Imperial College of Science and Technology

(University of London)

Department of Management Science

Interactive Computer Methods for Plant Layout Scheduling and Group Technology

by

Vizes Nakornchai B.Sc. (Eng), A.C.G.I., M.Sc.

A thesis submitted for the degree of
Doctor of Philosophy of the University of London
and for the
Diploma of Imperial College

August 1982

I would like to dedicate this thesis to my father, my mother , my aunt Apa and my former teacher Kru Aketritra Kokongka. In their own ways, they have made this study possible.

Colorless green ideas sleep furiously.
N.Chomsky.

Acknowledgements

I would like to thank Mr John King, my supervisor, for his constant encouragement and guidance during the course of my work. My wife, Choochit, has been supportive in many ways during these difficult years. Many of my collegues have made my tasks easier: Mark Thornton has made my venture into Pascal an enjoyable experience. Nick Bate has helped in the earlier stage of my attempt to solve the maximal planar graph problem. Patrick Collins has been helpful in more ways than one. Finally, I would like to thank the Thai Govarment who have supported me all through this period.

Abstract

Many combinatorial problems encountered in industry are NP-complete, and it is generally accepted that most of these problems cannot be solved optimally for any practical size. The aims of this thesis are two-fold; firstly to investigate various heuristic techniques that may be applied to certain of these problems; and secondly to investigate the possibility of combining human judgement with the heuristics in order to take into account unquantifiable factors or to overcome certain practical difficulties.

Three classes of problems are selected for the study: plant layout, scheduling and group technology. Two sub-problems of the plant layout problem, namely the quadratic assignment problem (QAP) and the maximal planar graph problem (MPG), are studied. For the QAP, the main emphasis is on an interactive partitioning method. As no computer implementation of a heuristic for the MPG has previously been published, the main effort is concentrated on the development of algorithms and data structures which would lead to efficient implementation of the heuristics. Various construction and improvement heuristics are implemented obviating the need for a planarity testing procedure. The sub-class of the scheduling problem selected for study is the one which can be formulated as an asymmetric travelling salesman problem (ATSP). Such a problem arises whenever the setting up time is sequence dependent. Various tour construction and improvement procedures are considered. In the case of group technology, a comprehensive survey of the literature on group formation is given as no such survey has previously been published. A new improved version of the ROC algorithm is devised. The new algorithm (ROC2) has a linear order of complexity and hence can be used to solve very large practical problems. A new relaxation procedure for bottleneck machines, together with the interactions allowed by the program, are used in conjunction with the ROC2 algorithm to provide solutions of published problems comparable to or better than those produced by existing algorithms, and with less effort.

Contents

List of Tables

List of Figures

1 Introduction

1.1 THE AIM OF THE THESIS

The works on computational complexity by Cook (1971) and Karp (1972) and subsequent authors have given us some understanding and insight into the difficulties encountered in attempts to find solutions to certain problems. There is also a growing acceptance that one class of problems, the NP-complete problem, may never be solved efficiently. Many real-life industrial problems belong to this class. Common problems such as scheduling and plant layout, even in their simpler forms, are very likely to be NP-complete and hence cannot be solved within an acceptable time scale. This applies even to moderately sized problems.

The primary purpose of this thesis is to investigate methods of achieving approximate solutions to some of these problems. The secondary objective is to investigate the possibility of combining human judgement with heuristics to take into account some of the factors that might have been left out during the formulation stage, or in order to take into account certain difficulties that may arise in practice.

1.2 COMPUTATIONAL COMPLEXITIES OF ALGORITHMS

According to computational complexity theory, there are at least two major classes of problems, P and NP. A problem in the P (polynomial) class is defined as a problem that can be solved in polynomially bounded time by a deterministic Turing machine. A deterministic Turing machine is a conceptual model which provides lower bounds on space and time required to solve a problem with a von Neumann computer; most of the computers in use today are of this type. A von Neumann computer, as far as the complexity issue is concerned, is one which executes the instructions sequentially. Hence, a P problem is in essence a problem which has a known polynomial algorithm for the present type of computer. An NP (nondeterministic polynomial) problem is one which can be solved on a nondeterministic Turing machine in polynomially bounded time. A nondeterministic Turing machine is in essence a machine which can carry out unlimited parallel computation. Therefore an NP problem, in practical terms, is a problem that can only be solved by an exponentially bounded algorithm on today's computers.

Another important concept in the complexity theory is the concept of reducibility. Two problems are said to be reducible to each other if there exists a polynomial algorithm to transform one problem to the other. Using this idea, a problem can be shown to be an NP problem if it can be shown to be reducible to another NP problem. Within the NP class, there is a large group of problems which are reducible to each other; the problems are called NP-complete problems. Some of these are the satisfiability, travelling salesman, set covering and language recognition problems. The implication of the existence of such a group is that if there is an efficient algorithm for any NP-complete problem, then there is an efficient algorithm for all the NP-complete problems.

1.3 AN OUTLINE OF THE THESIS

Three sets of problems in the NP-complete class are selected for study in this thesis; plant layout, scheduling and group technology. In chapter 2, a review of the two main analytical models, the quadratic assignment problem (QAP) and the maximal planar graph (MPG) which are normally used to solve the plant layout problem. In chapter 3, an interactive decomposition method is used in conjunction with a heuristic procedure to solve the QAP. Chapter 4 provides the detailed description of a set of heuristics for the MPG, implemented on a computer. Data structures for efficient implementations of these heuristics are also given. The heuristics, construction and improvement, are carried out in such a way that the need for a planarity testing procedure is avoided. It is believed that this is the first report of computer-implemented heuristics for the MPG. For group technology, it was felt that there was a need for a critical and comprehensive survey of the various methods that have been suggested during the last decade. Chapter 5 is the result of an attempt to fill this gap. In chapter 6, the main effort is concerned with an extension of a previously published algorithm, the Rank Order Clustering (ROC) algorithm. The new algorithm (ROC2) has a linear order of complexity and hence can be used to solve very large and realistic problems. A new relaxation procedure for bottleneck machines is also proposed. The new algorithm was implemented interactively and the tests that were carried out have shown that such an approach provides comparable or better solutions to published problems, with less effort, than those provided by existing methods. The sequence-dependent setup time scheduling problem (SDSTSP) is the subject of chapter 7. The SDSTSP is a problem which can be transformed into the well known travelling salesman problem (TSP). Various construction and improvement heuristics are discussed.

1.4 A NOTE TO THE READER

A brief explanation of the style of the presentation in this thesis is needed. The reader will find that formalized definitions, theorems and proofs are generally avoided, except where essential to subsequent discussions. The underlying concepts and ideas are explained in full, replacing the more familiar style of presentation. It is the author's belief that formalization, though necessary in many situations, is not always the best approach. The hope is that this method will provide a satisfactory explanation of the work carried out in this thesis in a more agreeable manner.

2 Plant Layout: Literature Survey

2.1 INTRODUCTION

Plant layout covers a wider range of activities than the simple process of laying out machinery. It involves many interrelated activities and items such as the products, operating equipment, storage space, material handling equipment, safety, personnel and all other supporting services. As Apple (1977, p7) suggests, the major objectives of plant layout are to

1 Facilitate the manufacturing process
2 Minimize material handling
3 Maintain flexibility of arrangement and operations
4 Maintain high turnover of work-in-progress
5 Hold down investment in equiptment
6 Make economical use of building cube
7 Promote effective utilization of manpower
8 Provide for employees' convenience, safety and comfort in doing the work.

Francis & White (1974, p34) suggest that "facilitate the organizational structure" should be included to the above list.

It is obvious from the list of objectives that plant layout is a highly complex problem. Many of the factors would be very difficult to measure in quantitative terms. It is unlikely that the plant layout problem can be described adequately by a mathematical model. This is one of the main reasons why, in spite of the efforts in the last few decades to develop mathematical models for the plant layout problem, practical approaches to tackling the problem are still largely qualitative in nature.

For the purpose of this survey, the approaches to the plant layout problem are divided into two categories: qualitative and quantitative. However, there is a considerable degree of overlap between the two. The qualitative approach is used in a method which relies primarily on visualising techniques to arrive at a solution, and only a limited number of solutions will be considered, due to the difficulties in arriving at a solution. The quantitative approach usually implies that explicit mathematical relationships between limited numbers of variables are formulated. Large numbers of alternative solutions are generated and evaluated to find the best layout, acccording to one or more objective functions. In most cases, the objective is usually a single materials handling cost function.

2.2 QUALITATIVE APPROACHES

Moore (1962, p114) suggests that the first major improvement in plant layout technique is to adopt the Time and Motion Study approach. The content of Hiscox's (1948) book tends to support this idea. El-Rayah & Hollier (1970) characterize the techniques of the earlier period as "one of developing flow diagrams and process charts for the orders judged to be dominant, and, with the aid of two dimensional templates and three dimensional scale models, alternative layout proposals were developed. It should be noted that the development and evaluation of these alternative layouts depended primarily on the judgement, intuition and experience of the layout analyst".

Cameron (1952) and Smith (1955) introduced the use of the *Travel Chart* in plant layout. The first step in this method is to make simplifying assumptions regarding the nature of the distance-volume matrix. By reallocation of machines, a new distance-volume matrix can be constructed and compared to the previous one. Reallocation is carried out until there is no obvious improvement. This approach can be seen as a simplified version of the quadratic assignment problem (QAP), with the distance as the number of rows (or columns) away from the main diagonal of the distance-volume matrix. It was the first attempt to use the large quantity of the material handling data in a concise way. As the number of calculations is large, a very limited number of alternatives can be considered in this way.

Sequence analysis (Buffa, 1955), as the name implies, is based on the analysis of the sequence of operations to be carried out on components. From this analysis, a "sequence summary" of how material flows between various work centres is developed. Other data, such as area requirements, are also collected. From inspection of these data an improved layout may be derived. The main advantage of this technique is that the data are handled subjectively, and hence alternative solutions can be proposed and evaluated quickly. The main drawback is that there is no obvious way that the data collected can be transformed into solutions; they depend entirely upon individual insights and manipulations.

There are other extensions to the sequencing method (Lundy (1955), Noy (1957), Llewellyn (1958) and Schnieder (1960)). In general, it is reckoned that they are not as useful as the Travel Chart method (El-Rayah & Hollier, 1970).

Muther (1961, 1962) introduces the concept of the "closeness-desired" rating and *relationship chart*. Closeness rating is a systematic method of taking into account various factors including material flow considerations. The closeness rating between two machines starts at the highly desirable *A*, progressively reduces to *E*, *I*, *O* and *U* and ends at *X* which is considered totally undesirable. By assigning values to all the machine pairs, a relationship chart (REL chart) is constructed. A *relationship diagram* (REL diagram) is drawn by shifting around various machines until the proper relationships, as indicated by the REL chart, can be obtained. The REL diagram together with the space requirement consideration will be the basis for the new layout.

The advantage of this method is that in the case where the flow of the material is not the only major factor, a meaningful layout could still be constructed. The two main disadvantages are the need to resort to subjective ratings and the lack of clear cut criteria for choosing among alternatives.

The major difficulty that is found in all the methods using the qualitative approach to plant layout is that the objective is rarely stated explicitly. Even when it is stated, the computational effort is usually too large to be carried out effectively by manual methods. This state of affairs was not satisfactorily resolved until the computer became more accessible in the early sixties.

2.3 QUANTITATIVE APPROACHES

There are two major mathematical models used in the study of plant layout, namely the quadratic assignment problem (QAP) and the maximal planar graph (MPG). In spite of intensive research in the past couple of decades, there has been very little progress made in the attempt to solve the QAP (Lawler 1975). To a lesser extent, the same can be said about the MPG. The major difficulty with the models is the combinatorial nature of the feasible solutions.

2.3.1 Quadratic Assignment Problem

The QAP, formulated as a generalized case of the linear assignment problem (Lawler, 1962), is defined as follows:

$$\text{Minimize } \sum_{i,j,p,q \in N} c_{ij}x_{pq} \qquad (2.1)$$

$$\text{subject to } \sum_{i \in N} x_{ij} = 1 \qquad (2.2)$$

$$\sum_{j \in N} x_{ij} = 1 \qquad (2.3)$$

$$x_{ij} = [0,\ 1] \qquad (2.4)$$

For a problem of n facilities, the problem is to determine values of n^2 variables x_{ij}, given the cost coefficient c_{ijpq} such that (2.1) is minimized. c_{ijpq} is the cost of handling material to be moved between the machine i, located at position p, and machine j located at position q. The equation (2.2) ensures that a machine is located only once, and the equation (2.3) requires that only one machine can be assigned to a particular location. The objective of the QAP is hence minimization of the material handling cost function only.

However in this form, the amount of storage for the cost matrix C alone will exceed 50K words for a modest 15 machine problem. Such a prohibitive memory requirement makes the earlier formulation by Koopmans & Beckmann (1957) more attractive as far as the use of computers is concerned. As the computer is absolutely indispensible in an attempt to solve QAP problems of any meaningful size, it is proposed that the Koopmans-Beckmann formulation is the subject of the discussion rather than Lawler's alternative. The Koopmans-Beckmann formulation is:

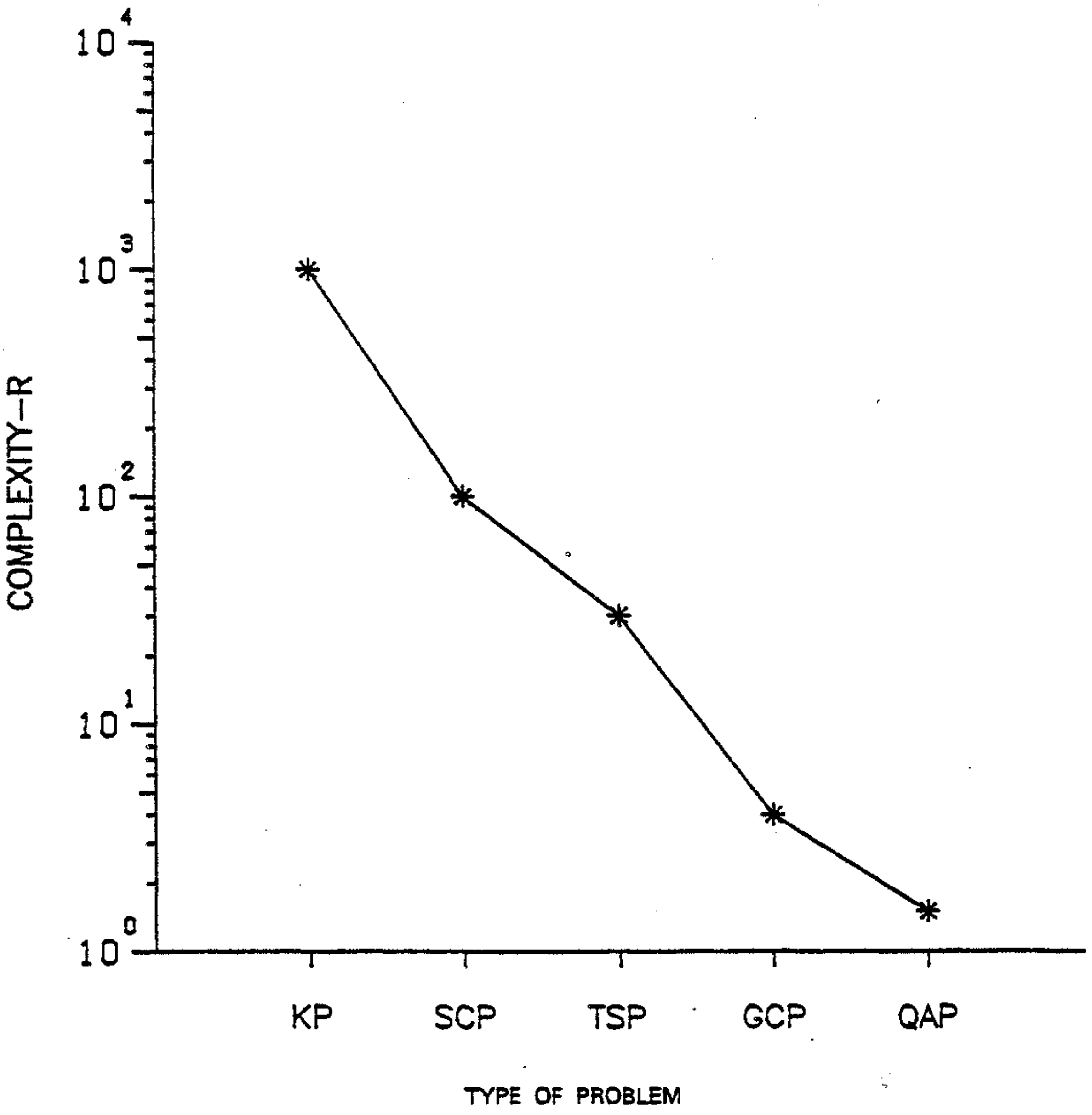
$$\text{Minimize } \sum_{1 \leq i < j \leq n} w_{ij} d_{a(i)a(j)} \qquad (2.5)$$
$$\text{subject to } (2.2) - (2.4)$$

w_{ij} is the material handling cost between machines i and j per unit distance, and is referred to below as the *weight*, following Francis & White (1974). $d_{a(i)a(j)}$ is the distance between machine i and machine j. $a(i)$, the assignment function, gives the present location of machine i. It can be seen from (2.5) that the evaluation of the objective function is more involved than that of the earlier formulation. The memory requirement of the coefficients is reduced from $n^4 + 2n^2$ locations to only $2n^2 + 2n$ locations. It can also be deduced that

$$c_{ijpq} = w_{ij} d_{a(i)a(j)} \qquad (2.6)$$
$$\text{where } a(i) = p$$
$$\text{and } a(j) = q$$

It should be noted that the original Koopmans-Beckmann formulation also includes a setup cost. This is to take into account the initial cost of having a facility at a particular location. This setup cost is usually ignored because, even in the simpler form, the QAP is intractably difficult.

