1 \leq d(r,x) \leq d(r,y) + 1$, and $d(r,x) - 1 \leq d(r,y) \leq d(r,x) + 1$.

The improvement algorithm attempts to replace lines in the spanning tree by lines from the co-tree to see whether a better tree is formed. This is done by successively trying each co-tree line and removing one of the lines in the spanning tree. If at any stage an improvement is made, then this is adopted as the new spanning tree and the algorithm restarted. When each line in the co-tree has been tried, and no improvement has been made, then the algorithm terminates. This method may be thought of as a kind of 'hill-climbing' technique in which the step length is one line. There is clearly a great danger of a 'local optimum' being found which can only be overcome by replacing two or more lines in the tree simultaneously. This is an obvious extension which could be made to the algorithm.

In fact, this particular method in which the spanning tree is changed as soon as an improvement is made is only one of a number of possibilities. An alternative strategy would be to make all permissible changes to the tree, and select that which gives the greatest improvement. This means that the method would be a sort of 'steepest ascent' method.

Apart from the extension mentioned above, there are other extensions which may also be considered. For any spanning tree, we may form a certain number of different spanning trees depending on the lines in the co-tree. The algorithm above takes the first of these to be found which is an improved tree, and the previous tree is immediately discarded. The alternative method is to take all the trees given by substitution of one line and choosing the best of them. We may however consider each of these trees which represents an improvement, developing each in turn, and seeing which one leads to the best tree by a sequence of one line replacements.

Such a process could be restricted so that each step forward leads to an improved tree, or perhaps so that each step forward leads either to an improved tree or an equivalent tree. In the limit, the method could develop every new tree that was found, in which case the algorithm would generate all the mushrooming trees which span the graph. These last methods are clearly using a backtrack type algorithm.

Since the algorithm as it stands begins with a mushrooming spanning tree, and stays within the set of mushrooming trees, we may make some remarks about the choice of lines to be inserted into or removed from the current best spanning tree. First, assume that we have a line (x, y) in the co-tree. Suppose that within the spanning tree $h(x) = h(y)$, i.e. in the graph $d(x, r) = d(y, r)$ where r is the root of the tree (and perhaps the centre of the graph). Then this line need not be considered as a candidate for inclusion since if it were put into the tree, either $h'(x) = h'(y) + 1$ or $h'(y) = h'(x) + 1$ (where $h'(x)$ is the height of the point x in the new tree). This therefore contradicts the assumption that the tree is mushrooming. We now know that the only lines which we need to consider inserting are those for which $|h(x) - h(y)| = 1$. We may assume, without loss of generality, that $d(r, x) + 1 = d(r, y)$. Thus we have the situation as shown in fig. 7. Now by inserting a single line in a tree, we create one simple cycle. It is now necessary to remove one line from this cycle in order to make this graph into a tree again. The proposition is that the only line which may be removed so that the mushrooming property is preserved is the line (y, z), where $z = \text{below}(y)$. For consider the removal of any other line in the path from y to the root. Then the node z would be connected to the root only by a path passing through y and x (in that order). In the mushrooming tree, since $z = \text{below}(y)$, we have $h(z) = h(y) - 1$. However, in the new tree, the path from z to the root is through y, and therefore $h(z) > h(y)$. Hence the mushrooming property is destroyed. Similarly we can argue that no line in the path from x to the root may be removed and the tree remain mushrooming. The task of testing co tree lines to see whether their inclusion in the tree would make an improvement is now considerably eased, since only a subset of the co-tree lines need be tested, and when one such line is inserted, there is only one line which may be removed.

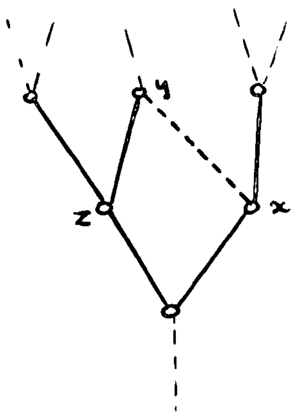


Fig. 7.

The second algorithm, which attempts to find the best spanning tree directly, is based on some of the work on matchings described earlier.

Although initially we had hoped that this algorithm would indeed find the best spanning tree directly, it was found that this goal could not be achieved, and so it was decided that this algorithm would be suitable for finding a 'near optimal' tree upon which the improvement algorithm could operate.

Using this technique, an attempt is made to distribute the nodes of height p + 1 as evenly as possible among the nodes of height p. We are growing the tree by stages, increasing the height at each stage, where at the p-th stage we have a partially constructed tree of maximum height p, and a collection of nodes to be attached at height p + 1.

Suppose we have grown a spanning tree up to and including height p. The next step is to attach all those nodes which are at a distance p+1 from the root. Thus a bi-partite graph B may be defined in which the two sets of nodes are respectively the set of nodes at height p (already in the tree) and the set of nodes at a distance p+1 from the root. Let these two sets be S_p and S_{p+1} respectively. Now if $x \in S_p$ and $y \in S_{p+1}$, there is a line in B if and only if there is a line (x,y) in G. The general method adopted to distribute the nodes of height p+1 as evenly as possible amongst the nodes of height p is to assign one node from S_{p+1} to each node in S_p, as far as this is possible, and when this has been done, all the assigned nodes are removed from S_{p+1} and the process repeated. Thus we allow each node at level p to acquire at most one node from the set of nodes at level p+1 before any node is allowed to acquire a second node from S_{p+1}.

It is obviously desirable to assign as many nodes from S_{p+1} to nodes from S_p as possible, subject to the restriction that no more than one level p+1 node is to be assigned to any level p node at each step, and this problem is then simply the problem of finding a maximal matching on the bi-partite graph B.

In section VI.2 we described Edmonds' algorithm for finding a maximal matching on a general graph. However, we may simplify this algorithm when looking for a maximal matching on a bi-partite graph by observing that in a bi-partite graph every cycle is of even length. Thus in the description of the actions to be taken when growing an alternating tree from an exposed node, cases (i), (ii) and (iii) are exactly as before, but case (iv) cannot occur, since this would imply the presence of an odd cycle in the graph, which is impossible in a bi-partite graph.

Thus we may find a maximal matching on B by growing an alternating tree from each exposed node in S and altering the matching until it is maximal.

Having found a maximal matching on B, we now attach these nodes in S_{p+1} to their corresponding nodes in S_p, so that if $x \in S_p$ and $m(x)$ is that node in S_{p+1} such that $(x, m(x))$ is a line in the matching, the spanning tree of G is enlarged by setting

$$\text{below}(m(x)) = x$$

and the node $m(x)$ and all its incident lines are deleted from the graph B. When this has been done for each line in the matching, a new maximal matching is found on the reduced graph B' and more nodes can then be inserted into the spanning tree. This process is continued until all the elements of S_{p+1} have been placed in the spanning tree.

In passing, we observe that the method of growing an alternating tree on a bi-partite graph may be used to prove by a graph theoretic method a theorem due to Philip Hall (1935) known as the theorem of Distinct Representatives.

Assume we have a set of objects S, and n subsets S_i $(i = 1,\ldots,n)$ of S. We assume that the union of the subsets S_i is the whole set S. Then we have a set of distinct representatives if one element $a_i \in S_i$ can be found for each i, such that $a_i = a_j$ if and only if $i = j$. The element a_i may be said to represent the subset S_i. Then Hall's theorem states:

<u>Theorem</u>

The subsets S_i $i = 1,\ldots,n$ of a set S possess a set of distinct representatives if and only if each set of k of the subsets has at least k distinct elements.

Now let each subset S_i be represented by a node $x \in X$ in a bi-partite graph B, and let each element in S be represented by a node $y \in Y$ of B. Then the lines of B represent the set inclusion of the elements of S in the subsets S_i, i.e. B contains the line (x, y) if and only if the element of S represented by y is contained in the subset of S represented by x. Now for any subset A of nodes of X, let $\Gamma(A)$ be the set of nodes y which are adjacent to at least one node in A. Then we may restate the theorem of Hall as:

Theorem

For a bi-partite graph B, there exists a matching which covers X if and only if for any subset $A \subseteq X$

$$|\Gamma(A)| \geq |A|$$

Proof

Clearly if we have a matching M which covers X, then for any subset A of X, $\Gamma(A)$ must contain at least those points $y \in Y$ such that $(x, y) \in M$ for each $x \in A$. Hence $|\Gamma(A)| \geq |A|$.

Now suppose $|\Gamma(A)| \geq |A|$ for each subset $A \subseteq X$, and suppose that we have a matching M which does not cover X. Then there is a node $r \in X$ which is exposed with respect to M. Now construct an alternating tree as follows. Let r be the root of the tree and join to r all those nodes in Y which are adjacent to r. (There must be at least one since $|\Gamma(\{r\})| \geq |\{r\}| = 1$). If any of these nodes are exposed, then we may immediately increase the matching by including the line joining r to this node. Otherwise, include in the alternating tree those nodes $y_i \in Y$ which are adjacent to r, and the nodes of x which are joined to the y_i by a matching line. We may continue adding y nodes and their matched x nodes to the tree until either:

(i) we include an unmatched y node in the tree; this now gives an augmenting path from r to this unmatched y node, and the matching may be enlarged by one line as before.

(ii) no new nodes can be included in the tree. In this case, we have some number of y nodes in the tree and for each of these there is a matching x node. The total set of x nodes in the tree is therefore the set of the matching nodes together with the node r. Thus the set of x nodes in the tree has one more element than the set of y nodes and hence the assumption that $|\Gamma(A)| \geq |A|$ is contradicted for this subset A of X.

If case (i) occurs, the process is then repeated until all the nodes of X are matched. Thus if $|\Gamma(A)| \geq |A|$ for every subset $A \subseteq X$ we can construct a matching which covers X.

Q.E.D.

Since the method of growing the spanning tree for a graph expands the tree level by level, and makes no reference to the previous structure of the tree, it is clear that counter-examples can be constructed to disprove the conjecture that trees constructed by this method are optimal spanning trees, but in many cases this does seem to be true, and in the cases where it is not, it appears that the tree constructed by this method is 'close' to the optimum in some sense. Thus it would seem to be a good starting tree from which to apply the improvement algorithm. It is also possible to modify the algorithm to take account of the counter-examples which have been found, but unless a general method can be found which can be proved to work, it is felt that piecemeal alterations will only push the point of failure of the algorithm back, rather than eliminate it altogether.

In the final section of Snow and Scoins (1969) a rather different approach to the problem of the optimal spanning tree was hinted at. It was observed that if the best spanning tree is a mushrooming tree, then its height vector is simply a re-ordering of any vector representing the distances from the root of the tree to every other node in the graph, i.e. the appropriate row of the shortest distance matrix S. Now clearly it is possible to take such a vector and rearrange it in such a way that it is bound to be optimal. It is not then certain however that this re-ordered vector represents a spanning tree of the graph. Thus it should be possible to move from the optimal re-ordering of this vector until a vector is found for which a corresponding tree does exist as a spanning tree. In this way we could be sure that the tree found was in fact optimal. In a way, this is rather like the dual approach to a linear programming problem in which we either start with a feasible solution and try to make it optimal also, or else we begin with a solution
which is constrained to

be optimal, but which lies outside the feasible region, and gradually change the solution until it is also feasible.

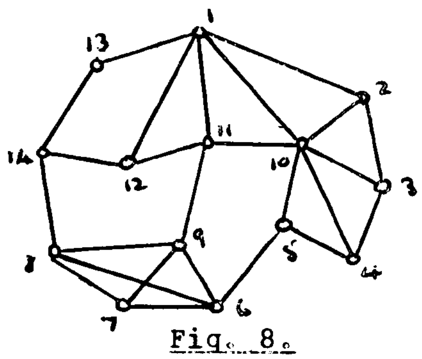


Fig. 8.

We shall show an example of a graph with these three methods operating upon it. The graph to be considered is shown in fig. 8. Let us assume that the initial mushrooming spanning tree is that shown in fig. 9(a). This is clearly a function of both the labelling of the graph, and the algorithm used to construct the tree. For the purposes of this example, any mushrooming spanning tree would be suitable and the tree in fig. 9(a) is just one of these. The height vector representation of this tree is:

0 1 2 3 3 1 2 3 2 1 2 1 1 1

and the labelling of the tree is shown in the diagram. The lines in the co-tree are:

1 - 2, 1 - 11, 1 - 12, 2 - 3, 3 - 4, 4 - 5,
6 - 9, 7 - 8, 7 - 9, 8 - 9, 8 - 14, 13 - 14.

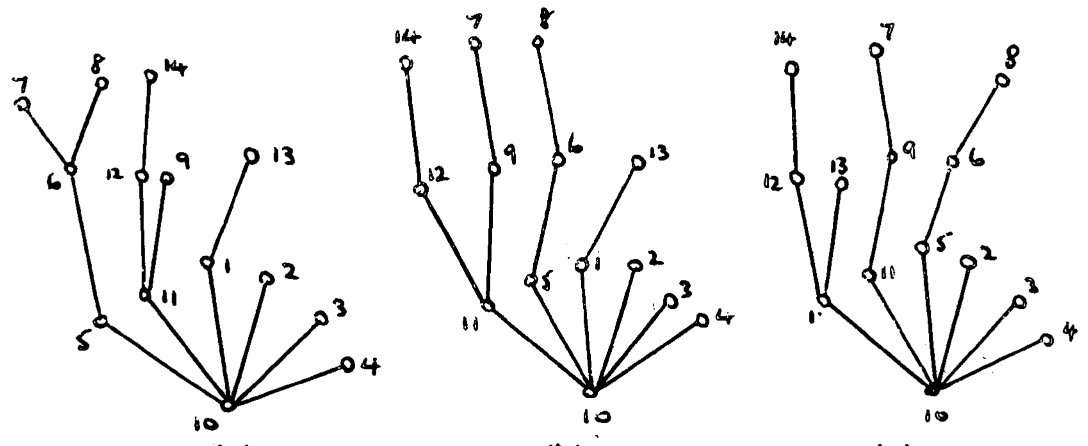


Fig. 9.

Of these only four lines, 1 - 12, 7 - 9, 8 - 9 and 13 - 14 satisfy the requirement that h(x) = h(y) + 1. By trying to insert the line 1 - 12 into the tree we find that the line to be removed is 11 - 12. If this is done the tree which results has the same height vector as the original tree, and therefore the previous spanning tree is retained. The next co-tree line which can be tried is the line 7 - 9. If this is done the line to be removed is 6 - 7, and the resulting tree is shown in fig. 9(b). This tree has height vector :

0 1 2 3 2 3 1 2 3 1 2 1 1 1

which is less than the previous height vector according to our earlier definitions. The list of lines in the co-tree is now recomputed, and in fact it is the same list as before except thatthe line 7 - 9 is replaced by the line 6 - 7. The whole list is scanned again and now we see that the line 1 - 12 can be inserted to make an improvement. This time the line 11 - 12 is to be removed, and if this is done, the resulting tree is as shown in fig. 9(c) and this tree has height vector

0 1 2 3 2 1 2 3 1 2 3 1 1 1

This turns out to be the final tree, as each of the new co-tree lines are tried but without any improvement in the spanning tree.

In the example shown here, it is relatively easy to verify by inspection that no better spanning tree can be found. However we are unable to say that this algorithm will always produce an optimal spanning tree, unless the algorithm is extended to produce <u>all</u> spanning trees of the graph and select the best.

Let us now consider the action of the matching algorithm on the same graph. We have assumed in the previous example that node 10 is to be regarded as the centre, since it has degree 6, which is greater than the degree of any other node in the graph. Thus the partitioning algorithm will put node 10 into class I, and the centre of the graph may be selected on this basis. Now there are 6 nodes adjacent to node 10 and they are nodes 1, 2, 3, 4, 5 and 11. These nodes can be attached immediately to node 10 which has already been designated as the root of the spanning tree. We can now look at the sets of nodes at a distance 2 from node 10, which are connected to nodes at a distance 1.

1	-	{12,13}
2	-	$\emptyset$
3	-	$\emptyset$
4	-	$\emptyset$
5	-	{6}
11	-	{9,12}

These connectivity relations are displayed by the bi-partite graph in fig. 10. and we see that a maximal matching may be found containing the lines (1,12), (5,6) and (11,9). The nodes 12, 6 and 9 are then attached to the spanning tree and removed from the bi-partite graph. A maximal matching on the reduced bi-partite graph contains the one line (1,13) and so the node 13 is inserted in the spanning tree with node 1 as its below.

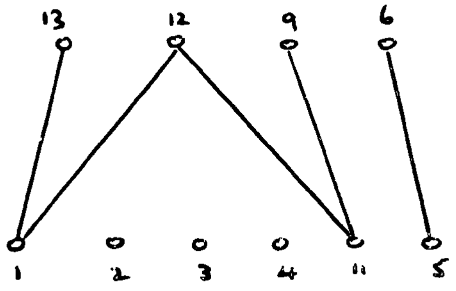


Fig. 10.

At the second level the growing points are the nodes 12, 13, 6 and 9, and the corresponding sets of the level 3 nodes are:

12	-	{14}
13	-	{14}
6	-	{7,8}
9	-	{7,8}

Node 14 is attached to node 12, 7 to 6, and 8 to 9, and the final tree is as shown in fig. 11. It will be observed that this tree has the same height sequence as the tree shown in fig. 9(c) although the labelling is different.

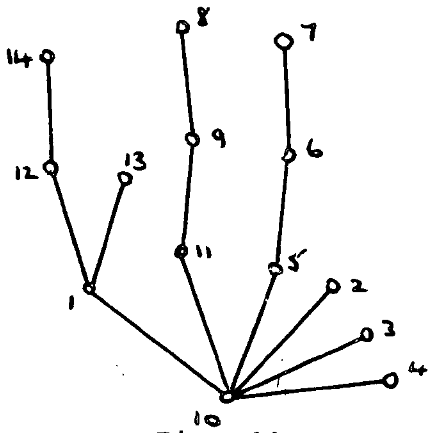


Fig. 11.