The intractability of the QAP is well known. Tests on optimal procedures show that the QAP can be solved in "reasonable time" up to a 15 facility problem (Burkard & Shalman, 1978). In fact, there is no report of optimal solutions for a problem of over 15 facilities. The degree of intractability of the QAP is summarized in Figure 2.1 (after Christofides, 1977).



Empirical Complexity R is defined as follows:

$$R = A/E \tag{2.7}$$

A is the size of a problem that can be solved using the best known optimal procedure and *E* is the size of the same problem that can be solved by complete enumeration, for the same number of "evaluations". For one million function evaluations:

			R
KP	Knapsack Problem	20000/20	1000
SCP	Set Covering Problem	2000/20	100
TSP	Travelling Salesman Problem	300/10	30
GCP	Graph Colouring Problem	80/4	4
QAP	Quadratic Assignment Problem	15/10	1.5

Figure 2.1
Complexity of Combinatorial Problems

Land (1963) shows that the *n* facility QAP can be transformed into a TSP for a complete graph of $n(n-1)/2$ cities, subject to extra constraints. Hence, a 15 facility problem is equivalent to a 105 city TSP. Another major difficulty of this type of transformation is that the distance matrix generated is likely to be non-Euclidean.

Approaches to solving the QAP can be divided into two major groups: optimal procedures and heuristic procedures. Most of the optimal procedures use the branch and bound method. Gilmore (1962) and Lawler (1963) use linear assignment approximation in the bound calculations. Edwards (1977, 1980) extends the procedure further, but no computational results are reported. Christofides *et al* (1980), also using a linear assignment approximation, suggest a two stage lower bound calculation. Land (1963) and Gavett & Plyter (1966) suggest a TSP-like transformation in the bound calculation. Kaufman & Broeckx (1978) suggest the use of Bender's decomposition, however, apparently without a great deal of success. Christofides & Gerrard (1976) suggest a dynamic programming formulation for a specially structured graph.

It is generally recognized that the calculations of the lower bounds as suggested above have not proved successful (Christofides *et al*, 1980). These bounds are on average about 5% from the optimal solution, a gap far greater than for other combinatorial problems.

2.3.2 Improvement techniques

Heuristic procedures have been developed in response to the recognition of the difficulty in obtaining an optimal solution to the QAP. Most of them are based on a pairwise exchange algorithm of some kind, or alternatively use a method which is now called the *construction technique.*

The first *hill climbing* improvement heuristic for the QAP, named CRAFT, was suggested by Armour & Buffa (1963) and was subsequently expanded by Buffa *et al* (1964). In essence, CRAFT is a steepest pairwise interchange algorithm. Starting from a given layout it will consider the cost or benefit of switching locations of a pair of machines, which is given by the equation:

$$DTC_{uv}(\underline{a}) = \textstyle\sum_{i \in N} (w_{iu} - w_{iv})(d_{a(i)a(u)} - d_{a(i)a(v)}) \\ - 2w_{uv}d_{a(u)a(v)} \qquad (2.8)$$

w and *d* are the weight and distance matrices respectively.

CRAFT will consider all the possible $n(n-1)/2$ pairs of interchanges and then select the pair of highest benefit. Once the interchange is carried out, the whole process is then repeated until no further improvement is possible. The updating part of the algorithm has an $O(n^3)$ complexity. A three way interchange was also proposed by Buffa *et al* (1964). The number of possible three way interchanges is $n(n-1)(n-2)/6$, and the complexity of the updating part of the algorithm is $O(n^4)$.

Even though three way interchange has resulted in a better final solution, the computing time could become a serious problem. For a twenty facility problem, the two way interchange algorithm will require about 5% of the time needed by the three way one. Los (1978), using fast updating of the three way interchange, concludes that because of the time and storage requirements, the method is not applicable to problems of size n greater than twenty-four. The quality of the solution using the three way interchange is usually only marginally better than those using the pairwise interchange. However, the combination of the two, using them in tandem, produces even better results.

The main difficulty with CRAFT is that the amount of time required to find the largest possible gain between each iteration is quite expensive, of the order $O(n^3)$. As the number of iterations required is $O(n)$ (Los, 1978), the original pairwise interchange algorithm of CRAFT has a time complexity of $O(n^4)$. For the three way interchange algorithm, the complexity becomes $O(n^5)$. In an effort to overcome this difficulty, various modifications of CRAFT have been introduced.

Vollman *et al* (1968) suggest a heuristic to overcome some of the difficulties in using CRAFT. Instead of calculating the possible benefits of all the interchanges, it concentrates during the first phase on the two machines which have the highest cost $P_i(a)$:

$$P_i(\underline{a}) = \sum_{j \epsilon N} w_{ij} d_{a(i)a(j)} - \sum_{j \epsilon K} w_{ij} d_{a(i)a(k)} \tag{2.9}$$

$$d_{a(i)a(k)} < \text{a constant} \tag{2.10}$$

From these two preselected facilities, two lists of the remaining machines are constructed. Interchanges between the preselected facilities and the ones in the lists, are carried out only if they lead to a cost reduction. In phase two, all possible interchanges are considered. The difference between this procedure and CRAFT is that the procedure will exchange two facilites and update the assignment vector as soon as the interchange is beneficial, whereas CRAFT will only exchange the pair which give the highest benefit. Only two complete cycles of phase two will be considered.

This heuristic is undoubtedly faster than CRAFT, however there are many points which need further clarification. Firstly, the question of selection of the constant in the equation (2.10) is left unanswered. Secondly, there is no adequate explanation of why there are only two iterations during phase 2. The claim that the heuristic provides solutions which are comparable to those produced by CRAFT is largely unsubstantiated.

FRAT (Khalil, 1973) can be seen as an attempt to systematically improve the idea suggested in the previous heuristic. Firstly, only movements over a limiting distance are considered. This limiting distance is initially set to be the difference between the maximum and the minimum distances travelled. The limiting value is successively decreased during the iteration process. Secondly, only limited combinations of all the possible $n(n-1)/2$ interchanges are considered. The main candidates, two are suggested by Khalil, are then considered for interchange with all remaining facilities in the same manner as that of CRAFT. The number of possible interchanges reduces to $2n-4$.

The Terminal Sampling Procedure (Hitchings, 1973; Hitchings & Cottam, 1976) adopts a slightly different strategy to that of FRAT. Two facilities are again preselected according to the criterion of Vollman *et al* (1968), and the 2*n*-4 interchanges between these and the remaining facilities are considered in the same way as those of CRAFT. Once no further improvement can be made on the basis of exchanging the two primary candidates alone, the full CRAFT procedure is then augmented.

Both approaches claim to provide better final solutions that those provided by CRAFT. These claims are based on the solutions to the test problems first suggested by Nugent *et al* (1968). Leaving aside the issue of time complexity, it is difficult to see, at least from a theoretical point of view, why FRAT or the Terminal Sampling Procedure should in general provide better solutions as has been claimed. Both approaches search only small portions of the solution space searched by CRAFT, and both utilize the same maximum pairwise interchange principle as CRAFT does.

The Terminal Sampling Procedure also backtracks to consider all the tie values. This is equivalent to having many more starting solutions than those indicated.

S-ZAKY (Abdel Barr & O Brien, 1976; Abdel Barr, 1978) adopts a slightly different line of attack. Unlike CRAFT, which only considers one interchange out of all the possible pairs in every iteration, S-ZAKY will consider the exchange of the 3 pairs of facilities which provide the highest overall benefit. By carrying out a multi pairwise interchange, it is hoped that the number of iterations required will be reduced. However, the overall complexity is still the same order as CRAFT.

Comparison of algorithms of similar speeds of execution made by converting run times on different computers via the use of constant factors is very unreliable. The speed of a code, as compared to the speed of an algorithm, depends on the compiler used, the operating system enviroment and programming style, as well as the computer in use. Only when these main factors are very similar, can the speeds of the codes be used for useful comparison of algorithms.

Problem	CRAFT	TSP		S-ZAKY	
	(secs)	(secs)	(% of	(secs)	(% of
			CRAFT)		CRAFT)
1	0.7	0.7	100	0.6	86
2	0.7	0.8	114	0.6	86
3	1.0	0.8	80	0.9	90
4	1.2	1.0	83	1.1	92
5	2.6	2.2	85	2.3	88
6	4.6	3.8	83	5.0	109
7	11.3	8.2	73	9.8	88
8	53.9	35.5	66	42.3	78

Time in *PRIME 400* cpu
The problems are suggested by Nugent *et al*
TSP - Terminal Sampling Procedure.
Adopted from Abdel Barr (1978)

Table 2.1
Run time comparison of three algorithms

Table 2.1 shows a comparison under which these conditions are fulfilled (Abdel Barr, 1978). It compares the run times used by CRAFT, the Terminal Sampling Procedure and S-ZAKY to solve the eight problems suggested by Nugent *et al* (1968). The table tends to confirm the idea that all three are of the same order of complexity. It also confirms that 'the Terminal Sampling Procedure is the fastest of the three.

There are many other variations to the same basic idea of pairwise interchanges (Ritzman 1972; Parker 1976; Burkard & Shatman 1978; Lewis & Block 1980; Liggett 1981). Most of these carry out a limited number of searches as in the Terminal Sampling Procedure, hence they are usually faster than CRAFT. The qualities of the solutions, however, are very much more difficult to interpret.

Los (1978) shows a set of recurrent relationships which exist in the updating part of the CRAFT algorithm. These relationships show that the updating part of the algorithm has the complexity of $O(n^2)$ for a pairwise interchange routine, and of $O(n^3)$ for a three way interchange routine. The overall complexity of the pairwise interchange algorithm is reduced to $O(n^3)$, the same as FRAT and the first phase of the Terminal Sampling Procedure. However Los does not compare the new codes with other approaches.

Hillier (1963) and Hillier & Connors (1966) suggest the concept of a *Move Desirability Table* (MDT). The MDT of a machine, with respect to a particular layout, is the potential saving in the material handling cost of making one facility occupy the same location as another. Locations under consideration are restricted to the ones along the same row or the same column or along the diagonals. This presupposes that the layout is on a rectangular grid system. In spite of this rather unusual concept, MDT has proved surprisingly robust in many situations (Ritzman, 1972).

All the pairwise interchange or improvement techniques described previously are deterministic in character: given an initial layout, the algorithm will always generate the same answer to a particular problem. Nugent *et al* (1968) introduced a sampling scheme which will select at random, an interchange from all the beneficial pairs. In spite of the increase in the complexity of the algorithm, the solutions to the test problems do not significantly differ from solutions obtained by deterministic algorithms. There is also very little theoretical justification that such a sampling scheme would produce better solutions than comparable deterministic algorithms.

2.3.3 Construction Techniques

All improvement heuristics have one feature in common, they assume the availability of an initial layout. If there is none, a randomly generated one is often used. Construction techniques, as the name implies, generate a layout in a systematic attempt to keep the objective, as specified by the equation (2.5), as low as possible.

Modular Allocation Technique (MAT) (Edwards *et al*, 1970) is one such algorithm. The underlying idea of MAT is that two facilities should be placed as close together as possible, so long as there is no conflict with previous allocations. This is carried out with the help of two vectors generated by sorting the distances in an ascending order and the weights in a descending order. The complexity of MAT is $O(n^2)$, and hence it can be used to generate a useful starting solution for large problems.

Lewis & Block (1980) extend the MAT approach further by multiplying both distance and weight vectors by a function which accounts for the overall movements and distances. The remainder of the procedure is identical to that of MAT. The complexity is still of the $O(n^2)$, though it is expected to be slower than MAT. Performance of both algorithms is very similar, but there are some indications that the new procedure has a slight edge in large problems.

Graves & Whinston (1970) suggest a construction approach which attempts to take into account all the global interactions in a way similar to the branch and bound method. As exact bound calculations are expensive, they suggest the use of expected values. An assignment will be chosen in such a way that the expected value of the remaining assignment is minimised. The complexity of the algorithm, to be called the GW algorithm, is $O(n^3)$. As the algorithm is a one pass heuristic, the procedure is adequately fast for very large problems. Liggett (1981) extends the procedure slightly in order to generate more than one final solution. This is usually carried out at the earlier stage of the heuristic when the expected value of the remaining assignment is very close to the best choice (0.5% is used).

Parker (1976) suggests a *Best Match* heuristic which is based on the idea that the facilities which have higher load movement should be placed towards the centre. The method is slightly revised by Burkard & Stratmann (1978) who apply the idea to restricted subproblems. Starting from a seed, facilities are added on in such a way that the objective function is minimised, taking into account interaction between assigned facilities only.