Finally we examine the row of the matrix S which corresponds to the node 10. This is:

1 1 1 1 1 2 3 3 2 0 1 2 2 3

The optimum way in which this can be arranged into a valid height sequence is:

0 1 2 3 1 2 3 1 2 3 1 2 1 1

but unfortunately this cannot be fitted to any spanning tree of the graph. This sequence may be constructed by distributing the elements with value p+1 evenly amongst the elements with value p consistent with the sequence remaining a valid height sequence. The problem comes when we attempt to discover whether a particular sequence has a corresponding spanning tree in the graph. This can be achieved by considering all possible assignments of nodes at a particular distance from the root to the elements of the vector with that value. However, this could involve a considerable amount of work, and thus it cannot be considered as a practical method. This optimal sequence can however be thought of as a 'lower bound' on the possible spanning trees, and as such may be used to restrict the searching done by the improvement algorithm. The next smallest vector which can be constructed from the tenth row of the shortest distance matrix is:

0 1 2 3 2 1 2 3 1 2 3 1 1 1

which is the same as the height vector for each of the two previous optimum trees. It is now a matter of making an assignment to the values in this height vector so that it represents a real spanning tree in the graph. In fact we have already seen that there are at least two trees which satisfy these requirements.

VI.4.2 The Structure Tree.

A rooted tree may be considered as a representation of a hierarchy. It is therefore not surprising that the decomposition operations discussed earlier in this thesis should display an hierarchical or recursive structure. By contrast, a graph is essentially non-hierarchical and non-recursive in the representations so far considered. This section is intended to investigate the possibilities of representing a graph by some hierarchical structure. The general approach is to decompose the graph (which we assume to be connected) by some convenient rule, and continue the decomposition recursively on the component parts. The representation takes the form of a tree in which each node is labelled to indicate the exact form of the decomposition.

The decomposition operation is the operation of removing a spanning tree as described in the previous sections, and this spanning tree will then be used as the label representing the decomposition. The connected components of the co-tree are each treated as a separate connected graph and are decomposed similarly. In fact, the methods of extracting the optimal spanning tree described here ensure that the co-tree is disconnected, since all the lines from the root are put into the spanning tree, and the root itself is left as an isolated node in the co-tree. This gives rise to the concept of a <u>structure tree</u> for a graph. The graph is decomposed into a spanning tree and a co-tree. The root of the structure tree represents this decomposition, and the spanning tree which is extracted is used as the label for the root of the structure tree. The root of the structure tree has m nodes above it, where m is the number of connected components in the co-tree. If any component is an isolated point, the corresponding node of the structure tree is not developed, otherwise this node is the root of the structure tree for this component. Thus we arrive at a structure tree in which each internal node is labelled with a spanning tree, and each terminal node represents a single node of the original graph.

An example will help to show how the structure tree is formed. By some means, we find the centre of the graph, and then some algorithm such as one of the algorithms described earlier is applied to find a spanning tree. Let us consider the decomposition of the graph G shown in fig. 12, and the first spanning tree is illustrated in fig. 13. Let us refer to this tree as the tree T, and let the co-tree be the graph C as shown in fig. 14. The tree T now labels the structure tree, and since C has two connected components, there are two nodes in the structure tree above the root. The node 3 of the original graph is an isolated node in C, and therefore it requires no further decomposition. Hence it also corresponds to a terminal node in the structure tree. The other connected component of the co-tree C can now be decomposed in the same way, and the resultant spanning tree T_1, and the co-tree C_1 are shown in figs. 15 and 16 respectively. C_1 now consists of 5 connected components, of which 3 are single nodes, and the other two are each trees with one line and two nodes. The structure tree S is now of the form shown in fig. 18, where the internal nodes are labelled with the tree T, T_1, T_2 and T_3, where these latter two are shown in fig. 17, and the terminal nodes of S are labelled with the node labels of the original graph G.

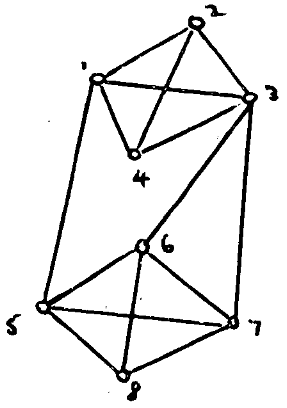


Fig. 12.

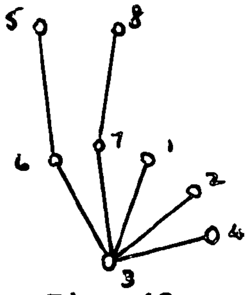


Fig. 13.

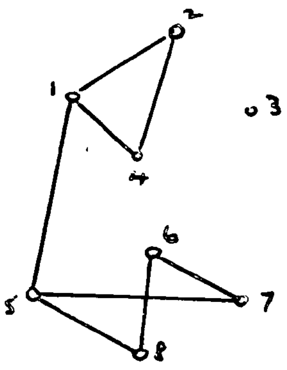


Fig. 14.

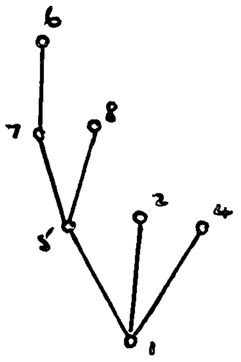


Fig. 15.

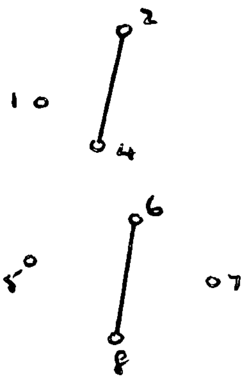


Fig. 16.

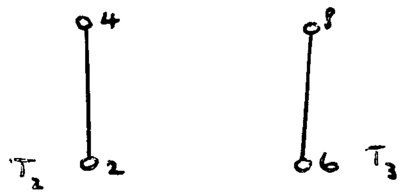


Fig. 17.

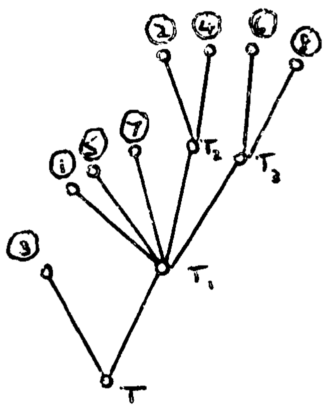


Fig. 18.

Thus we completely represent the process of decomposing a graph by its corresponding structure tree. Having decomposed a graph into its structure tree, we would like to make this process reversible, i.e. from the structure tree with its node labels can we reconstruct the original graph? Clearly this can be done if the node labels of the graph are kept in each spanning tree as well as at the terminals of the structure tree. The question then is: How much of this information can be discarded such that the reconstruction can be done unambiguously and successfully? This is still an open question, but we may make some observations which allow us to reduce the ambiguities which are present with the information which is available. Firstly, since the extraction of the spanning tree implies that all the lines incident to the root of the spanning tree are put into the spanning tree, we know that there must be at least one isolated node at each level of the structure tree, and the root of the spanning tree which labels each non-terminal node of the structure tree corresponds to one of the isolated points of the co-tree associated with that spanning tree.

The second observation which can be made is to do with the points which may legitimately be joined. Consider the graph G shown earlier, in which an attempt is made to reconstruct the graph corresponding to the spanning tree T and its associated co-tree. Now by the way in which it was constructed we know that T_1 is mushrooming with respect to the graph which it spans, and the two lines that form T_2 and T_3 join nodes x and y for which $|h_1(x) - h_1(y)| \leq 1$ where $h_1(x)$ is the height of the node x in the tree T_1. However, even after applying this restriction on the graph we still have a choice of how to compose this graph with the tree T and we conclude that the structure tree alone is not sufficient to specify a graph completely, but some further information must also be carried. As has been pointed out previously, to carry the node labels of the graph is clearly sufficient, but considerably more than is necessary. Thus some intermediate quantity of information must be carried, but it is not immediately clear how much.

Some remarks may be made about the structure tree however

1. The total number of terminal nodes in the structure tree is equal to the number of nodes in the graph.
2. The number of lines in the graph is also represented in the structure tree. By remark 1, we know that the number of nodes in a tree which labels a node x of the structure tree is n(x), where n(x) is the number of terminals of the structure tree which are contained in the subtree whose root is x. Thus the number of lines in the tree which labels x is n(x) - 1. Hence, the graph which is represented by this subtree of the

structure tree is composed of the graphs represented by the principal subtrees standing on x, and the tree which labels x. Thus if y_i $(i = 1,\ldots,k)$ are the nodes above x, and w(x) is the number of lines in the graph represented by the subtree standing on x, we have

$$w(x) = \sum_{i=1}^{k} w(y_i) + n(x) - 1$$

We also have

$$n(x) = \sum_{i=1}^{k} n(y_i)$$

and for each terminal node t in the structure tree,

$$n(t) = 1, \qquad w(t) = 0$$

Thus we may evaluate n(x) and w(x) for each node x in the structure tree, and the number of nodes and lines of the decomposed graph are given by n(r) and w(r) respectively where r is the root of the structure tree of the decomposition.

It is clear from the fact that any spanning tree of a graph with n nodes has n-1 lines, and from the way in which the structure tree is built, that the more lines there are in any subgraph, the taller will be the corresponding portion of the structure tree. For instance, if the graph to be decomposed has only n-1 lines (the minimum number which it can have without becoming disconnected) it is itself its own spanning tree, and so the structure tree is the tree of height 1 with n+1 nodes, the root being labelled with the spanning tree which is the whole graph. On the other hand, if we look at the structure tree for the complete graph of order n, we see that the only mushrooming spanning tree of this graph is the tree of height 1 with n nodes, and the co-tree consists of the root of this spanning tree as an isolated node, together with the complete graph of order n-1. Hence the structure tree would be as shown in fig. 19, where the T_{i1} are the labels of the non-terminal nodes and the label T_{i1} is the tree of height 1 with i nodes. The structure tree in this case has height n-1.

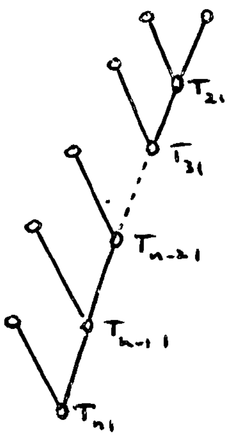


Fig. 19.

By application of the formula for the number of lines in the graph corresponding to this tree we have:

$$\begin{aligned} w(r) &= w(ur(r)) + n - 1 \\ &= w(ur(ur(r))) + (n - 1) - 1 + n - 1 \\ &= w(ur(ur(r))) + (n - 2) + (n - 1) \\ &\dots \\ &= 1 + 2 + \dots + (n - 2) + (n - 1) \\ &= n(n - 1)/2 \end{aligned}$$

which is indeed the number of lines in the complete graph of n nodes. (In the formula we denote the non-terminal node above and to the right of node i by ur(i)).

This observation suggests that we may be able to use the notion of the structure tree to investigate graphs for highly connected regions and in particular to look for complete subgraphs sometimes known as <u>cliques</u>. Such an investigation deserves separate study, and is outside the scope of the present work.

A complete study of the properties of the structure tree has not been attempted, but it is felt that it is at least a possibility for further work in the study of representations of graphs, which field currently lacks a truly recursive and hierarchical representation. A further advantage which may result from such a study is the ability to apply our techniques for describing trees in a labelling independent way to the more general problem of the description of an unlabelled graph.

VII Practical Observations And Conclusions.

In the preceding sections we have discussed methods which are applicable to certain problems connected with the generation or the decomposition of trees and graphs. All of the algorithms described previously have been coded in the Algol W language (see Wirth and Hoare 1966) for the IBM 360/67 computer. In the following sections we describe some of the practical details of the coding and give some indication of the size of problem that may reasonably be tackled using these techniques.

VII.1 Generation Of Trees.

Under this heading we describe the two methods of ordering the set of trees as described in chapter III. In particular we discuss the methods based on the two main representations employed.

VII.1.1 Height Representation.

As was pointed out earlier, the height representation did not allow us to solve the problem of finding the k-th tree in this set without storing the whole table of numbers of trees by height and number of nodes as given by Riordan (1960) (see also Appendix I). However, a program was written which successfully generated the whole sequence of trees of a particular size by generating each tree from its predecessor, the first being supplied explicitly. The first program written to perform this task took a very simplified view, by taking the first tree, i.e. the tree with height sequence

0 1 1 ... 1

and successively attempting to increase each element in turn, beginning with the last element, subject to the constraint that no element may exceed its predecessor by more than one, and subject also to the constraint that the whole sequence must be a valid height sequence. This, like the method described in Snow (1966), was inefficient in the sense that a number of complete sequences were generated and subsequently rejected. Thus the time required to generate all the rooted trees was more nearly related to the number of ordered rooted trees than to the number of unordered rooted trees.

In Scoins (1968), however, a very elegant recursive method of producing the list of all rooted trees of n nodes is presented. The heart of this method is a recursive procedure which determines the upper limit of any element in the height vector by keeping a record of the element with which the current element is to be compared. Thus a backward pointer is kept for each element indicating the subtree and the node which limit the value of that element if the sequence is to satisfy the conditions of the canonical ordering. The Algol stack mechanism takes the responsibility for maintaining the list of backward pointers upon which this method relies. By keeping such a stack it is possible to generate the height vectors of all rooted trees with a given number of nodes without duplications, and without either generating any sequences which have subsequently to be rejected or requiring any complicated checking process to ensure the validity of any sequence produced. Thus using this method, the time taken to generate all the trees is much more closely related to the number of rooted trees, i.e. the time taken per tree is more nearly constant.

VII.1.2 Weight Representation.

A program was written to discover the k-th tree with n nodes for any n, and any k, $1 \geq k \geq T_n$ in the Cayley (weight representation) ordering, but since to some extent this involved counting up through the trees with partitions less than the partition of the k-th, it is not a fair comparison to generate the k-th tree for each $k = 1,\ldots,T_n$ and compare this method with a technique in which each tree is generated from its predecessor. This is because in generating the (k+1)-th tree, a large amount of work already done in finding the k-th tree is repeated. However this method, it is claimed, has a different function from that of generating all trees of a certain size and is therefore to be considered on its own merits.

We recall that the weight representation uses the number of nodes in a subtree (or the 'weight' of a subtree) to describe the node at the root of each subtree, and that for a tree with n nodes, the weights of the nodes adjacent to the root form a partition of n-1. This was discussed in section III.3.2

The program discovers first the partition of the k-th tree, by inspecting each partition in turn and asking how many trees there are with this partition, and subtracting this value from k. When the value of k finally becomes negative we know that the current partition is the partition corresponding to the required tree, and that we require the k'-th tree with this partition, where k' is the last value taken by k before becoming negative. The next step is to set up a vector kd(i) i = 1,...,p where p is the number of parts in the partition for the required tree, and where kd(i) is the index number of the tree of size part(i). This tree may then be found recursively for each i, and the weight representations of these trees may be inserted into the appropriate positions in the weight vector to obtain a complete representation of the k-th tree with n nodes in the Cayley ordering.

The problem of generating all trees in the Cayley ordering, disregarding the problem of generating the k-th tree, is somewhat more akin to the problem of generating the last (T_n-th) such tree by the method described above. This is because it was found necessary to 'count through' all the other trees in the set in order to find the last, and the action of actually generating them on the way past is a relatively small amount of additional work.

The comments made here about the amount of work required by these various methods are summarised in table I, in which programs A, B and C are respectively Scoins' method for generating trees by height sequence, the method which finds the k-th tree for k = 1,...,T_n, and the method which generates the T_n-th tree, but is extended to display the other trees 'incidentally'. The programs were all written in Algol W and the figures are seconds of 360/67 CPU time.

n	Prog.A	Prog.B	Prog.C
5	0.05	0.10	0.04
6	0.10	0.31	0.12
7	0.23	1.13	0.40
8	0.53	4.11	1.03
9	1.39	15.52	2.76
10	3.81	60.76	7.65

Table I.

VII.2 The Canonical Ordering Algorithm.

Some indication of the time requirements for Corneil's algorithm as described in chapter IV are given in the final chapter of Corneil's thesis (Corneil 1968), in which he claims that the isomorphism algorithm which he describes has a running time proportional to n^k where n is the number of nodes in the graphs, and k is related to the highest level of strong regularity of any subgraph in the graphs, and that if the graphs are no more than 2-strongly regular, then k = 5. This is a significant improvement over previous graph isomorphism algorithms, for which the worst case running time would apparently be proportional to n! for a pair of n node graphs. Corneil deals in some detail with the establishment of timing estimates for all his algorithms and examines the performance of his isomorphism algorithm on a series of special classes of graphs, such as polygons and generalisations of polygons. He also shows that his estimated results agree very closely with the observed timings. Another observation made by Corneil is that his observed results suggest that most pairs of graphs can be tested for isomorphism in a time considerably less than the upper bound of n^5, and in general the time required is proportional to about n^3.

The whole of Corneil's algorithm has been re-coded in Algol W and tested on a number of trial graphs. It has not been possible to perform a satisfactory series of tests on the performance of this algorithm, since this should be done by generating a large number of random graphs with n nodes and taking the average time for the execution of this algorithm for various values of n. The disadvantage of this method is a lack of understanding of what is meant by randomness in graphs. A very crude method would be to generate random upper triangular binary matrices with various controllable parameters, such as the number of nodes n, and the density (the proportion of 'ones' in the matrix). This method, however, does not allow any control over the structural properties of the graph. We are unable even to guarantee connectedness (or otherwise) of the graphs produced by this method. The algorithm was however tested on a sample of graphs of various sizes and all that can be said is that our observations agreed to a significant extent with those of Corneil and with his theoretical predictions.

VII.3 Generation Of Graphs.

The algorithms developed to generate graphs, discussed in chapter V, were somewhat disappointing in their performance, in the sense that the generation of any graphs with more than about 8 nodes seemed to be quite impractical by these methods. However, it must be pointed out that this is at least in part due to the explosion of the number of graphs to be generated. A more meaningful measure would perhaps be the rate of generation of the graphs, that is, the time per graph taken by a particular algorithm.