2.3.4 Empirical Complexity and Test Problems

One of the major problems in the use of heuristic approaches to the QAP is the complete lack of any worst case analysis of the published algorithms. Hence, comparison between various heuristics is based on their performances on artificially constructed problems. The most frequently used test problems are the eight problems suggested by Nugent *et al* (1968). The problems range from five to thirty facilities. The layout assumes a rectangular shape whenever possible. The material movements or flows between the facilities range from 0-10. These flow patterns are kept roughly to the same *flow dominance* (*f*) figure:

$$f = 100n^2 \sqrt{(\textstyle\sum_{i,j \in N} w_{ij} - ((\sum_{i,j \in N} e_{ij})^2/n^2)/(n^2 - 1))} / \; (\textstyle\sum_{i,j \in N} w_{ij}) \qquad (2.11)$$

Block (1979) derived the theoretical lower and upper limits of the flow dominance. A lower bound is reached when the flow pattern is of the flowshop type.

$$f_{lb} = 100n\sqrt{(n^2 - n)} \qquad (2.12)$$

The maximum limit is reached when all the flows are in the same direction.

$$f_{ub} = 100n(n^2 - n + 1)/\;((n - 1)(n^2 - 1)) \qquad (2.13)$$

Vollmann & Buffa (1966) suggest that layout problems with flow dominance over 200% can probably be solved by inspection, with results comparable to those achieved by CRAFT. This guideline is an oversimplification. The effect of the size of the problem on the complexity of the problem is not of a quadratic order, as indicated by the equation (2.11). Block (1979), in an effort to overcome some of the shortcomings, defines the *Complexity Rating* C_f as:

$$C_f = 100(f_{ub} - f\,)/(f_{ub} - f_{lb}) \qquad (2.14)$$

This definition of complexity rating is unsatisfactory and misleading, as it suggests the complexity of the problem to be of an order less than $O(n)$. Results from computational complexity theory and the failure to achieve optimal solutions for problems with more than fifteen facilities, in spite of the vastly improved computer speeds of the last decade, firmly indicate that the complexity of the QAP is far more than that suggested by Block.

In spite of this weakness, flow dominance is still a useful measure, provided that it is used to compare problems which have the same number of facilities. Attempts to infer that Nugent's problems have roughly the same degree of difficulty, as they have roughly the same flow dominances, are inaccurate.

2.3.5 Comparative Results

Claims that various heuristics provide better solutions than CRAFT must be treated with caution. The implementational aspects can be very important as was indicated earlier. This is compounded by the characteristics of the test problems used. Most of the claims are based on the results of Nugent's test problems which are too small and have fairly uniform flow patterns, as measured by the low flow dominances. Liggett (1981) points out that for the Nugent's as well as Steinburg's problems, it does not matter very much what kind of strategy is used in the pairwise exchange procedure, the final results are of similar quality.

More extensive tests were carried out by Ritzman (1972) and Parker (1976). Ritzman uses a total of 26 problems, whereas Parker employs 75 problems. Parker varies the flow dominances considerably. Both conclude that on average, using random starting layouts, CRAFT produces better solutions than other improvement methods they have tested.

For construction techniques, it is generally agreed that the GW heuristic is better than all the others tested (Parker, 1976; Liggett, 1981). The GW heuristic also saves considerable computing time when it is used in tandem with an improvement heuristic as compared with the use of random starting layouts. Liggett (1981) reports savings ranging from 40% to 100% for larger problems. More substantial savings are reported by Parker (1976).

2.3.6 Human Interactions

Vollmann & Buffa (1966) suggest that problems with flow dominance of over 200% can be solved by inspection, and results comparable to those achieved by CRAFT can be obtained. Scriabin & Vergin (1975) suggest that the traditional qualitative aids used by industrial engineers would enable the planner to produce better layouts than computer generated solutions such as those produced by CRAFT. However, their experiment has been subject to many criticisms (Buffa, 1976; Block, 1977; Trybus & Hopkins, 1980). One of these is that the flow dominances, around 250%, are high and hence would favour manual techniques. A more serious charge is that the subjects were given the results generated by the computer in advance, and hence targets to beat. As there are no records of the number of attempts each subject made, a fair comparison is difficult. Ironically, the numerical evaluations were carried out by a computer.

Block (1977) shows that in solving Nugent's problems, the average flow dominance of which is around 115%, the subjects perform as well as CRAFT up to the 8 department problem. When the size becomes larger, CRAFT's performances are far superior to those of the subjects. Trybus & Hopkins (1980) produce similar results when the flow dominance is around 150%. The differences become smaller as the flow dominance increases to 250% or reduces to around 40%.

From these results, there is little doubt that man alone, without the aid of a computer, would be unlikely to outperform heuristics, like CRAFT, for large problems, due to the sheer number of possible solutions as reported by Scriabin & Vergin (1975). However, if we reinterpret the results as the combined effort of man and machine, there are indications that this might produce a more useful result than the one generated by the heuristic alone.

2.4 MAXIMAL PLANAR GRAPH

The maximal planar graph (MPG) problem is formulated as an extension of the use of the REL chart (Muther, 1961, 1962). The MPG is defined as: Given a complete graph $G(V, A)$ with no negative arc weight c_{ij}, find a planar partial graph with maximum total arc weight (Christofides *et al*, 1980). A graph $G_p(V, A_p)$ is a partial graph of the graph $G(V, A)$ if A_p is a subset of A. A graph is said to be planar if it can be drawn in a plane so that its edges intersect only at their ends. A maximal planar graph is a graph to which an arc cannot be added to without it losing planarity. The MPG can be formalized as:

$$\text{Maximize } \textstyle\sum_{1 \le i < j \le n} c_{ij} x_{ij} \tag{2.15}$$

$$\begin{aligned} \text{subject to } x_{ij} &= 1 \text{ if } a_{ij} \in A_p \\ &= 0 \text{ otherwise} \end{aligned} \tag{2.16}$$

$$G_p(V, A_p) \text{ is planar.} \tag{2.17}$$

In the use of the REL chart, the relationships are considered to be ordinal. An ordinal scale of measurement is a ranking scale and hence further manipulations, such as addition, on these relationships are not appropriate. In order that the MPG could be used in this context, the relationships must be at least of the interval type. Non-negativity of the arcs is necessary in the case where the optimal solution is required.

The underlying idea of the MPG can be traced back to the development of the REL chart. However, the explicit recognition and the use of the MPG model is due to Krejcirik(1968, 1969). Seppanen & Moore (1970) investigated the underlying structure in some detail. A heuristic was proposed based on the use of a maximal spanning tree as a starting point (Seppanen & Moore, 1975; Moore, 1976). Arcs are then systematically added until the graph becomes maximal planar. Foulds & Robinson (1976) suggest a branch and bound scheme to solve the MPG optimally. The major drawback is that the only bounding procedure enforced is the planarity condition. It is unlikely that the bounding scheme is effective enough for large problems. Recognizing the computational difficulty in checking the planarity of a graph, Foulds & Robinson (1978) suggest two construction heuristics which avoid the planarity testing altogether, based on the idea first suggested by Hopcroft & Tarjan (1974). By utilizing the property of a maximal planar graph that every face of the graph is triangular, the graph is built up by constructing only triangular faces. Both heuristics use a tetrahedron as a starting point. Geometrically, a tetrahedron is made up with three triangles. In the

S construct, vertices are inserted in the descending order of the sums of weights of the arcs incidence to the vertices, so that the increase in the total weight is maximized. In the *R construct,* a vertex is added to a triangular face if the difference between the highest and second highest benefits is maximum. Both heuristics have the computational complexity of the same order, $O(n^2)$.

Improvement techniques were also suggested by Foulds & Robinsons (1976). They are essentially a greedy algorithm. The procedures were implemented manually, and depended heavily on the ability to visualise the intermediate results. There are no suggestions as to the coding aspect of the algorithms to overcome the topological problem, which must be solved if the techniques are to be implemented via a computer.

Baybars (1979) formulated the MPG as an integer programming problem. The formulation is, however, so complex that it is unlikely to lead to a reasonable computational scheme (Christofides *et al,* 1980). A branch and bound procedure is suggested by Christofides *et al* (1980). The bound is calculated by a Lagrangean relaxation procedure. The average computing time to achieve an optimal solution for a randomly generated problem of fifteen vertices is about thirty five *CDC 7600* seconds.

In addition to the attempt to solve the MPG as formulated by equations (2.15-2.17), there are other published heuristics for solving the MPG with additional constraints. These usually include the space and shape requirements. The heuristics are primarily construction procedures, with additional ad hoc rules for handling the extra constraints. They are aimed primarily at achieving sensible solutions quickly rather than attempting to optimise the results as such (Muther & McPhearson, 1970; Moore, 1973). A survey (Moore, 1977) of the usage of these heuristics suggests that they are primarily used for scoring and providing alternative layouts. Even then, there were criticisms expressing dissatisfaction with the quality of the generated solutions.

3 An Interactive Approach to the QAP

3.1 INTRODUCTION

There are two major features of the QAP which are not treated explicitly by the approaches reviewed in the previous chapter: namely, the sparsity of problems, and the duplication of machines. These features are common in most real life problems: the material flow to and from a particular machine is restricted to a small subset of the other machines. It is also common to find several centre lathes or vertical milling machines in the same shop. These practical aspects indicate that a partitioning approach to the QAP may be beneficial. This chapter provides an account of how an initial layout of the QAP may be generated effectively by the use of a partitioning algorithm.

The improvement algorithm used in this chapter is CRAFT, which is the most general pairwise exchange algorithm, with the updating procedure suggested by Los (1978). This combination has proved to be sufficiently fast for experimental purposes; the 20 vertex problem suggested by Nugent *et al* (1968) was solved, on average, in less than one second on a *CDC Cyber 174.*

3.2 SOME THEORETICAL CONSIDERATIONS

Pairwise exchange heuristics have empirical complexities of $O(n^3)$ or more. Hence, a partition into smaller subproblems might be anticipated to lead to a substantial saving in the computing time required to solve a problem. It should be noted that such a saving could only be achieved without sacrificing the quality of the final solution if the problem could be partitioned into groups with few material movements between them. An algorithm that may be used for partitioning the problem is the ROC2 algorithm, which is discussed in detail in chapter 6. The ROC2 algorithm is an interactive clustering method for grouping machines and associated components, which can be extended to solve similar problems where group membership is required. It also contains features for dealing with the duplication of machines, and for exploiting the sparsity of a problem. Consequently, it can be used to investigate the partitioning of the QAP.

3.3 AN EXPERIMENT IN INTERACTIVE LAYOUT USING THE ROC2 ALGORITHM

The objective of the experiment is to determine whether a sparse QAP that has underlying group structure can be solved more efficiently with the use of partitioning or without. To construct a test problem, a weight matrix is generated from the machine-component matrix first used by Burbidge (1973). This is illustrated in Figure 6.3.1 (page 72): the numbers between brackets represent the row numbers; the numbers next to the row numbers are the machine numbers. The weight (as defined on page 6) between any two machines is represented by the number of components which visit both of them; for instance, the weight between machines 1 and 2 is two, comprising the components in locations 37 and 42. A partitioning solution to the problem of Figure 6.3.1 using the ROC2 algorithm is represented in Figure 6.3.4 (page 75). The solution is achieved interactively and is based on the assumption that duplication of some machines is possible. In this chapter, the emphasis is on the grouping of machines and hence adjacency of rows is of primary interest.

It can be seen that machines in rows 1 to 4 of Figure 6.3.4 form a distinct group and are independent of the rest, since all the machines required for the making of the components in locations 1 to 7 can be found within this group. In fact only component 9 (location 29) requires machining in two groups (as represented by an asterisk). A weight value of 10 units was arbitrarily assigned to the inter-group movement between machine 5 in row 13 and machine 11 in row 18, which is considerably higher than the weight value for an intra-group movement. A higher value is chosen for two reasons: firstly to reflect an additional cost associated with inter-group movement, as is likely in practice; and secondly to provide an additional incentive for the two machines, and their associated groups, to be located near each other.

For identification purposes in this chapter, some of the duplicated machines in Figure 6.3.4 were renumbered, since each machine has a different pattern of material movements. Machines 6 in rows 8 and 17 were renumbered as machines 17 and 18 respectively. Similarly, machines 8 in rows 9, 16 and 19 were called 19, 20 and 21 respectively. The four machine groups in Figure 6.3.4 can now be identified as follows: machines 10, 7, 6 and 8; machines 9, 2, 16, 17, 19, 14, 1, and 3; machines 5, 4, 15, 20 and 18; machines 11, 21, 13 and 12.

Three alternative *configurations* for the layouts are used, and are illustrated in Figures 3.1-3.3. (The number at the top right hand corner of each square is the location number. The number in the centre of the square is the machine that has been assigned to that location. The dotted lines indicate group boundaries). The first configuration, shown in Figure 3.1.1, consists of 24 locations arranged in 4 rows: Three dummy machines are required, machines 22, 23 and 24; there is no flow to or from these machines. This configuration allows all machine groups to be situated in a blocklike fashion. It can be seen as an extension of the second configuration, the 16 location layout, shown in Figure 3.2, which represents the original problem in which no duplication of machines is allowed. The third configuration, a 21 location layout shown in Figure 3.3, is used to investigate the potential benefit of partitioning when a blocklike layout cannot be readily achieved. A distance matrix for each of the three configurations was generated by calculating the rectilinear distance between any pair of locations, as suggested by Nugent *et al* (1968). For example, in Figure 3.1.1, the distance between locations 1 and 4 is three, and the distance between locations 1 and 10 is four. Similarly, the distance between locations 1 and 16 is five. The distance and weight matrices of the 24 location problem are shown in Appendix A (page 119).

To construct the initial layout, the partitions generated by the ROC2 algorithm (Figure 6.3.4) are used. There are four groups, two of which are independent. The initial layout is then constructed manually. The first stage of the construction is to consider the relative spatial arrangement of the groups. It is preferable to assign larger groups early on, as it becomes progressively more difficult to assign them later. For example, the two larger groups in the lower half of Figure 3.1.1 were assigned first. The second stage is to decide on the layout of machines within each group, taking into account any external flow required. The initial layouts of the 24 and 21 location problems constructed manually in this way are shown in Figures 3.1.1 and 3.3.1 respectively. These initial layouts are then solved in two steps. Firstly, each group of machines within the same boundary (shown as a dotted line) is solved as a separate sub-problem using CRAFT. In the second step, the solutions to the sub-problems are combined to provide a new starting layout for the whole problem and this is then solved, again using CRAFT, as a single problem.

Ten random layouts are also generated for each configuration for comparison. These are used as starting layouts and are solved using CRAFT without any reference to any group membership.

The result of using the manual layout of Figure 3.1.1 as the starting condition for the 24 location configuration is shown in Figure 3.1.2 with a total material handling cost (as defined by equation 2.5) of 238. The execution time was 0.41 seconds. (The same solution is achieved if the first step in the solution method described previously is ignored, at the expense of a 20% increase in the computational time.) This result compares favourably with the results obtained using random starting layouts; the best of these has a total material handling cost of 240, and the average cost is 248.5. The average execution time in the random layout cases is 1.46 seconds, the minimum value being 1.1 seconds. The difference between these results indicates that CRAFT cannot be relied on to detect the underlying structure of the problem. The results for the 21 location configurations are slightly more encouraging as far as the pairwise exchange procedure is concerned: out of the ten random starting conditions CRAFT produces two solutions equal to the ones achieved by the use of the manual layout starting plan, with a cost of 244. However, the execution times required using the random starting layouts are about three to four times that required using the manual solution. The solutions and execution times of the 21 and 24 location configurations are shown in Tables 3.1 and 3.2. The cost of the best solution for the 16 location configuration using random starting layouts is 266, which is more than 12% higher than the cost of the best solution obtained in the 24 location configuration, demonstrating the potential savings to be made in material handling costs if duplication of machines is allowed.

3.4 CONCLUSIONS

The results from this short experiment seem to indicate that in the case where an underlying group pattern exists, pairwise exchange routines such as CRAFT very often fail to detect the underlying relationships, and human interactions are useful in such cases. The benefits of human interaction are twofold; firstly, superior final layouts are usually obtained, and secondly, the computing time required is considerably reduced. This is not to say that human performance is generally better than that of heuristics as claimed by some authors. Both man and heuristics perform different but complementary roles, and the results obtained using both should be superior to those achieved by one or the other alone. It is also notable that the benefit of obtaining prior solutions to sub-problems is not as great in this example as was anticipated. This is probably due in part to the fact that in the problem considered here the manual solutions are close to the local optima, and hence the iteration times are artificially lower than in a general case. The effect of this would be accentuated by the fact that CRAFT is relatively more expensive in the setting up stage than in the iteration stage.