VII.3.1 The Sieve Algorithm.

In spite of the fact that the sieve algorithm was able to use the canonical ordering algorithm to determine the automorphism partitioning and hence the automorphism group, it still turned out to be necessary to obtain every permutation in the set of n! for each distinct graph generated. Thus it was necessary to generate a total of $n!*g_n$ permutations, where g_n is the number of graphs with n nodes. This method could only be justified if the cost of applying a particular permutation to a graph is significantly more than the cost of multiplying permutations. This is clearly not true in the representations which have been chosen here, and we conclude by saying that n! operations (in this case permutation multiplications) per graph is unacceptably high. Beside this figure, the additional requirement of (say) n^5 for discovering the automorphism group of each graph may be regarded as negligible.

VII.3.2 The Equivalence Graph.

Before we discuss the efficiency or otherwise of the equivalence graph method of generating all the graphs of n nodes, it is appropriate to digress a little, and look at the asymptotic behaviour of the sequence g_n for n = 1, 2, ...

From the work of Harary (1955) and also Davis (1953) we have an expression for g_n:

$$g_n = \frac{1}{n!} \sum_{\pi \in P_n} 2^{d(\pi)}$$

where $d(\pi)$ is given by

$$d(\pi) = \sum_{k=1}^{n} p_k \{[k/2] + k(p_k - 1)\} + \sum_{h<k} p_h p_k (h,k)$$

where (h,k) is the highest common factor of h and k, and the cycle class of the permutation π is given by $(p_1, p_2, \ldots, p_n)$. The minimum value of $d(\pi)$ is taken for any cyclic permutation, where the cycle class is (0,0,...,1). In this case,

$$d(\pi) = [n/2]$$

Thus, g_n is a sum of n! terms each of which is not less than $2^{[n/2]}/n!$. Hence

$$g_n \geq 2^{[n/2]}$$

Similarly, the largest value of $d(\pi)$ appears when π is the identity permutation, whose cycle class is (n,0,...,0). In this case

$$d(\pi) = n(n-1)/2$$

Thus, by a similar argument to that given above,

$$g_n \leq 2^{n(n-1)/2}$$

This is in fact only a confirmation of the fact that there are fewer unlabelled graphs than there are labelled graphs, which, as we saw in chapter V, can be represented by the set of bitstrings of length n(n-1)/2.

It is clear, however, that the number of graphs is related in some way to 2^k where

$$n/2 \leq k \leq n(n-1)/2$$

The number of nodes in the equivalence graph is $2^{n(n-1)/2}$ and the number of connected components is g_n which is proportional to some power of 2. It is therefore clear that while the size of the equivalence graph is growing faster with n than is the number of connected components, the difference in rates of growth is not as marked in this case as it was in the case of the sieve algorithm. As for the program to find these graphs, it was felt that the number of permutation applications required to establish the connectivity of the equivalence graph was a fair measure of the work required to obtain the set of graphs. It was observed in the examples tried that the number of such applications required appeared to be approximately proportional to the number of nodes in the equivalence graph, the constant of proportionality being about 1.4.

The largest value of n for which graphs were actually generated was 6. There were two reasons why no more were attempted.

(a) this took 90 secs. of CPU time, and it was estimated that n = 7 would require 90 minutes.

(b) the space requirement for representing the equivalence graph was causing a problem, since the graph for n = 7 would have 2^{21} nodes.

VII.3.3 The Partition Method.

In this section there are two distinct aspects to consider. Firstly, it is necessary to generate all the graphical partitions of length n, and secondly, the set of graphs with this partition has to be completely generated. The algorithms under discussion here are described in section V.5.

As regards the set of graphical partitions, it is expected that the time taken to generate a single partition from its predecessor would be proportional to n. In fact, the algorithm, in broad outline, scans along a partition looking for an element to reduce, and having found one, it moves back to the beginning of the partition recomputing all the earlier elements. Thus we may expect to have to recompute on average one half of the elements at each stage. In any case the recomputation required is proportional to n. Actually, as table II shows, the time required per sequence seems to be proportional to some power of n which is slightly greater than one. This can perhaps be explained by observing that the algorithm actually generates some partitions of <u>odd</u> numbers which satisfy the criteria, and these of course have to be discarded as non-graphical. In terms of feasibility, however, we see that we can generate all the graphical partitions of length 9 in only 30 secs, and thus it is quite practicable to generate these and possibly also those of length 10 and 11.

n	no. of sequences	time (secs)	time/seq (ms/seq)
4	11	0.06	5.45
5	31	0.16	5.16
6	102	0.53	5.20
7	342	1.86	5.44
8	1223	7.42	6.07
9	4450	34.78	7.82

<u>Table II.</u>

The creation of an initial graph with a given partition is trivial, particularly if the adjacency matrix is used to represent the graph. The generation of all graphs with a given partition is however a completely unknown quantity as far as any analytic treatment is concerned. All that is possible at present is to quote some experimental observations.

We saw in the example at the end of chapter V that there were a total of 18 different 4-tuples to be inspected, whereas the number of graphs which were actually produced was 5, and one of these was found directly.

<u>VII.4 Factorisation.</u>

We present here some general remarks related to the performance of the various factorisation techniques described in the preceding chapter. As in section VII.2, these techniques are highly dependent upon the actual structure of the graph, and it is therefore necessary to perform a series of tests, applying the algorithms to a large number of randomly generated graphs. Again we must remark that the subject of randomness in graphs is not well understood, and thus such tests are of limited value only.

VII.4.1 Matchings.

Edmonds suggests that his method operated in time proportional to n^4 for finding a maximal matching on a graph of n nodes. His argument was based on some very generalised statements about the maximum number of times a particular operation can be carried out, and how many operations are involved in these operations. Now while Edmonds' estimate is only presented as an upper bound on the method, we claim that it is far from being a precise upper bound. For instance, Edmonds remarks that the matching can be found by growing an alternating tree from at most n points, and the alternating tree algorithm requires time proportional to n^3 to find an augmenting path and amend the matching. Apart from the trivial observation that only n/2 trees are required, since each improvement in the matching includes two further nodes, we also observe that in at least one case, and except for very special graphs in other cases as well, the operation of finding an augmenting path is simply that of finding a node adjacent to the root of the alternating tree, which is an order n operation at most. This case arises when the matching is empty, i.e. when the first alternating tree is being grown.

The method of backtracking to find a perfect matching, in common with most backtrack type algorithms can be unpredictable with regard to its requirements. In the best possible situation, as would be the case in (say) the complete graph, a perfect matching can be found in as few as n/2 attempts (where an attempt is to be thought of as removing a line and its end-points from the graph), whereas in general it would be more difficult than this. In the worst case, and this includes all those cases where the algorithm fails to find a perfect matching, the work done will be nearer n! than n. This method, as was noted earlier, also has the disadvantage of failing completely when no perfect matching exists, while on the other hand, Edmonds' algorithm will always find a maximal matching, whether this is perfect or not.

VII.4.2 Articulation Points.

The most efficient spanning tree finding algortihm will in theory operate in time proportional to m, the number of lines in the graph, but in practice this is probably more like n^2 for most implementable algorithms. This we will use as the starting point for our discussions.

The articulation point algorithm as described in the preceding chapter will require an amount of work proportional to $n\tau$, where τ is the work required to find a spanning tree in a graph. This is an upper bound, since we need to grow at most one spanning tree for each node. We notice in passing that the operation of growing a planted spanning tree is no different from growing any other type of spanning tree. The slight saving to be made by marking all the terminal nodes of any spanning tree, so that no spanning tree is required from these nodes, will only make a difference of an approximately constant factor in the time of operation of the algorithm. This factor appears to be around 2/3, but is of course dependent on the actual graph being considered.

VII.4.3 Spanning Tree Algorithms.

The time τ to grow a spanning tree on a graph is of course the time required to grow a labelling dependent tree, and the algorithms described in section VI.4 require more operations than those required for any spanning tree. In the first place, a mushrooming tree requires more work than a general tree. With regard to the improvement algorithm, although the number of choices of lines which are candidates for insertion is much smaller than the total number of lines in the co-tree, and once a choice has been made, the line to be removed from the tree is completely determined, there seems to be no way of predicting the number of iterations which may be carried out by the algorithm.

The matching method for constructing the first mushrooming spanning tree is of course trading time to set up this tree for (hopefully) a saving in the number of iterations of the improvement algorithm. No detailed comparisons have been carried out, but preliminary observations suggest that a small amount can be saved by combining these two algorithms. Some more concrete results on the extraction of spanning trees from a graph are reported in Chase (1970).

VII.4.4 Complete Decomposition.

The complete decomposition of a graph, and the construction of a structure tree is simply a repeated application of the spanning tree algorithms described earlier. One decomposition is required for each internal node of the structure tree, and the number of these is related in some way to m, the number of lines in the original graph. A first estimate of the work required to perform the complete decomposition of a graph would then be $m\tau$.

VII.4.5 Summary.

The somewhat imprecise remarks made above about the performance of the algorithms described under the general title of Factorisation are included to illustrate two main facts.