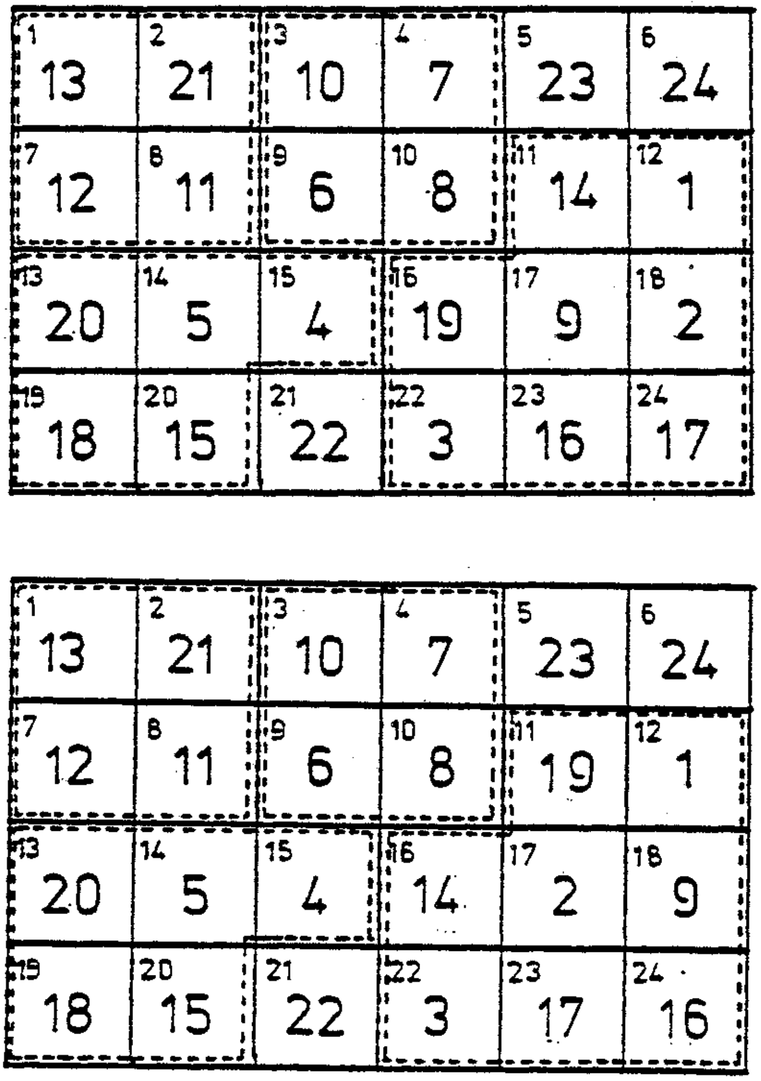


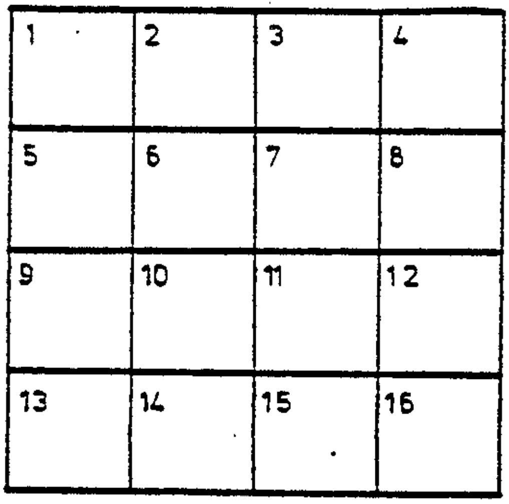
Figure 3.1.1

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

Figure 3.1.2

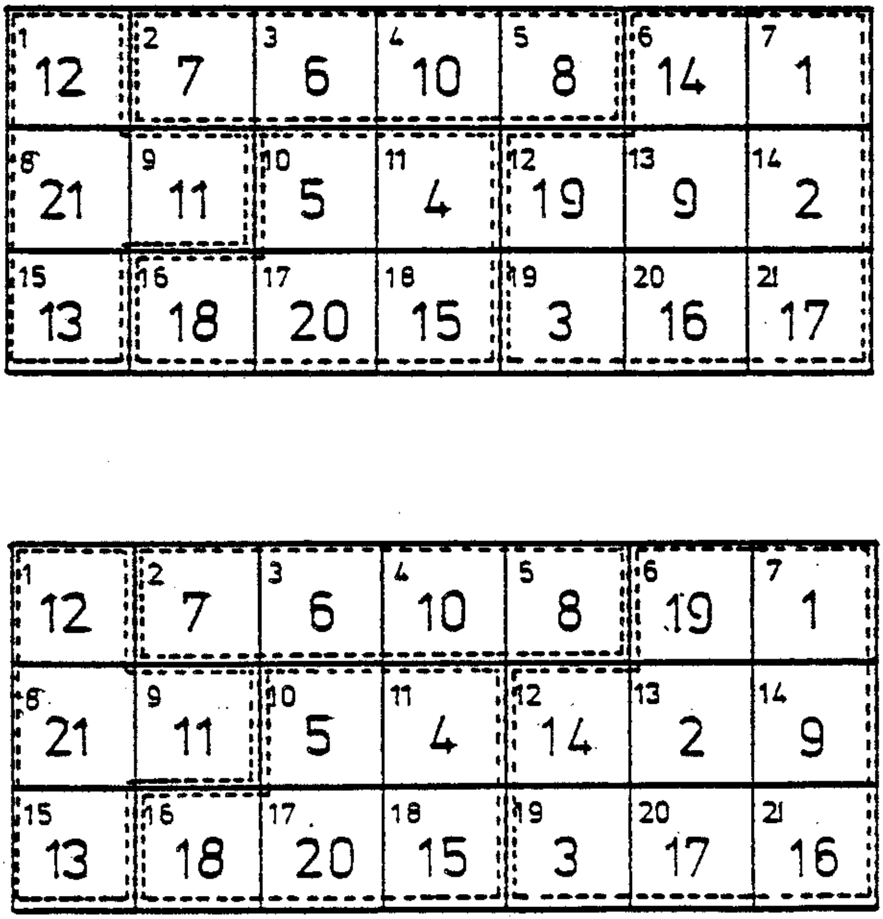
1 13	2 21	3 10	4 7	5 23	6 24
7 12	8 11	9 6	10 8	11 19	12 1
13 20	14 5	15 4	16 14	17 2	18 9
19 18	20 15	21 22	22 3	23 17	24 16

Figure 3.1
Layouts for the 24 location configuration



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Figure 3.2
The plan for the 16 location configuration



1 12	2 7	3 6	4 10	5 8	6 14	7 1
8 21	9 11	10 5	11 4	12 19	13 9	14 2
15 13	16 18	17 20	18 15	19 3	20 16	21 17

Figure 3.3.1

1 12	2 7	3 6	4 10	5 8	6 19	7 1
8 21	9 11	10 5	11 4	12 14	13 2	14 9
15 13	16 18	17 20	18 15	19 3	20 17	21 16

Figure 3.3.2

Figure 3.3
Layouts for the 21 location configuration

PROBLEM IDEN.	FINAL COST	NO. OF ITERATION(S)	EXEC. TIME (CYBER174 SEC)	
manual	238	0	0.412	(with subproblems)
manual	238	3	0.521	(without subproblems)
1	262	15	1.450	
2	240	17	1.595	
3	249	15	1.480	
4	243	17	1.654	
5	253	11	1.137	
6	243	17	1.603	
7	244	15	1.443	
8	249	16	1.523	
9	259	12	1.228	
10	243	16	1.509	

Table 3.1
The solutions to the 24 location configuration

PROBLEM IDENT.	INITIAL LAYOUTS											
1	2	14	13	3	9	4	18	20	15	16	7	5
	10	8	6	1	22	21	12	17	23	24	11	19
2	18	3	7	12	22	8	13	20	9	23	11	24
	21	16	6	4	1	2	14	19	10	17	5	15
3	23	8	21	10	18	24	9	15	4	3	2	22
	6	16	13	12	17	14	7	5	19	11	1	20
4	8	20	4	9	17	3	22	16	24	12	1	15
	10	18	23	11	19	7	14	13	21	2	5	6
5	13	16	21	14	2	22	15	5	10	8	9	24
	3	19	18	7	11	1	23	12	4	17	6	20
6	9	12	7	16	6	22	3	14	18	23	11	20
	13	8	15	21	1	24	19	10	4	17	2	5
7	6	21	20	9	19	12	4	16	14	11	5	17
	23	18	22	24	13	8	15	1	3	10	2	7
8	18	6	2	20	24	9	22	8	13	17	21	5
	19	7	12	23	16	1	15	3	4	10	14	11
9	16	12	9	20	13	5	17	19	8	15	21	6
	1	2	24	22	7	23	18	14	4	11	3	10
10	23	14	15	18	9	19	22	16	6	13	7	4
	17	2	11	1	21	10	5	20	24	3	12	8

Table 3.2
Random starting layouts for the 24 location configuration

PROBLEM IDEN.	FINAL COST	NO. OF ITERATION(S)	EXEC. TIME (CYBER174 SEC)	
manual	244	2	0.400	(with subproblems)
manual	244	3	0.372	(without subproblems)
1	252	12	0.929	
2	259	14	1.027	
3	252	13	0.977	
4	244	13	0.980	
5	244	14	1.008	
6	249	14	1.029	
7	267	17	1.202	
8	252	10	0.784	
9	248	13	0.976	
10	249	12	0.897	

Table 3.3
The solutions to the 21 location configuration

PROBLEM IDENT.	INITIAL LAYOUTS										
1	2	13	11	3	8	4	16	18	12	17	14
		6	5	9	7	1	20	19	10	15	21
2	7	6	16	20	14	1	11	18	13	9	5
		19	10	15	21	12	17	2	4	3	8
3	3	7	9	4	15	12	13	14	21	6	16
		10	19	5	1	20	8	17	11	18	2
4	13	8	14	18	21	6	15	16	17	12	19
		3	1	10	9	11	2	4	7	5	20
5	6	8	13	11	20	16	1	12	15	10	3
		21	18	14	7	4	2	19	9	17	5
6	19	17	3	12	18	2	1	10	4	6	15
		11	8	16	5	21	9	7	14	13	20
7	19	11	15	12	18	7	13	1	5	6	21
		20	14	16	17	2	8	4	9	10	3
8	21	9	12	15	8	6	10	4	7	13	19
		2	18	16	20	5	3	1	11	14	17
9	9	12	16	11	10	2	13	17	5	8	18
		19	21	7	1	15	3	6	20	14	4
10	20	9	16	11	4	15	3	2	13	5	6
		1	12	10	17	21	14	7	18	19	8

Table 3.4
Random starting layouts for the 21 location configuration

4 Maximal Planar Graph Heuristics

4.1 INTRODUCTION

Heuristic approaches to the MPG problem, like their counterparts for the QAP, can be divided into two classes; namely, construction and improvement heuristics. Whereas the construction procedures of the QAP can often be disregarded, this is generally not an option in the case of the MPG problem. As the graph required has to be both planar and maximal, a certain procedure must be adopted to ensure that these two constraints are met. During the improvement phase, any exchange of the arcs or vertices must also ensure that the constraints are not violated. It is relatively simple to ensure that the planar and maximal conditions are maintained if the graph can be visualized on a sheet of paper. To implement the scheme using a computer, a way must be found to store the topological information of the graph. As far as can be ascertained, there is no previously published heuristic implementation of the MPG problem using a computer.

4.1.1 Some Properties of a Maximal Planar Graph

It can be shown that for all maximal planar graphs if *v*, *a* and *f* are the numbers of the vertices, arcs and faces respectively, then:

$$a = 3(v\text{-}2) \tag{4.1}$$

$$f = 2(v\text{-}2) \tag{4.2}$$

$$\text{All faces are triangular.} \tag{4.3}$$

A face is the region enclosed by arcs and there are no arcs or vertices in its interior.

Consider the maximal planar graph in Figure 4.1. There are four vertices and hence there should be six arcs and four faces. The number of arcs can be easily verified. The four faces are *ABD, ACD, BCD* and *ABC*. *ABC* refers to the outer triangular face, which surrounds the tetrahedron. The triangularity of the faces is also confirmed. Hence, it can be concluded that the graph in Figure 4.1 is a maximal planar graph.

In a computer implementation, these properties, represented by equations (4.1) to (4.3), can be used to ensure that the graph is maximal and planar.

4.1.2 Design and Implementation Considerations

The speed and storage requirements of a computer program often require a careful trade-off. The approach suggested by Seppanen & Moore (1970) requires a comparatively small amount of topological data. The likely penalty is an excessive computational requirement. If a lot of redundant information is kept, it would result in unacceptable storage requirements for larger problems.

Apart from classifying heuristics according to purpose, as described earlier, heuristics for the MPG problem can also be classified by strategy. The first group relies on the use of a planarity testing procedure and hence only adjacency of nodes is required. This is generally used by optimal procedures. Seppanen & Moore (1970) favour such an approach. Alternatively, by keeping extra information regarding the arcs and the faces, the planarity testing can be disregarded. One such approach was suggested by Hopcroft & Tarjan (1974), in a slightly different context, and adopted for the MPG problem by Foulds & Robinson (1978). However Foulds & Robinson implement the heuristic manually and do not attempt to work out the data required for a computer implemented heuristic.

4.2 PROGRAMMING LANGUAGE SELECTION AND DATA STRUCTURES

In order that the orientation of the graph can be easily recognised by a computer implementation, the following data fields are needed:

Node information: all the adjacent nodes.
Arc information: two end nodes, adjacent faces.
Face information: the three vertices.

An adjacent face of an arc is a face which has the arc as part of its boundary. There are two adjacent faces for every arc.