(i) Considerably more effort is required to understand more fully what is meant by 'randomness' in graphs, and the 'expected' performance of an algorithm operating on a 'typical' graph.

(ii) All of these algorithms work in time proportional to some power of n, where this power is fairly small. This means that, in general, the amount of time required by any of these algorithms will not be embarrassingly great, and that any graph which can be fitted the main store of a modern computer can be processed in a reasonable time. This is in contrast to the methods (such as those described in chapter V) where the number of entities (e.g. graphs) is exploding as n increases, and although the objects may be quite small in themselves, the number of them, and hence the time required to generate them all, is increasing in an unmanageable way.

VII.5 Conclusions.

In this final section we attempt to make some remarks about what has been achieved in the work, and to draw together all the final remarks which can be made about the work as a whole. In addition we wish to point to further problems and investigations along the same lines which we feel it would be profitable and interesting to pursue.

In the tree section there is not a great deal to say. Clearly we have not exhausted the possibilities for representing trees, but we have been successful in that the two representations on which we have been concentrating our attentions have both yielded reasonably satisfactory methods of generating trees, the second of which - the weight representation - relates this generation very closely to one method of counting trees.

It is unfortunate, but seemingly unavoidable, that the two representations are to some extent in conflict. The natural ordering of the trees, as displayed by the height representation, is more closely related to the structure of the tree than is the weight representation, but the weight representation is bound to be more related to the counting sequences because these latter are always expressed in terms of numbers of nodes. A radically different approach to the counting problem could perhaps reconcile these two representations.

With regard to the graph isomorphism problem, we raised two questions related in detail to the work of that chapter, namely, the reconstruction of a graph given only its cycle vectors or its vertex quotient graphs. A more general problem might be attacked using some modifications of specific techniques used in this work, and that is the subgraph isomorphism problem. Given two graphs A and B, is the graph A isomorphic to any subgraph of B? This problem seems to be inherently more difficult than the straight isomorphism problem, since the implied (or explicit) set equality relations between nodes are now set inclusion relations, and whereas in the isomorphism problem corresponding nodes were required (for example) to have equal degrees, the relation between degrees of corresponding nodes are now inequalities. Whether the methods of partitionings and refinements of partitionings, and the quotient graph notion will generalise to the subgraph isomorphism case is still an open (and this author believes, interesting) question.

The graph indexing problem was not solved completely to our satisfaction, in the sense that it is felt that a more efficient method should be possible. However we have made some progress in this field by the comparison of the two methods presented, and also the algorithms in connection with the generation of graphs by partition. It is felt that further study of the representation of graphs would help in the search for a more efficient method of generating all graphs, and we might further hope that the problem of generating any objects which may be counted using Polya's theorem. This implies, incidentally, that graphs with a given partition may also be generated.

The section on factorisation shows a number of methods for extracting a spanning tree from a graph which in some sense may be regarded as 'optimal' or 'near optimal'. We feel however that there are many other algorithms which would give similar results, but which may have the added advantage of making a proof of optimality possible. We also feel that the properties of the structure tree have not been fully explored, especially with regard to the problem of finding cliques in a graph. It also seems possible that the discovery of 'near-cliques' i.e. subgraphs which are nearly but not entirely complete, could be effected using the idea of the structure tree. This may lead to a new way of defining and discovering clusters in a graph, which could have applications in a number of fields, two such being information retrieval and linguistic analysis.

References.

Augustson J.G. and Minker J. (1970) An Analysis of some Graph Theoretic Cluster Techniques. J.A.C.M. 17 pp571 - 588

Berge C. (1958) The Theory of Graphs. Dunod Paris (Tr. by A. Doig Methuen London 1962)

Busacker R.G. and Saaty T.L. (1965) Finite Graphs and Networks: An Introduction with Applications. McGraw-Hill New York

Cayley A. (1889) Collected Mathematics Papers, Cambridge 3 p242, 9 p427, 11 p365

Chase S. (1970) Analysis of Algorithms for Finding All Spanning Trees of a Graph. IBM Research Report RC3190 Yorktown Heights, New York.

Corneil D.G. (1968) Graph Isomorphism. Ph.D. Thesis, University of Toronto.

Corneil D.G. (1972) Algorithm to Find the Automorphism Partitioning of a Graph. B.I.T. 12 pp161 - 171

Corneil D.G. and Gotlieb C.C. (1970) An Efficient Algorithm for Graph Isomorphism. J.A.C.M. 17 pp 51 - 64

Davis R.L. (1953) The Number of Structures of Finite Relations.

Proc. Amer. Math. Soc. 4 pp486 - 495

de Bruin N.G. (1964) Polya's Theory of Counting. in Applied Combinatorial Mathematics (E.F.Beckenbach ed) Wiley New York

Dijkstra E.W. (1959) A Note on Two Problems in Connection with Graphs. Num. Math 1 pp269 - 271

Edmonds J. (1965) Paths, Trees and Flowers. Canad. J. Math. 17 pp449 - 467

Erdos P. and Gallai T. (1960) Graphs with Prescribed Degrees of Vertices. Math. Lapok 11 pp264 - 274

Fortet R. (1959) L'algebre de Boole et ses Applications en Recherche Operationelle. Mimeograph, Societe de Mathematiques Appliquees

Gotlieb C.C. and Corneil D.G. (1967) Algorithms for Finding a Fundamental Set of Cycles for an Undirected Linear Graph. C.A.C.M. 10 pp780 - 783

Hakimi S. (1962) On the Realizability of a Set of Integers as Degrees of the Vertices of a Graph. J. SIAM Appl. Math. 10 pp496 - 506

Hall P. (1935) On Representatives of Subsets. J. Lond. Math.

Soc. 10 pp26 - 30

Harary F. (1955) The Number of Linear, Directed, Rooted and Connected Graphs. Trans. Amer. Math. Soc. 78 pp445 - 463

Harary F. (1960) Unsolved Problems in the Enumeration of Graphs. Publ. Math. Inst. Hungar. Acad. Sci. 5 pp63 - 95

Harary F. (1964) Combinatorial Problems in Graphical Enumeration. in Applied Combinatorial Mathematics (E.F.Beckenbach ed) Wiley New York

Harary F. (1967) Unsolved Problems in Graphical Enumeration III. in Graph Theory and Theoretical Physics (F.Harary ed) Academic Press New York

Harary F. (1968) The Theory of Graphs. Addison-Wesley, New York

Harary F., Norman R.Z. and Cartwright D. (1965) Structural Models: An Introduction to the Theory of Directed Graphs. Wiley, New York

Harary F. and Prins G. (1959) The Number of Homeomorphically Irreducible Trees and Other Species. Acta Math. 101 pp141 - 162

Knuth D.E. (1968) The Art of Computer Programming: Volume I. Addison-Wesley, New York

Ledermann W. (1961) An Introduction to the Theory of Finite Groups. Oliver and Boyd, Edinburgh

Lehmer D.H. (1964) The Machine Tools of Combinatorics. in Applied Combinatorial Mathematics (E.F.Beckenbach ed) Wiley New York

Liu C.L. (1968) An Introduction to Combinatorial Mathematics. McGraw-Hill New York

Meetham A.R. (1968) Partial Isomorhisms in Graphs and Strucural Similarities in Tree-like Organic Molecules. Proc. IFIP Congress, Edinburgh ppA108 - A110

Mulligan G.D. (1972) Algorithms for Finding Cliques of a Graph. Technical Report TR41, University of Toronto Computing Centre

Mulligan G.D. and Corneil D.G. (1972) Corrections to Bierstone's Algorithm for Generating Cliques. J.A.C.M. 19 pp244 - 247

Nicholson T. (1966) Finding the Shortest Route between Two Points in a Network. Comp. J. 9 pp275 - 280

Obruca A.K. (1966) The Representation and Manipulation of Trees and Linear Graphs within a Computer and Some Applications. Ph.D. Thesis, University of Newcastle upon Tyne

Otter R. (1948) The Number of Trees. Ann. of Math. 49 pp583 - 599

Page E.S. (1971) Systematic Generation of Ordered Sequences using Recurrence Relations. Comp. J. 14 pp150 - 153

Parthasarathy K.R. (1968) Enumeration of Graphs with Given Partition. Canad. J. Math. 20 pp40 - 47

Paton K. (1969) An Algorithm for Finding a Fundamental Set of Cycles of a Graph. C.A.C.M. 12 pp514 - 518

Pohl I. (1969) Bi-directional and Heuristic Search in Path Problems. SLAC-104, Stanford Linear Accelerator Center, Stanford, California

Polya G. (1937) Kombinatorische Anzahlbestimmungen fur Gruppen, Graphen und Chemische Verbindungen. Acta Math. 68 pp145 - 254

Read R.C. (1969) Teaching Graph Theory to a Computer. in Recent Progress in Combinatorics (W.Tutte ed) Academic Press, New York

Read R.C. and Parris R. (1966) Graph Isomorphism and the Coding of Graphs. UWI/CC3 Research Report, University of West Indies

Riordan J. (1960) The Number of Trees by Height and Diameter. IBM J. Res. Dev 4 pp473 - 478