These requirements suggest that the use of a language with data structuring facilities would be an advantage, for it is usually the case that most of the data fields of a particular group of information are accessed together. Pascal is one such language. It also has a facility to define data types, and as such it is ideally suited for this purpose. We can define nodes, arcs and faces in a way similar to their representations on a sheet of paper. These facilities allow a program to be developed that is analogous to the manual implementation on a sheet of paper. For reasons of computational efficiency, extra fields of data are added and the following data types used:

```
ANodeTable = PACKED RECORD
                 CASE active: BOOLEAN OF
                     TRUE: (pointer to insertion information);
                     FALSE: (valence;  pointer to the node list);
             END;
```

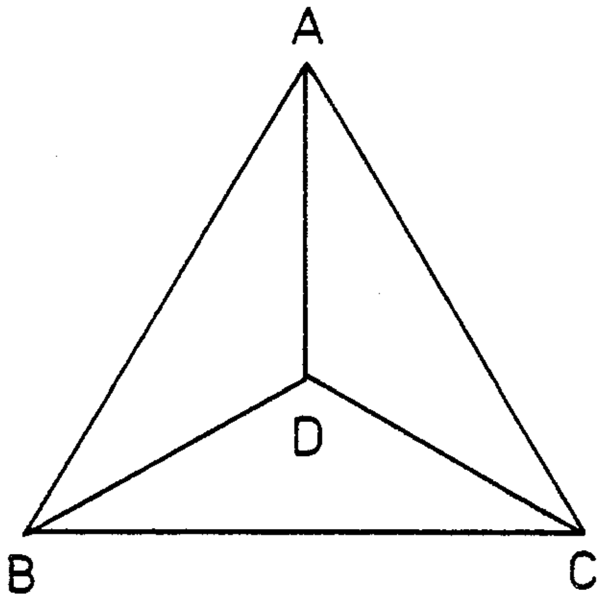


Figure 4.1
A maximal planar graph

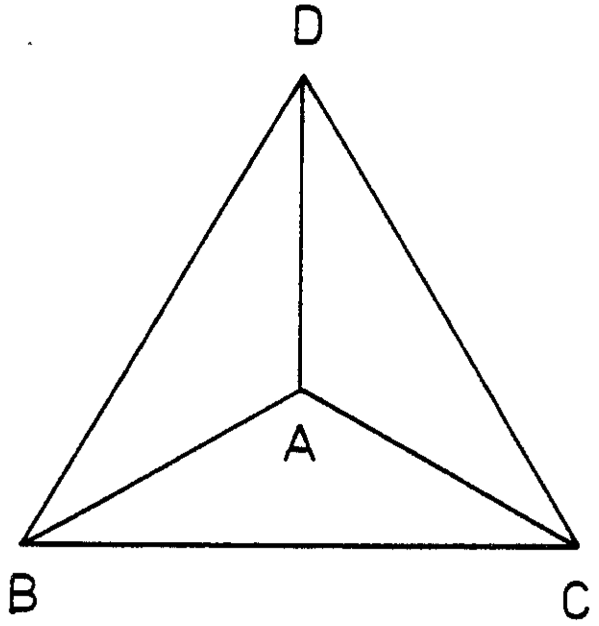


Figure 4.2
An alternative realisation of figure 4.1

```
NodeList = PACKED RECORD
               pointer to the next node in the list;
               pointer to the arc in the arc list {ArcInUse};
           END;
ArcInUse = PACKED RECORD
               the two end nodes;
               pointer to the two adjacent faces;
               pointer to the next arc;
           END;
Faces = PACKED RECORD
            the three corner nodes;
            pointer to the next faces;
        END;
```

ANodeTable is used for monitoring the availability of a node for a possible assignment. If a node is not assigned, it is classified as active, and there is a pointer to some further information regarding probable assignments and associated benefits. The calculation of the probable assignments depends upon the construction heuristic used. When a node is assigned, it is classified as nonactive. Information stored in this case consists of the number of connecting nodes, or *valence*, the pointer to the next node in the list, and the pointer to the arc list. The pointer to the arc list (*ArcInUse*) provides a convenient access to the arc information, and also ensures that the arc data fields are stored only once. As will be seen, a major part of the proposed improvement procedure involves arc-oriented operations. Data fields in the arc list (*ArcInUse*) are aimed at facilitating an efficient implementation of this procedure. The data fields consist of the two end nodes, and the pointers to the two adjacent faces, as well as to the next arc. Similarly, the data fields of a face are aimed at facilitating efficient implementations of construction heuristics.

4.3 CONSTRUCTION HEURISTICS

The strategy adopted here for the construction of a maximal planar graph is of the second kind, namely the exclusion of a planarity test. The required graph is constructed by building up from a smaller subgraph, ensuring that the subgraph is maximal and planar at all times. Thus the expensive overhead of the planarity test can be avoided.

The first stage of the construction heuristics is to build an initial planar subgraph. As three vertices are needed to generate the first pair of faces, it is possible to start with a three vertex configuration. In fact a four vertex configuration, a tetrahedron, is used in the hope that a certain initial global search for these four vertices might prove profitable. There are many strategies that can be adopted to find the initial tetrahedron. Three have been selected; the four highest weight vertices (HW), the heaviest tetrahedron (HT), and randomly generated vertices (RD). The HW strategy has a time complexity of $O(n)$, and the HT strategy has an $O(n^4)$ complexity. The complexity of the RD heuristic is not directly dependent on the size of the problem.

Insertions of the remaining nodes are carried out one by one. Each time a node is inserted into a face, by joining that node to the three corners of the face, that face is removed from the face list and three new ones are generated. By this device, the subgraph always maintains its maximal and planar properties.

Three strategies are adopted for the insertion procedure: the weight order (WO) strategy, the highest gain (HG) strategy, and the highest shadow cost (HC) strategy. For the WO strategy, all the nodes are sorted into the descending order of their weights (the weight of a node is defined as the sum of the weights of all the arcs connecting that node to the other nodes). The nodes are then inserted successively in that order into whichever face yields the highest benefit. In the HG strategy, a node is inserted into a face when its insertion maximizes the increase in the total weight of the subgraph. In the HC strategy, the node selected is the node with the largest difference between the benefits resulting from its two best insertions. The node is then inserted to the face that provides the most benefit.

Six combinations of the three starting tetrahedron strategies and the last two insertion strategies are used. 'HTHG' is used to signify the heuristic that uses the heaviest tretrahedron (HT) as the starting point, and the highest gain (HG) as the insertion strategy. In section 4.6.2, it will be shown that the weight order (WO) insertion strategy is too restrictive and will not provide useful results. It is used, however, in conjunction with the highest weight (HW) strategy as an implementation of the 'S' heuristic, suggested by Foulds & Robinson (1978). They also suggest the 'R' heuristic which is not implemented here, as the starting tetrahedron used by the heuristic is selected on the basis that it could be implemented efficiently by hand. There seems to be no sufficient justification for the restriction from the computational point of view alone.

As the insertions strategy are of $O(n^2)$ complexity, the overall complexity of the heuristics starting with the heaviest tetrahedron (HT) is $O(n^4)$. The remaining heuristics are of $O(n^2)$ complexity. It should be noted that the 'R' heuristic is of complexity $O(n^4)$.

4.4 IMPROVEMENT HEURISTICS

An improvement heuristic in the MPG problem must ensure that equations (4.1) to (4.3) are satisfied at all times. The problem is exacerbated by the fact that the graph can be realized in more than one form. Graphs in Figures 4.1 and 4.2 are identical as far as the faces, edges, nodes, and their adjacencies are concerned. In fact, they are two of the four *identical graphs* which can be realized from this very simple case. To imply that D is *inside* the triangle ABC, as seems to be the case in Figure 4.1, is not meaningful or obvious if Figure 4.2 is referred to. The technique to get around this topological uncertainty will be discussed later.

4.4.1 Arc Oriented Operations

As with the construction heuristic, the improvement heuristic can only be carried out efficiently if it does not entail planarity testing. This requirement tends to restrict the number of arcs or nodes considered for interchange during each stage. If each stage consists of removing one arc and inserting a replacement arc, it is possible to keep track of the topology of the graph without requiring excessive computing time.

An exception to the application of the pairwise exchange of arcs occurs when one or more of the nodes have minimum valence. The minimum valence is a direct consequence of the triangularity property of the face. For a graph with more than three vertices, the minimum valence is three. In the case of a node having minimum valence, other strategies (discussed later) must be applied.

4.5 THE DESIGN OF THE IMPROVEMENT HEURISTICS

In considering a pairwise arc interchange improvement procedure, the topological nature of the graph must be taken into account. When an arc is picked for consideration, it can be classified into three categories, according to the topology of the arc. Firstly **A**, one or both of the end nodes have the minimum valence. Pairwise exchange of the arcs is not applicable in such cases. Secondly **B**, no end nodes have the minimum valence and the third vertices of the adjacent faces of the arc are not connected. A possible exchange is between the arc selected and the arc joining the third vertex pair. Figure 4.3 shows a *part* of a maximal planar graph, from which nonessential details have been removed. An arc which is classified in this second category (**B**) is, for example, *CD*. The adjacent faces of the arc are *bCD* and *BCD*. *B* and *b* are the third vertices of the faces *BCD* and *bCD* with respect to the arc *CD*, and the vertices are not connected. If arc *bB* has higher weight than arc *CD*, the interchange between them would lead to a higher overall weight of the graph. The faces *bCD* and *BCD* would be replaced by the faces *bBC* and *bBD*. The adjacent faces of the arcs *bC*, *bD*, *BC* and *BD* would require updating.

Arcs in the third category **C**, are the ones in which neither of the end vertices have the minimum valence, and the third vertices of the adjacent faces are connected. An example of such an arc is *Aa* in Figure 4.3. The adjacent faces of *Aa* are *F*1 and *F*2. The third vertex pair is connected. In such a case, there are three possible options. However, all of these options are based on the assumption that the third vertex pair of the original third vertex pair *CD*, namely *Bb* is not connected. This assumption can be proved to be justified in all cases.

Start with the fact that the third vertex pair, namely *C* and *D*, of arc *Aa* are connected; so are *AC* and *AD*. *ACD* is, then, a closed circuit. One of the faces adjacent to arc *CD* must lie on one side of this circuit, and the other is on the opposite side. *B* and *b* must lie on the opposite side of the

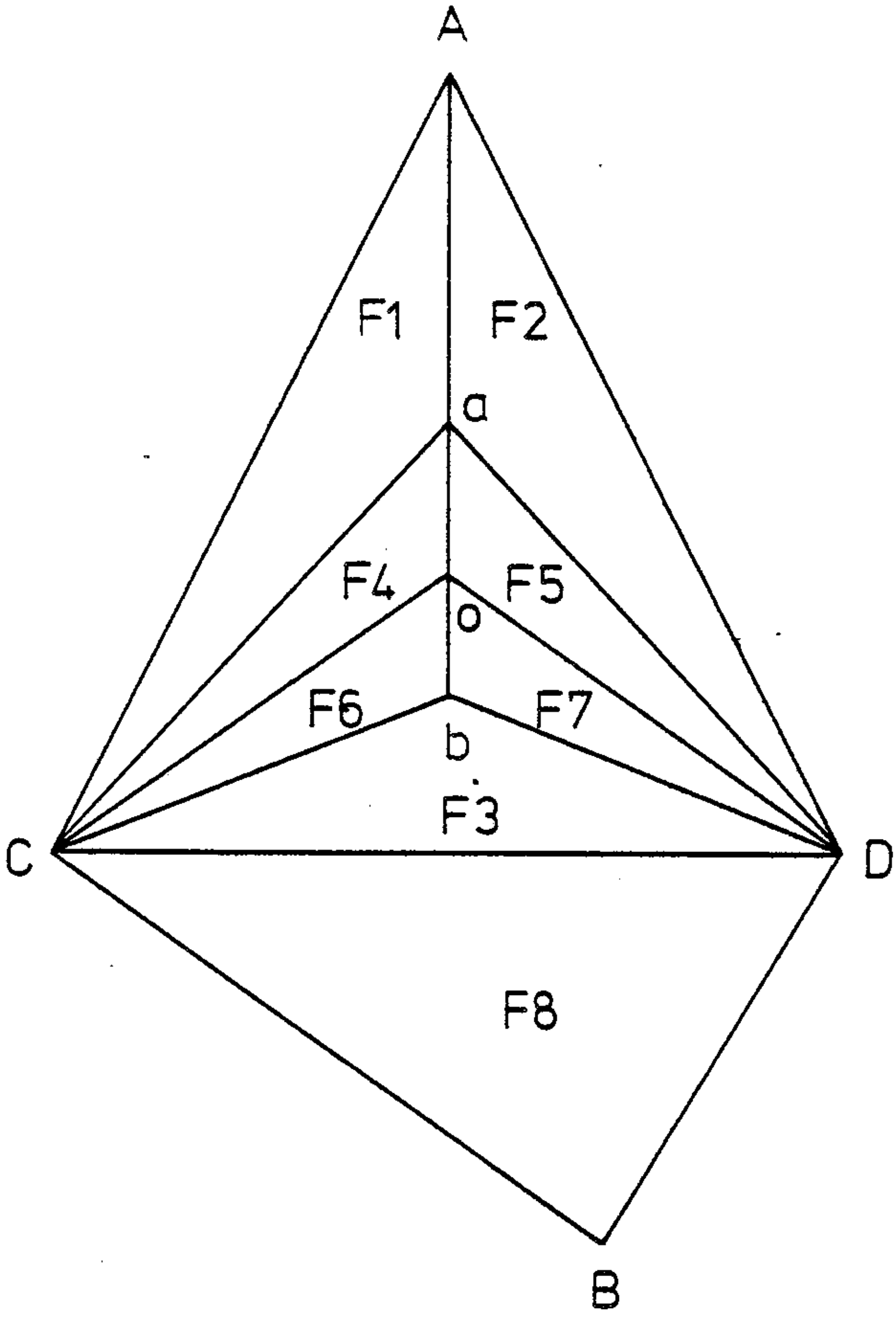


Figure 4.3
Part of a maximal planar graph

circuit *ACD* and hence cannot be connected, because the only way that the two can be joined together is to have an arc drawn across this closed circuit, thus violating the planarity constraint.

The first possible exchange in category **C** of *Aa* is with *bB*. The face changes involved in this operation are faces *bCD*, *BCD*, *aAc* and *aAD* removed; faces *bBC*, *bBD*, *aCD* and *ACD* inserted. The exchange was first suggested by Foulds & Robinson (1978). The result of the exchange is illustrated in Figure 4.4. However, to avoid unnecessary operations, this process is implemented as two exchanges of arcs in category **B**. The first exchange involves replacing *CD* by *bB*. The second involves replacing *Aa* by *CD*. As these exchanges can be carried out very quickly, the two stage implementation provides an acceptable alternative.

The second possible exchange of arc *Aa* is with *bA*. This can be visualized with reference to Figure 4.3. Firstly, *Aa* is removed, and then faces *F4-F7* are rotated 180 degrees, about *CoD*. Insert arc *Ab*. The result of this exchange is shown in Figure 4.5. The third possible exchange of *Aa*, can be illustrated with the help of Figures 4.6-4.7. Notice the changes in the positions of nodes *a*, *A*, *b* and *B* from the previous set of figures, (the reason for which will become apparent later). In this case, arc *Aa* is to be replaced by *Ab*. This can be visualized as having *Aa* removed, then faces *F4-F8* are rotated 180 degrees about arc *CD*, such that the faces *F4-F8* are *inside* the closed circuit *CbD*. Insert arc *Ab*.

In both the second and third kinds of exchange of arc *Aa* in category **C**, to be refered to as *Long Switch*, we require the topological knowledge that node *b* and faces *F3-F7* are *inside* the closed circuits *ACD* and *aCD*, as shown in Figure 4.3; or node *B* and faces *F4-F8* are *inside* the closed circuits *ACD* and *aCD*, as shown in Figure 4.6. As discussed earlier, the meaning of the word *inside* is only in reference to a certain realization of the graph, and there can be many realisations. Since not every combination of the vertices *a*, *A*, *b* and *B* will satisfy the constraints in equations (4.1) to (4.3), (*eg AB* and *ab* are not acceptable), the orientation problem must be overcome or circumvented.