Scoins H.I. (1967) Linear Graphs and Trees. In Machine Intelligence 1 (N.L.Collins and D.Michie eds) pp1 - 15

Scoins H.I. (1968) Placing Trees in Lexicographical Order. In Machine Intelligence 3 (D.Michie ed) pp43 - 60

Snow C.R. (1966) An Investigation into the Equivalence of Free Trees. M.Sc. Dissertation, University of Newcastle upon Tyne

Snow C.R. and Scoins H.I. (1969) Towards the Unique Decomposition of Graphs. in Machine Intelligence 4 (B.Meltzer and D.Michie eds) pp45 - 55

Sussenguth E.H. (1965) A Graph-theoretic Algorithm for Matching Chemical Structures. J. Chem. Doc. 5 pp36 - 43

Turner J. (1969) Keyword Indexed Bibliography of Graph Theory. in Proof Techniques in Graph Theory (F.Harary ed) Academic Press, New York

Ulam S.H. (1960) A Collection of Mathematical Problems Wiley (Interscience) New York

Unger S.H. (1964) GIT - a Heuristic Program for Testing Pairs of Directed Line Graphs for Isomorhism. C.A.C.M. 7 pp26 - 34

Warshall S. (1962) A Theorem on Boolean Matrices. J.A.C.M. 9 pp11 - 12

Wirth N. and Hoare C.A.R. (1966) A Contribution to the Development of Algol. C.A.C.M. 9 pp413 - 432

Witzgall C. and Zahn C.T. (1965) Modification of Edmonds' Maximum Matching Algorithm. J. Res. Nat. Bur. Stds. 69B p9

Appendix I. Counting Series For Trees And Graphs.

In this appendix, we give the methods by which the various numbers which have been used in this work may be computed, and the numbers themselves are also presented.

AI.1 Ordered Rooted Trees.

The method by which the number of ordered rooted trees of n nodes may be computed is suggested in chapter III. By considering an ordered rooted tree as being composed of two parts, one with i nodes, and the other with n-i nodes, we have the recurrence relation

(1) .... $\mathcal{T}_n = \sum_{i=1}^{n-1} \mathcal{T}_i \mathcal{T}_{n-i}$ $n \geq 2$

where $\mathcal{T}_n$ is the number of ordered rooted trees with n nodes. If we also define

$$\mathcal{T}_1 = 1$$

we may deduce the generating function equation

$$\mathcal{T}(x) = x + \{ \mathcal{T}(x) \}^2$$

where

$$\mathcal{T}(x) = \sum_{n=1}^{\infty} \mathcal{T}_n x^n$$

It is equation (1), however, which is used to compute the values of $\mathcal{T}_n$, and these values are given in Table I for n = 1,...,24.

n	$\mathcal{T}_n$
1	1
2	1
3	2
4	5
5	14
6	42
7	132
8	429
9	1430
10	4862
11	16796
12	58786
13	208012
14	742900
15	2674440
16	9694845
17	35357670
18	129644790
19	477638700
20	1767263190
21	6564120420
22	24466267020
23	91482563640
24	343059613650

Table I.

AI.2 Rooted Trees.

It was demonstrated in chapter III that of the two methods for counting unordered rooted trees discussed, each was derivable from the other by a suitable piece of algebra on the generating function equations.

The Cayley method of counting trees was the simpler to implement, but it involved generating all the partitions of n-1 (for the number T_n of trees with n nodes). If it is only required to find the values of T_n for n = 1, 2, etc., a method using Harary and Prins equation is more convenient. Harary and Prins showed that if

$$T(x) = \sum_{n=1}^{\infty} T_n x^n$$

where T_n is the number of trees with n nodes, then

$$T(x) = x \exp \{ \sum_{r=1}^{\infty} T(x^r) / r \}$$

Now let

$$A(x) = T(x) / x$$

and

$$B(x) = \sum_{r=1}^{\infty} T(x^r) / r$$

where

$$A(x) = a_0 + a_1 x + a_2 x^2 + \ldots$$

and

$$B(x) = b_0 + b_1 x + b_2 x^2 + \ldots$$

then

$$A(x) = \exp \{ B(x) \}$$

Differentiating with respect to x, we have

$$A'(x) = \exp \{ B(x) \} B'(x)$$
$$= A(x) B'(x)$$

This immediately leads to

$$a_1 + 2a_2x + 3a_3x^2 + \ldots = (a_0 + a_1x + a_2x^2 + \ldots)(b_1 + 2b_2x + 3b_3x^2 + \ldots)$$

or

$$na_n = a_0nb_n + a_1(n-1)b_{n-1} + \ldots + a_{n-1}b_1$$

(2) ....$na_n = \sum_{i=1}^{n} ib_i a_{n-i}$

Now we also have

$$B(x) = \sum_{r=1}^{\infty} T(x^r)/r = \sum_{r=1}^{\infty} \frac{x^r}{r} A(x^r)$$

i.e.

$$\sum_{n=0}^{\infty} b_n x^n = \sum_{r=1}^{\infty} \frac{x^r}{r} \sum_{k=0}^{\infty} a_k x^{kr}$$

and thus

$$b_n = \sum_{\substack{r(k+1)=n \\ r\geq 1,\ k\geq 0}} a_k/r$$

Thus replacing k+1 by p, and multiplying by n, we have

(3) ....$nb_n = \sum_{\substack{rp=n \\ r,p\geq 1}} na_{p-1} = \sum_{\substack{\text{all divisors} \\ p \text{ of } n}} pa_{p-1}$

Equations (2) and (3), together with the relation

$$T_n = a_{n-1}$$

are sufficient to calculate all the a's, b's and T's. These numbers are given in Table II.

n	T_n
1	1
2	1
3	2
4	4
5	9
6	20
7	48
8	115
9	286
10	719
11	1842
12	4766
13	12486
14	32973
15	87811
16	235381
17	634847
18	1721159
19	4688676
20	12826228
21	35221832
22	97055181
23	268282855
24	743724984
25	2067174645
26	5759636510
27	16083734329

Table II.

AI.3 Rooted Trees Of A Given Height.

A similar technique may be used to compute the number of trees with n nodes and height h. Riordan (1960) shows that

$$T^{(h+1)}(x) = x \exp \{ \sum_{r=1}^{\infty} T^{(h)}(x^r)/r \}$$

where $T^{(h)}(x)$ is the generating function for the numbers of trees with maximum height h. The generating function for trees with height exactly h is therefore

$$T^{(h)}(x) - T^{(h-1)}(x)$$

If we now define

$$A^{(h)}(x) = T^{(h)}(x) / x$$

and

$$B^{(h)}(x) = \sum_{r=1}^{\infty} T^{(h)}(x^r) / r$$

by analogous manipulation to that used previously, we have

$$na_n^{(h+1)} = \sum_{i=1}^{n} ib_i^{(h)} a_{n-i}^{(h+1)}$$

and

$$nb_n^{(h)} = \sum_{\substack{\text{all divisors} \\ p \text{ of } n}} pa_{p-1}^{(h)}$$

Now by choosing suitable boundary conditions, namely

$$a_n^{(1)} = 1 \qquad \text{for all } n \geq 2$$

and

$$a_0^{(h)} = 1 \qquad \text{for all } h \geq 1$$

we may compute the values of $a_n^{(h)}$. These numbers for n = 1,...,20 and h = 0,...,19 are given in Table III.

h \ n	1	2	3	4	5	6	7
0	1	0	0	0	0	0	0
1	0	1	1	1	1	1	1
2	0	0	1	2	4	6	10
3	0	0	0	1	3	8	18
4	0	0	0	0	1	4	13
5	0	0	0	0	0	1	5
6	0	0	0	0	0	0	1

h \ n	8	9	10	11	12	13	14
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	14	21	29	41	55	76	100
3	38	76	147	277	509	924	1648
4	36	93	225	528	1198	2666	5815
5	19	61	180	498	1323	3405	8557
6	6	26	94	308	941	2744	7722
7	1	7	34	136	487	1615	5079
8	0	1	8	43	188	728	2593
9	0	0	1	9	53	251	1043
10	0	0	0	1	10	64	326
11	0	0	0	0	1	11	76
12	0	0	0	0	0	1	12
13	0	0	0	0	0	0	1

Table III.

n / h	15	16	17	18	19	20
0	0	0	0	0	0	0
1	1	1	1	1	1	1
2	134	175	230	296	384	489
3	2912	5088	8823	15170	25935	44042
4	12517	26587	55933	116564	241151	495417
5	21103	51248	122898	291579	685562	1599209
6	21166	56809	149971	390517	1005491	2564164
7	15349	45009	128899	362266	1002681	2740448
8	8706	27961	86802	262348	776126	2256418
9	3961	14102	47816	156129	494769	1530583
10	1445	5819	21858	77878	266265	880883
11	414	1948	8282	32695	121963	435168
12	89	516	2567	11481	47481	184903
13	13	103	633	3318	15564	67249
14	1	14	118	766	4218	20697
15	0	1	15	134	916	5285
16	0	0	1	16	151	1084
17	0	0	0	1	17	169
18	0	0	0	0	1	18
19	0	0	0	0	0	1

Table III. (cont)

AI.4 Free Trees.