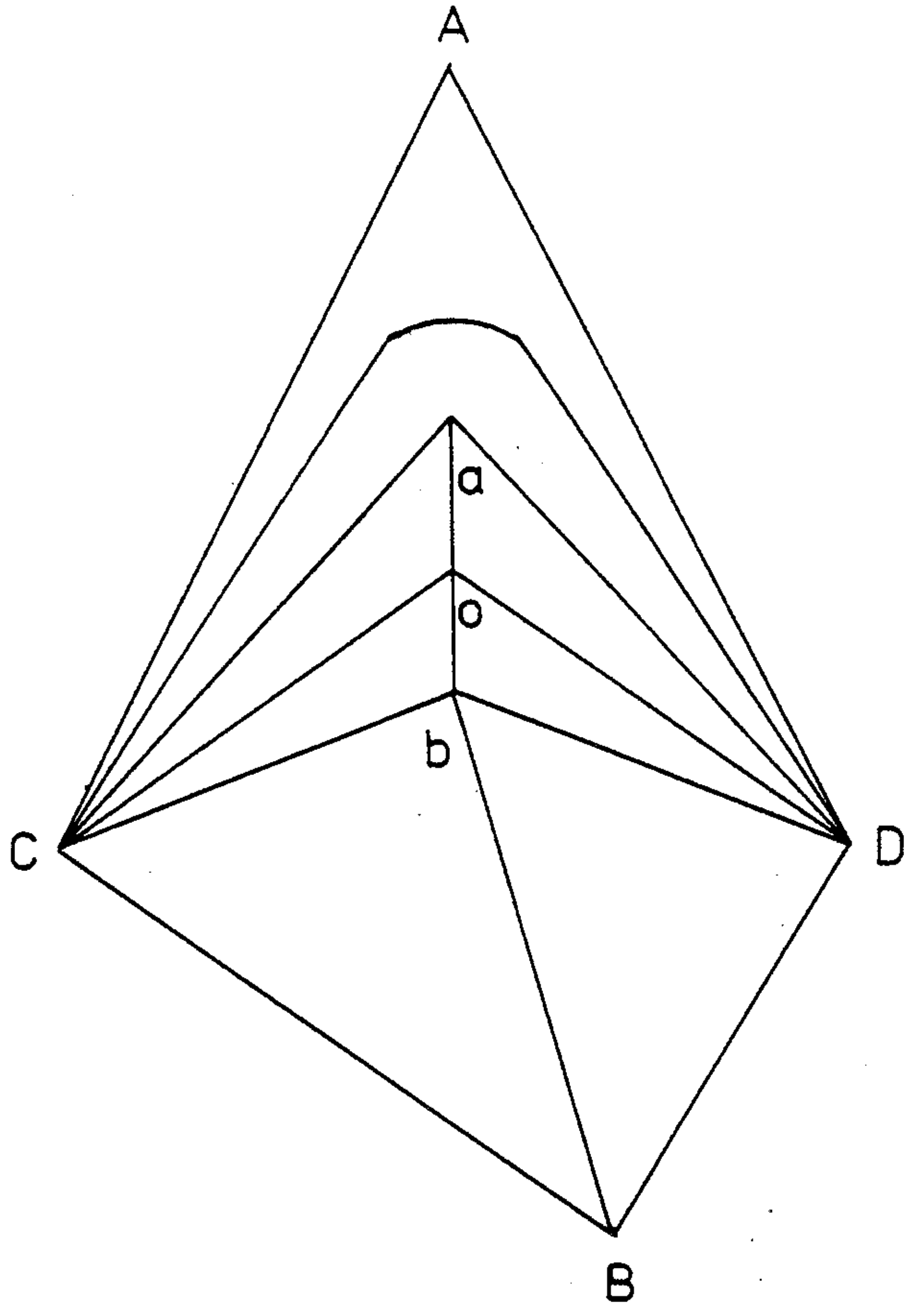


Figure 4.4
Figure 4.3 after a C arc exchange

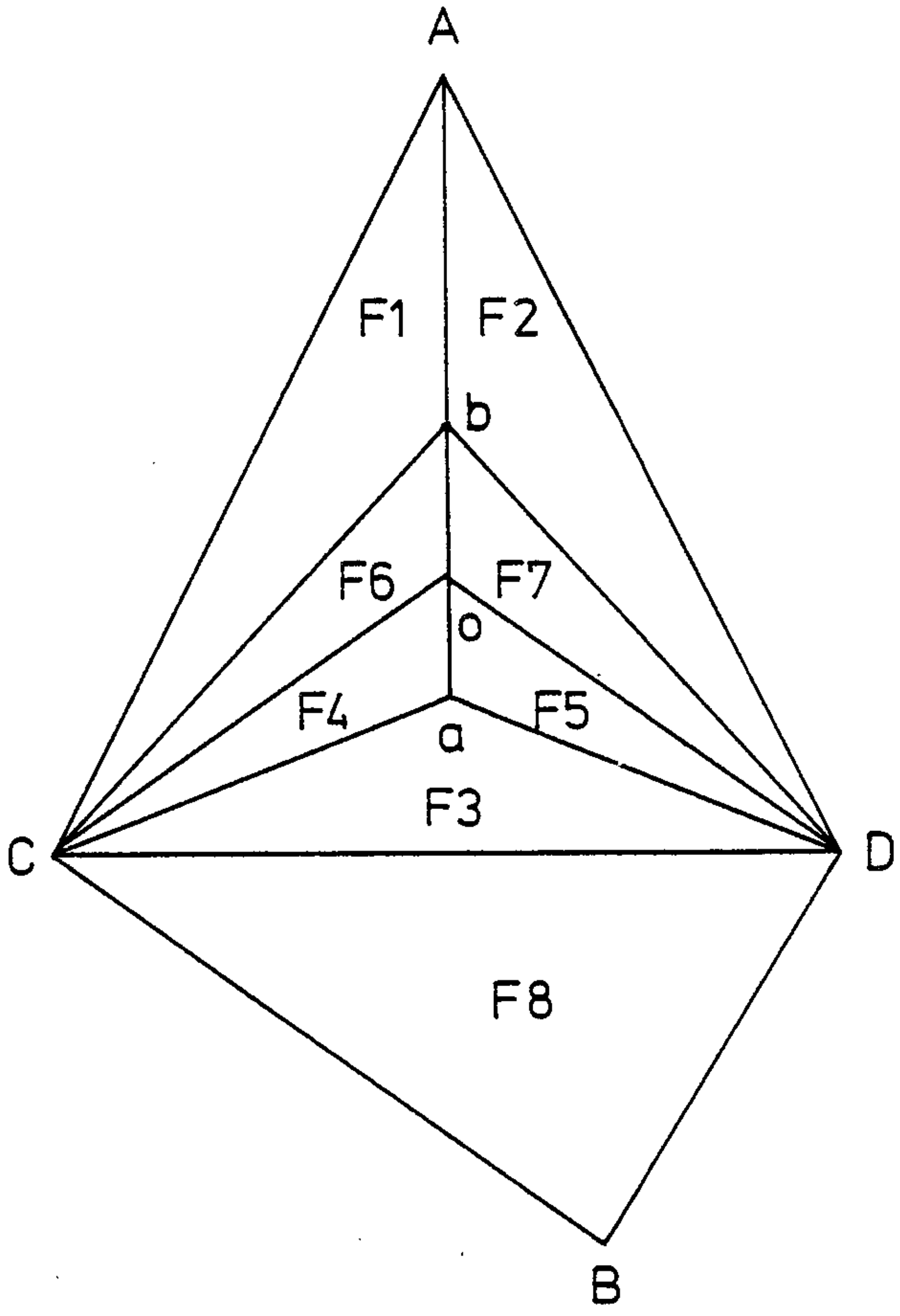


Figure 4.5
Figure 4.3 after another C arc exchange

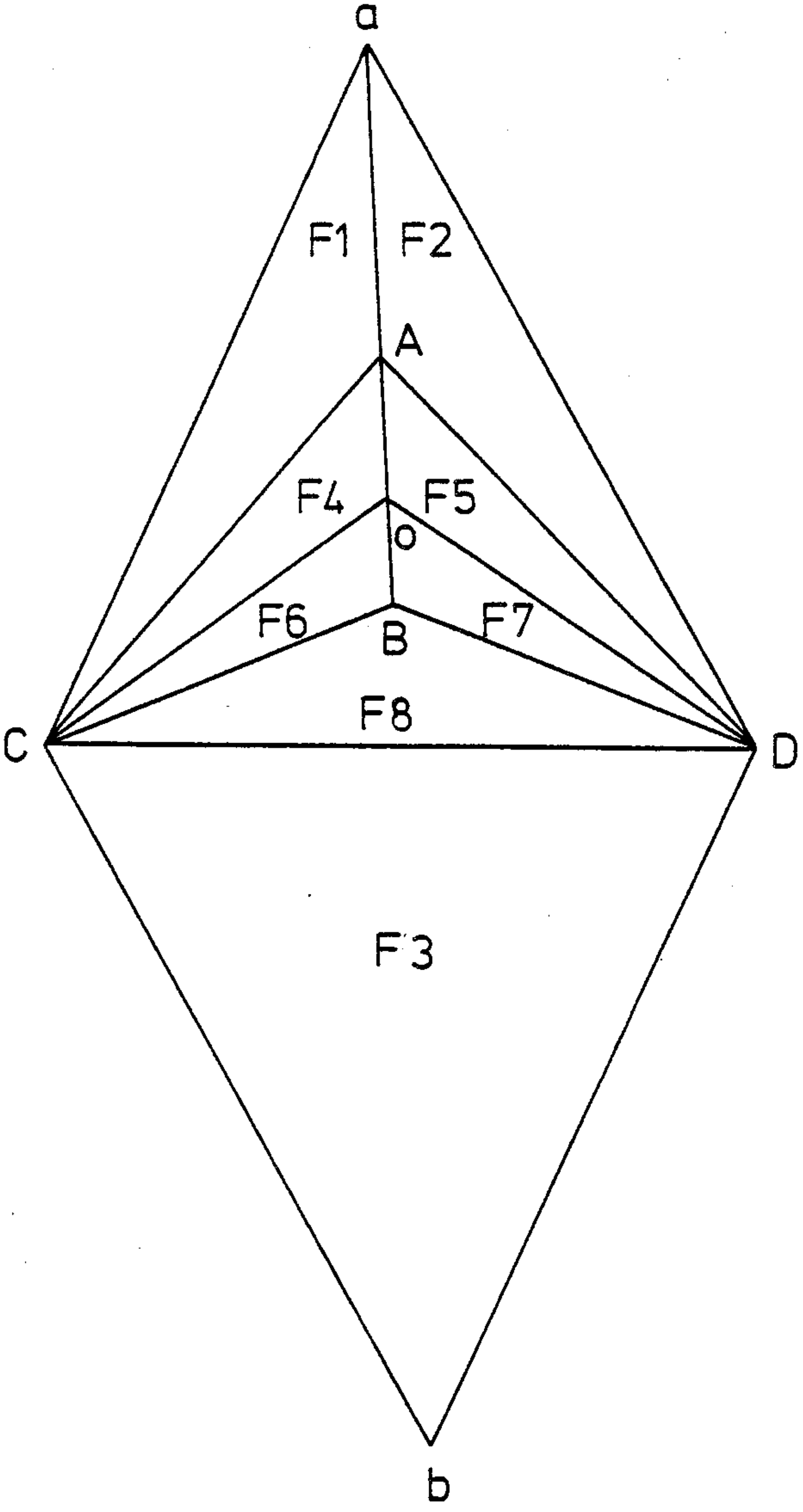


Figure 4.6
An alternative labelling scheme for figure 4.3

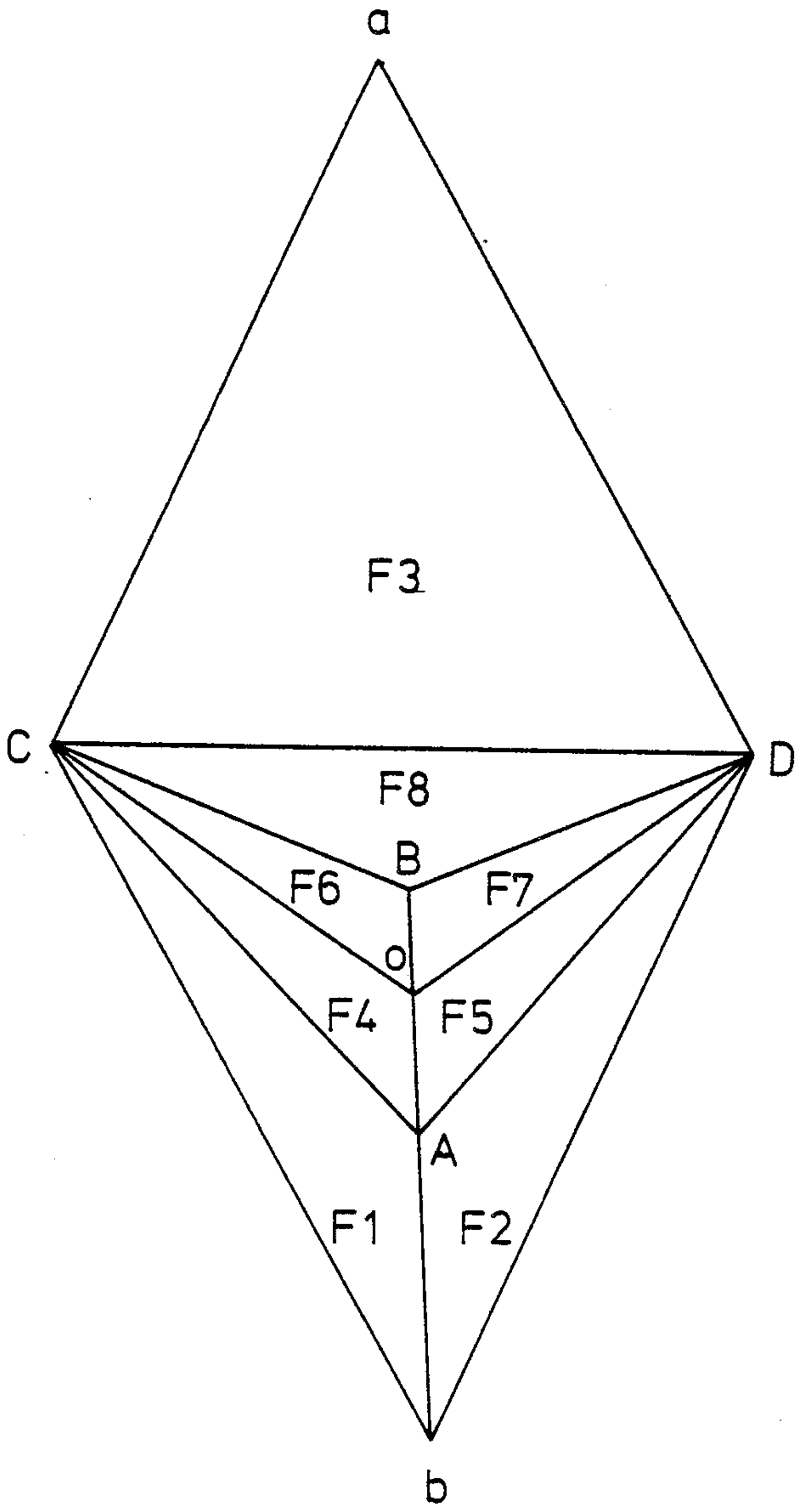


Figure 4.7
Figure 4.6 after a C arc exchange

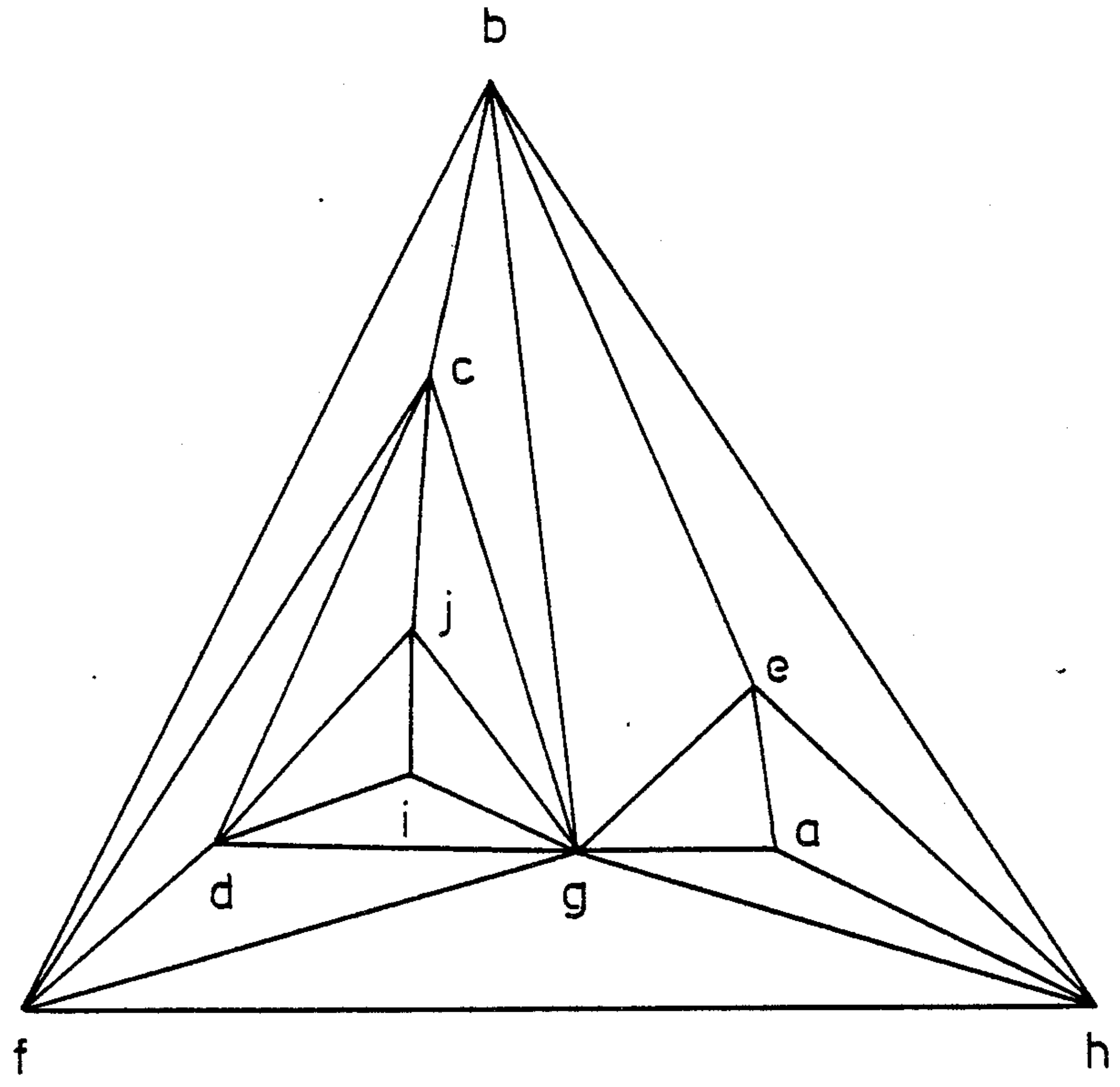


Figure 4.8
A solution to Fould & Robinson's 10 vertex problem

This orientation problem can be avoided by adopting the labeling and transformation schemes, suggested in the following Long Switch algorithm:

```
{Given an arc which is in category C}
{Labelling phase}
Label the third vertex pair of the given arc as C and D;
Pick the third node from one of the faces adjacent to CD,
    label this node b;
Label the third node from the other adjacent face of CD as B;
Using C {or D} as the pivoting point and bC {or bD} as datum;
REPEAT
    Locate the next node adjacent to C {or D} by moving in
        the opposite direction to the one towards CB {or DB};
UNTIL the located node is one end of the given arc;
Label that found node a, and the other end node as A;
Label faces aAC, aAD and bCD as F1, F2 and F3 respectively;
{End of labelling phase}

{Transformation Phase}
Remove arc Aa and associated information;
Insert arc Ab and associated information;
Replace vertices in face F1 by A, b and C;
Replace vertices in face F2 by A, b and D;
Replace vertices in face F3 by a, C and D;
Replace pointer to face F1 of arc aC by pointer to F3;
Replace pointer to face F2 of arc aD by pointer to F3;
Replace pointer to face F3 of arc bC by pointer to F1;
Replace pointer to face F3 of arc bD by pointer to F2;
{End of the transformation phase}
```

To illustrate the use of the Long Switch algorithm, consider the graph in Figure 4.3. In this case, the arc *Aa* is chosen for examination. At this stage it is neither possible nor neccesary to state which end of the arc is node *A* and which is node *a*. The third vertex pair of arc *Aa* are nodes *C* and *D*, which are connected. The third vertex pair of arc *CD* are *B* and *b*. Assume that the node selected is *inside* the circuits *ACD* and *aCD*, and hence labelled *b* as shown. The other vertex of the pair is then labelled *B*. Using *bC* as the reference line and *C* as the pivoting point, locate the next node, node *o*, by moving in the opposite direction to the one towards *BC*. Repeat the process again, this time the node found is one end of the given arc. The node is then labelled *a*. The other end of the arc is labelled *A*. The exchange is carried out, if so desired, by the transformation suggested in the algorithm. The result can easily be verified by inspection of the graph in Figure 4.5.

Figure 4.6 represents the case when the third node of the face adjacent to arc *CD* is not *inside* the faces *ACD* or *aCD*. It can be seen that by adopting the same labelling scheme, the transformation phase will also provide the correct outcome. Figure 4.7 can be used to verify the result. Note that faces *F4*-*F8* and some of the arcs are not directly involved with the transformation process. They are included in order to indicate the orientations of the various components of the graph before and after the transformation.

It should be emphasised that arc exchanges involving the two types of the Long Switch are not mutually exclusive; it is possible to consider exchange of either type. Hence, for an arc in category **C**, there are three possible candidates for exchange, and there is only one candidate for the arc in category **B**.

The complete arc exchange procedure can be summarised as follows:

```
IF the third vertex pair of the selected arc not connected
   THEN
      {category B}
      IF type B switch beneficial
         THEN exchange arcs of type B;
      {ENDIF beneficial}
   ELSE
      {category C}
      select appropriate swithcing type;
      CASE
         First type:  exchange category B twice;
         Second and third types:  LongSwitch algorithm;
      END CASE;
   {ENDIF not connected}
{END of the algorithm}
```

This procedure can be more efficiently implemented than the procedure suggested by Foulds & Robinson, as well as being more comprehensive: the Foulds & Robinson procedure does not include the Long Switch type of exchanges. The first type of the category **C** exchange is also inefficiently carried out, involving the search for cliques of size four.

In the case mentioned earlier where pairwise arc exchange is not possible due to the triangularity constraint, the improvement procedure is a *node* oriented operation. This is carried out by considering the possible benefit of moving a node of minimum valence and its associated arcs from their present location to another face. This process is parallel to the one carried out during the construction phase. Implementation of this procedure is summarised as follows:

```
WHILE the NodeTable is not exhausted DO
    BEGIN
        IF valence of the node = 3
        THEN
            BEGIN
                find the best new location if removed;
                IF beneficial THEN switch to new location;
            ENDIF;
        move to the next node in the table;
    ENDWHILE;
```

4.6 IMPLEMENTATION AND COMPARISONS OF THE HEURISTICS

All the heuristics and supporting procedures are written in Pascal. It was decided that, in order to overcome the usual criticisms levelled against tests of heuristics of comparable complexity, the heuristics would be loaded together and executed immediately one after the other, hence reducing the influence of the operating conditions on the final results. The entire program consists of approximately 1500 lines of source code. The compiled code requires less than 8*K* words for 30 vertex problems and less than 12*K* words for 100 vertex problems when run on a *CDC Cyber 174* using the *Pascal 6000* compiler with runtime checking suppressed. The compactness of the code suggests many possible elaborations. Firstly, it can be made to run faster either by having more data fields in the packed format, or by using the data in the normal mode, one word per field, in place of the packed version currently implemented, without running into storage problems for relatively large classes of problems. Secondly, using the present storage scheme, the program can handle problems with 300 or 400 vertices without any practical difficulty. It is estimated that the 300 vertex problem executed by an $O(n^2)$ heuristic would require approximately 200 *Cyber 174* seconds. Finally, if so desired, further data compaction would allow problems of much larger size, perhaps 800 vertices, to be solved at the expense of a higher runtime overhead. It is interesting to note that the program produces a solution to the Foulds & Robinson 10 vertex problem with a total weight of 1103 (Figure 4.8). This result is higher than the optimum of 1096 suggested in their paper.

4.6.1 Design of the experiment

The main aims of the experiment are to assess the relative merits, the comparative speeds of execution and the effects of the problem size on various strategies. To achieve these objectives, eight classes of problems, ranging from 10 to 100 vertices, are used. In each class, five random symmetrical and completed graphs are generated. The arc costs are limited to the range of one to one hundred. All the forty test problems are solved by all the $O(n^2)$ heuristics. As the expected runtimes of the $O(n^4)$ heuristics for the larger problems become excessive with respect to the resources available, it was decided that only 25 smaller problems were to be tested on this class of