Table IV gives the number of free trees with n nodes for n = 1,...,27, and these are computed directly from the values of T_n (given in Table II) using Otter's formula:

$$t(x) = T(x) - \tfrac{1}{2}\{ T(x)^2 - T(x^2) \}$$

where $t(x) = t_1 x + t_2 x^2 + \ldots$ is the generating function for free trees, from which we obtain:

$$t_n = T_n - \tfrac{1}{2}\sum_{\substack{p\geq 1 \\ q\geq 1 \\ p+q=n}} T_p T_q + \tfrac{1}{2} T_{n/2}$$

where $T_{n/2}$ is considered to be zero when n is odd.

n	t_n
1	1
2	1
3	1
4	2
5	3
6	6
7	11
8	23
9	47
10	106
11	235
12	551
13	1301
14	3159
15	7741
16	19320
17	48629
18	123867
19	317955
20	823065
21	2144505
22	5623756
23	14828074
24	39299897
25	104636890
26	279793450
27	751065460

Table IV.

AI.5 Graphs.

In Harary (1955) we learn how to count the graphs with n nodes and m lines. This method, however, involves the generation of all the cycle classes of permutations in the symmetric group of order n, which implies the generation of all the partitions of n. A program was written to obtain these values, but the method was extremely time consuming, and so no values were obtained for $n > 7$. The values which were computed are given in Table V. A slightly quicker method for computing the total number of graphs of n nodes is based on the method of Davis (1953).

n / m	1	2	3	4	5	6
0	1	1	1	1	1	1
1		1	1	1	1	1
2			1	2	2	2
3			1	3	4	5
4				2	6	9
5				1	6	15
6				1	6	21
7					4	24
8					2	24
9					1	21
10					1	15
11						9
12						5
13						2
14						1
15						1

Table V.

Appendix II h-strongly Regular Graphs.

In chapter IV it was pointed out that Corneil's algorithm only works in time proportional to n^5 provided the graph under consideration does not contain a particular type of subgraph. The condition on this subgraph is that it should not be 2-strongly regular.

A 2-strongly regular graph (2-SR) is a graph $G = (V, E)$ in which

(a) for any two nodes $x, y \in V$, s.t. $(x, y) \in E$,

$$|\{v \mid v \in V,\ (x, v) \in E,\ (y, v) \in E\}| = p^1_{11}$$

$$|\{v \mid v \in V,\ (x, v) \in E,\ (y, v) \notin E\}| = p^1_{10}$$

$$|\{v \mid v \in V,\ (x, v) \notin E,\ (y, v) \notin E\}| = p^1_{00}$$

(a) for any two nodes $x, y \in V$, s.t. $(x, y) \notin E$,

$$|\{v \mid v \in V,\ (x, v) \in E,\ (y, v) \in E\}| = p^0_{11}$$

$$|\{v \mid v \in V,\ (x, v) \in E,\ (y, v) \notin E\}| = p^0_{10}$$

$$|\{v \mid v \in V,\ (x, v) \notin E,\ (y, v) \notin E\}| = p^0_{00}$$

where p^k_{ij} are constants for a particular graph.

It can be seen that this concept is an extension to two nodes, taking into account the two cases where the nodes are joined and not joined, of the concept of regularity, which could be restated as

for any node x,

$$|\{v \mid v \in V,\ (x, v) \in E\}| = k$$

$$|\{v \mid v \in V,\ (x, v) \notin E\}| = 1 \quad (= n - k)$$

From this definition of 2-strong regularity we can see

(i) G is regular with degree k

(ii) $k = 1 + p^1_{11} + p^1_{10}$

(iii) $k = p^{0}_{11} + p^{0}_{10}$

(iv) $n = 2 + p^{1}_{11} + 2p^{1}_{10} + p^{1}_{00}$

(v) $n = 2 + p^{0}_{11} + 2p^{0}_{10} + p^{0}_{00}$

For the graph in fig. 1., we have

$n = 10$, $k = 3$

$p^{1}_{11} = 0$, $p^{1}_{10} = 2$

$p^{1}_{00} = 4$, $p^{0}_{11} = 1$

$p^{0}_{10} = 2$, $p^{0}_{00} = 3$

This graph, however, is also transitive (see chapter IV for the definition of transitive).

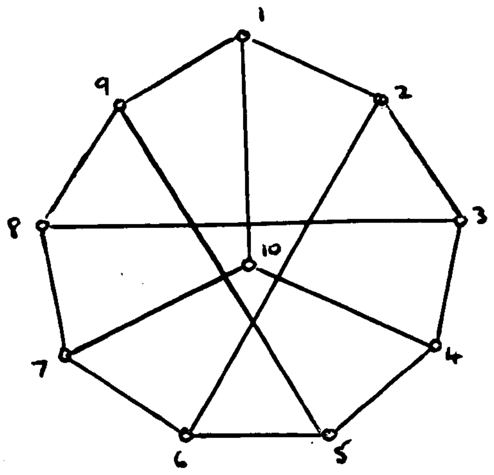


Fig. 1.

Corneil shows that in a 2-SR graph, each node has the same vertex quotient graph, and gives an example of a 2-SR graph which is not transitive. This graph has 26 nodes, and is the smallest known non-transitive 2-strongly regular graph.

In order to distinguish between nodes of a 2-SR graph, the 2-vertex quotient graph is defined. A partitioning of the nodes of the graph into the sets $\{x\}$, $\{y\}$, $V-\{x,y\}$ for some pair of nodes x and y, gives a starting partitioning of the nodes for the refinement algorithm. The final partitioning given by the algorithm defines a quotient graph which is known as the 2-vertex quotient graph for the nodes x and y, and this graph is called Q_{xy}. Thus for any node x, we have a set or family of 2-vertex quotient graphs Q_{xy} for all $y \in V-\{x\}$.

The idea of 2-strong regularity may be extended to 3-strong regularity.

A graph $G = (V,E)$ is 3-strongly regular if:

(a) for all $x,y,z \in V$, s.t. $(x,y) \in E$, $(y,z) \in E$, $(z,x) \in E$,

$$|\{v | v \in V, (v,x) \in E, (v,y) \in E, (v,z) \in E\}| = p^{111}_{111}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \in E, (v,z) \notin E\}| = p^{111}_{110}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \notin E, (v,z) \notin E\}| = p^{111}_{100}$$

$$|\{v | v \in V, (v,x) \notin E, (v,y) \notin E, (v,z) \notin E\}| = p^{111}_{000}$$

(a) for all $x,y,z \in V$, s.t. $(x,y) \in E$, $(y,z) \in E$, $(z,x) \notin E$,

$$|\{v | v \in V, (v,x) \in E, (v,y) \in E, (v,z) \in E\}| = p^{110}_{111}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \in E, (v,z) \notin E\}| = p^{110}_{110}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \notin E, (v,z) \in E\}| = p^{110}_{101}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \notin E, (v,z) \notin E\}| = p^{110}_{100}$$

$$|\{v | v \in V, (v,x) \notin E, (v,y) \in E, (v,z) \notin E\}| = p^{110}_{010}$$

$$|\{v | v \in V, (v,x) \notin E, (v,y) \notin E, (v,z) \notin E\}| = p^{110}_{000}$$

(a) for all $x,y,z \in V$, s.t. $(x,y) \in E$, $(y,z) \notin E$, $(z,x) \notin E$,

$$|\{v | v \in V, (v,x) \in E, (v,y) \in E, (v,z) \in E\}| = p^{100}_{111}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \in E, (v,z) \notin E\}| = p^{100}_{110}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \notin E, (v,z) \in E\}| = p^{100}_{101}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \notin E, (v,z) \notin E\}| = p^{100}_{100}$$

$$|\{v | v \in V, (v,x) \notin E, (v,y) \notin E, (v,z) \in E\}| = p^{100}_{001}$$

$$|\{v | v \in V, (v,x) \notin E, (v,y) \notin E, (v,z) \notin E\}| = p^{100}_{000}$$

(a) for all $x,y,z \in V$, s.t. $(x,y) \notin E$, $(y,z) \notin E$, $(z,x) \notin E$,

$$|\{v | v \in V, (v,x) \in E, (v,y) \in E, (v,z) \in E\}| = p^{000}_{111}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \in E, (v,z) \notin E\}| = p^{000}_{110}$$

$$|\{v | v \in V, (v,x) \in E, (v,y) \notin E, (v,z) \notin E\}| = p^{000}_{100}$$

$$|\{v | v \in V, (v,x) \notin E, (v,y) \notin E, (v,z) \notin E\}| = p^{000}_{000}$$

The property of h-strong regularity can be defined, but in order to do so, it is necessary to specify all non-isomorphic graphs of order h. An h-vertex quotient graph is found in the obvious way by distinguishing h vertices at the start of the partitioning algorithm. A family of h-vertex quotient graphs for a given node x is the set of all h-vertex quotient graphs for which x is one of the distinguished nodes.

The determination of h-strongly regular graphs or subgraphs makes the full graph isomorphism algorithm less efficient, as it increases the power of n to which the time taken by the algorithm is proportional.