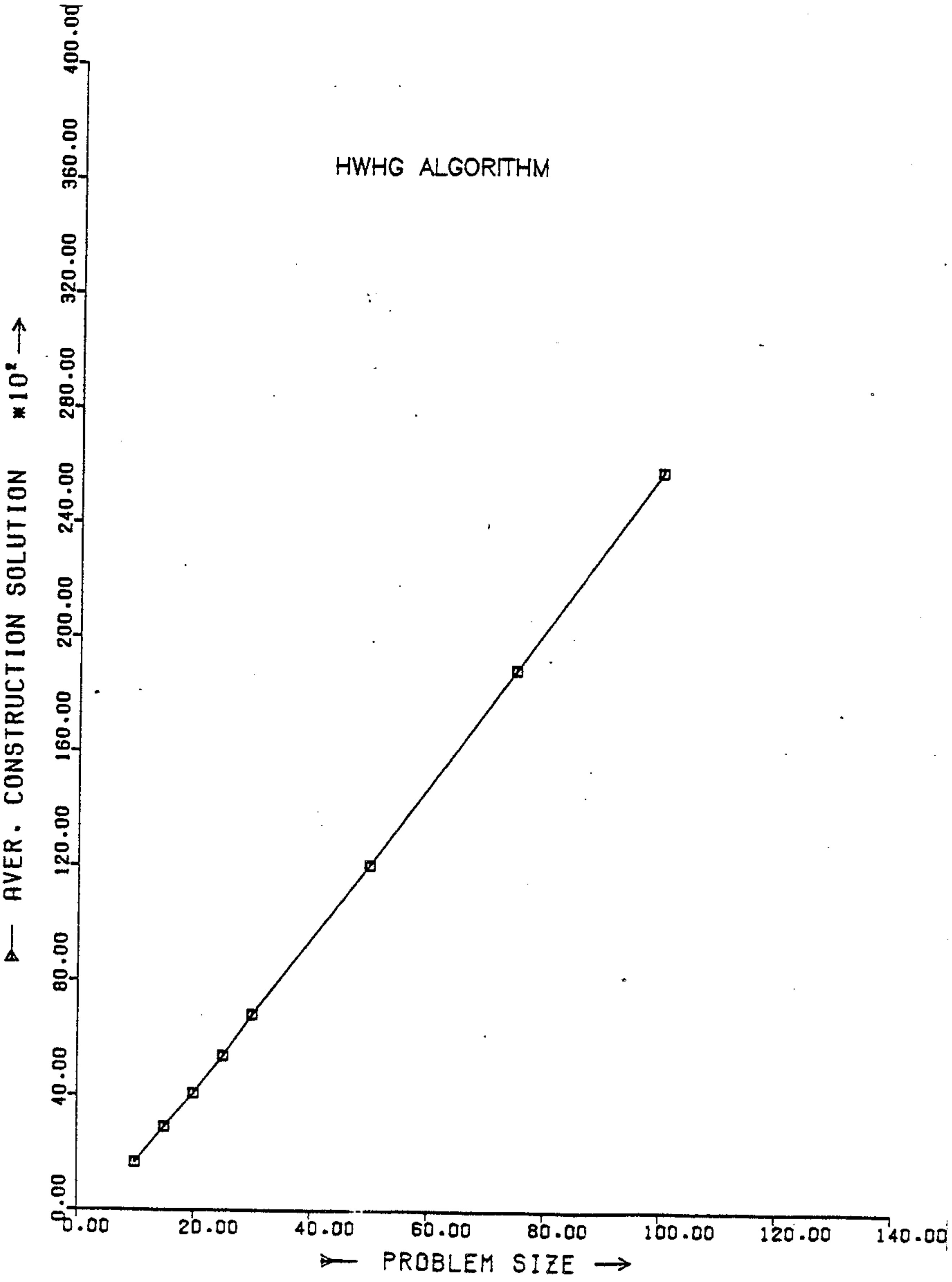


Figure 4.9
Average construction solutions of HWHG heuristic for the MPG

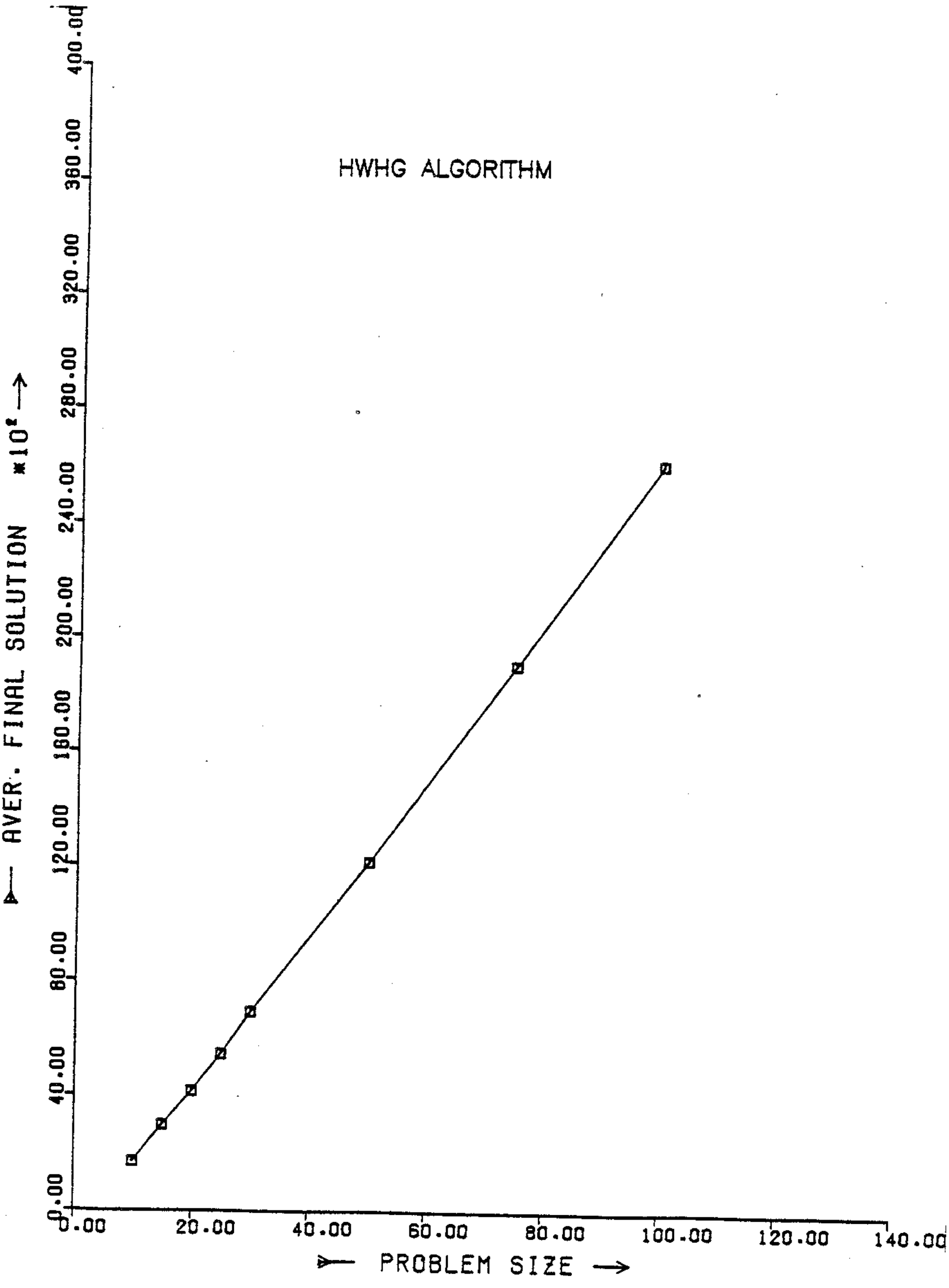


Figure 4.10
Average final solutions of HWHG heuristic for the MPG

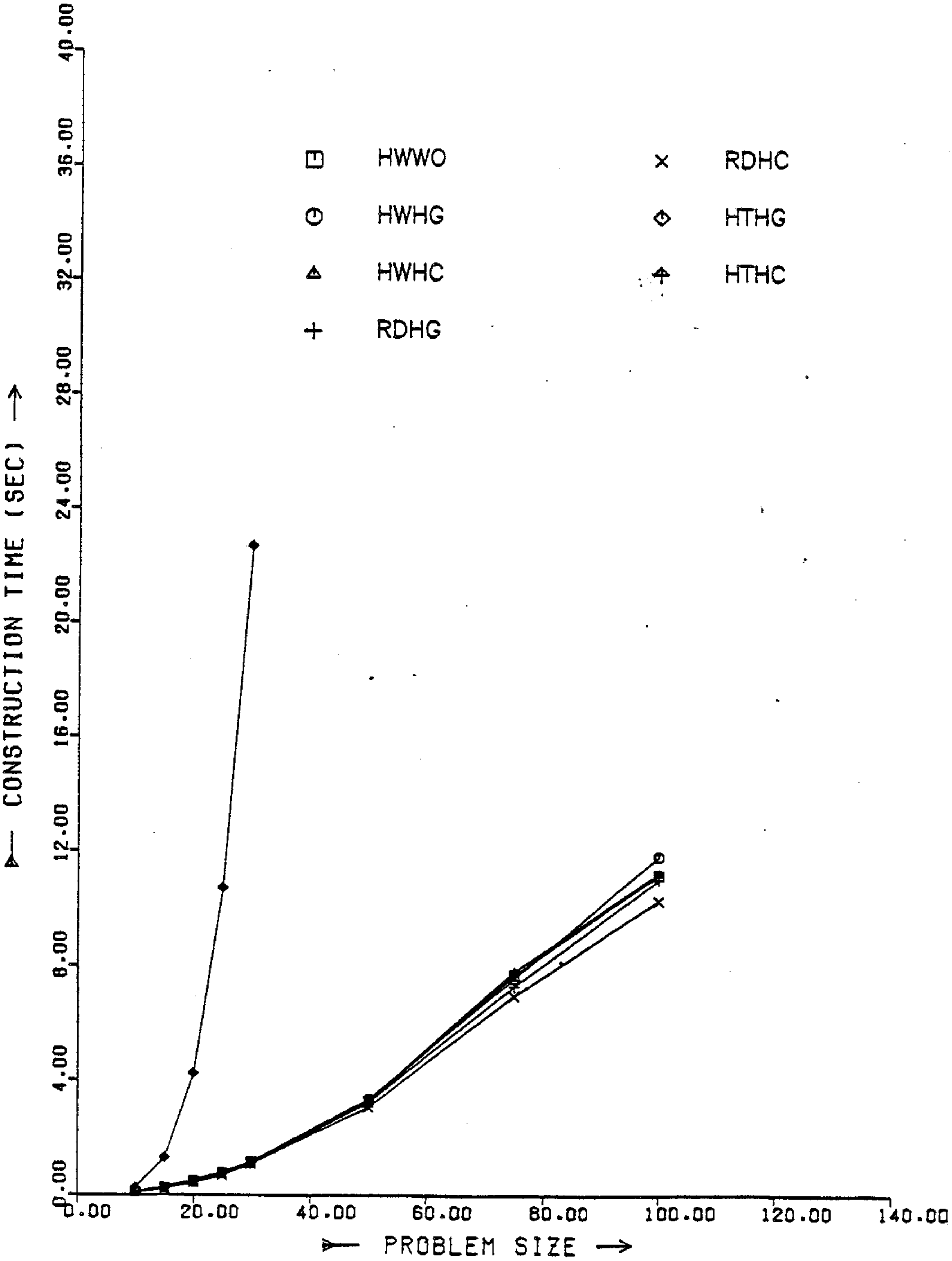


Figure 4.11
Average construction times of heuristics for the MPG

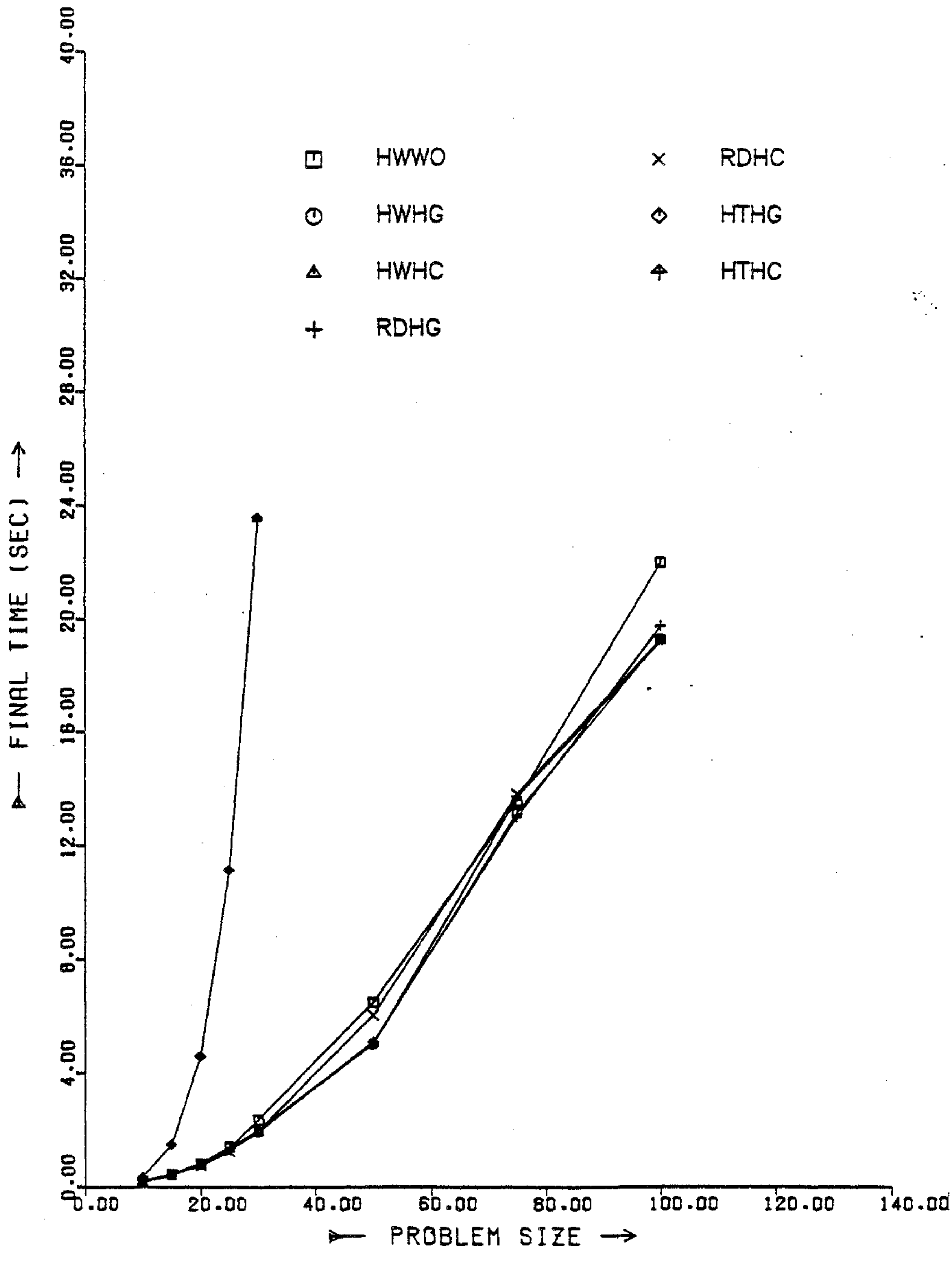


Figure 4.12
Average final runtimes of heuristics for the MPG

PROBLEM		HEURISTICS								
SIZE	NO.	HWWO	HWHG	HWHC	RDHG	RDHC	HTHG	HTHC	MAX	MIN
	1	1585	1631	1620	1493	1551	1617	1578	1631	1493
	2	1647	1621	1647	1595	1569	1621	1647	1647	1569
10	3	1566	1648	1648	1643	1652	1694	1660	1694	1566
	4	1747	1730	1726	1691	1570	1749	1677	1749	1570
	5	1708	1718	1685	1588	1503	1627	1700	1718	1503
	AVER.	1651	1670	1665	1602	1569	1662	1652		
	6	2899	2927	2847	2797	2760	2927	2834	2927	2760
	7	2905	2909	2924	2792	2868	2918	2848	2924	2792
15	8	2850	2864	2906	2834	2785	2919	2914	2919	2785
	9	2967	3076	2996	2762	2819	3076	2967	3076	2762
	10	2778	2792	2846	2788	2614	2867	2861	2867	2614
	AVER.	2880	2914	2904	2795	2769	2941	2885		
	11	3923	3996	3943	3919	3891	4015	3935	4015	3891
	12	4053	4097	4018	4113	3843	4143	3952	4143	3843
20	13	4003	4081	4063	4007	4091	4092	3993	4092	3993
	14	4004	4075	4176	4043	3860	4047	4060	4176	3860
	15	4057	4167	4090	3941	3884	4062	4133	4167	3884
	AVER.	4008	4083	4058	4005	3914	4072	4015		
	16	5305	5409	5357	5132	4922	5191	5362	5409	4922
	17	5207	5274	5222	5395	5041	5447	5182	5447	5041
25	18	5332	5345	5365	5303	5083	5462	5276	5462	5083
	19	5434	5436	5549	5332	5495	5557	5552	5557	5332
	20	5180	5474	5451	5365	5129	5521	5451	5521	5129
	AVER.	5292	5388	5389	5305	5134	5436	5365		
	21	6689	6855	6681	6667	6457	6878	6691	6878	6457
	22	6753	6892	6639	6697	6353	6889	6602	6892	6353
30	23	6551	6779	6634	6760	6484	6763	6663	6779	6484
	24	6523	6833	6561	6590	6601	6802	6648	6833	6523
	25	6660	6692	6582	6528	6378	6757	6739	6757	6378
	AVER.	6635	6810	6619	6648	6455	6818	6669		
	26	11663	12074	11695	12113	11689			12113	11663
	27	11656	12043	11822	11825	11861			12043	11656
50	28	11856	12075	12036	12034	11534			12075	11534
	29	11659	11826	11674	11975	11619			11975	11619
	30	11781	12225	11788	12042	11787			12225	11781
	AVER.	11723	12049	11803	11998	11698				
	31	18388	18794	18270	18601	18328			18794	18270
	32	18388	19107	18540	18950	18472			19107	18388
75	33	18591	18884	18517	18842	18328			18884	18328
	34	18448	18801	18382	18658	18207			18801	18207
	35	18495	18873	18596	18908	18726			18908	18495
	AVER.	18462	18892	18461	18792	18412				
	36	25171	25526	25126	25600	25186			25600	25126
	37	25453	26222	25576	25946	25436			26222	25436
100	38	25296	25872	25175	25844	24985			25872	24985
	39	25053	25820	25372	25674	25055			25820	25053
	40	25066	25754	25382	25736	25334			25754	25066
	AVER.	25208	25839	25326	25760	25199				

Table 4.1
Construction Solutions of MPG Heuristics

PROBLEM		HEURISTICS								
SIZE	NO.	HWWO	HWHG	HWHC	RDHG	RDHC	HTHG	HTHC	MAX	MIN
	1	1627	1631	1627	1627	1617	1617	1619	1631	1617
	2	1679	1679	1679	1679	1626	1679	1679	1679	1626
10	3	1710	1712	1712	1705	1717	1715	1660	1717	1660
	4	1766	1749	1737	1717	1724	1749	1754	1766	1717
	5	1719	1720	1719	1637	1593	1647	1700	1720	1593
	AVER.	1700	1698	1695	1673	1655	1681	1682		
	6	2960	2960	2943	2888	2801	2960	2869	2960	2801
	7	2975	2975	2939	2890	2910	2925	2927	2975	2890
15	8	2977	2951	2943	2887	2933	2945	2935	2977	2887
	9	2991	3082	3052	3029	2952	3082	3012	3082	2952
	10	2870	2934	2956	2878	2815	2915	2930	2956	2815
	AVER.	2955	2980	2967	2914	2882	2965	2935		
	11	4037	4056	4016	3989	3990	4019	4002	4056	3989
	12	4208	4161	4114	4192	4027	4167	4136	4208	4027
20	13	4028	4159	4113	4213	4102	4193	4046	4213	4028
	14	4061	4140	4194	4094	4106	4107	4157	4194	4061
	15	4203	4282	4259	4012	3962	4156	4174	4282	3962
	AVER.	4107	4160	4139	4100	4037	4128	4103		
	16	5427	5424	5430	5390	5203	5270	5416	5430	5203
	17	5389	5361	5303	5395	5151	5466	5248	5466	5151
25	18	5375	5395	5418	5477	5345	5473	5354	5477	5345
	19	5484	5504	5609	5517	5509	5589	5552	5609	5484
	20	5427	5568	5481	5472	5410	5544	5528	5568	5410
	AVER.	5420	5450	5448	5450	5324	5468	5420		
	21	6936	6928	6980	6847	6777	7017	6707	7017	6707
	22	6910	6950	6822	6833	6731	6975	6880	6975	6731
30	23	6774	6948	6818	6834	6609	6876	6793	6948	6609
	24	6714	7010	6795	6876	6733	6838	6806	7010	6714
	25	6833	6807	6791	6834	6665	6838	6823	6838	6665
	AVER.	6833	6929	6841	6845	6703	6909	6802		
	26	11991	12274	11982	12291	12085			12291	11982
	27	12078	12218	11984	12082	12104			12218	11984
50	28	12325	12191	12224	12315	11904			12325	11904
	29	11923	11971	11929	12220	12070			12220	11923
	30	12295	12245	12035	12158	12037			12295	12035
	AVER.	12122	12180	12031	12213	12040				
	31	18839	18978	18688	19000	18549			19000	18549
	32	18852	19322	18868	19122	18735			19322	18735
75	33	18868	19073	18875	19195	18731			19195	18731
	34	18746	18972	18707	18789	18659			18972	18659
	35	18846	19013	19054	19149	19070			19149	18846
	AVER.	18830	19072	18838	19051	18749				
	36	25531	25842	25566	26082	25661			26082	25531
	37	25793	26470	26062	26281	25728			26470	25728
100	38	25804	26141	25803	25931	25545			26141	25545
	39	25804	26002	25550	25961	25605			26002	25550
	40	25738	26184	25990	25997	25895			26184	25738
	AVER.	25734	26128	25794	26050	25687				

Table 4.2
Final Solutions of MPG Heuristics

PROBLEM		HEURISTICS								
SIZE	NO.	HWWO	HWHG	HWHC	RDHG	RDHC	HTHG	HTHC	MAX	MIN
	1	94	101	106	95	86	258	255	258	86
	2	98	99	100	83	96	249	255	255	83
10	3	99	94	95	74	73	257	249	257	73
	4	98	90	103	91	87	250	257	257	87
	5	90	104	97	78	91	250	260	260	78
	AVER.	96	98	100	84	87	253	255		
	6	239	235	245	253	231	1291	1314	1314	231
	7	239	251	242	228	230	1309	1284	1309	228
15	8	231	242	249	212	210	1301	1312	1312	210
	9	255	259	285	235	221	1316	1285	1316	221
	10	242	256	272	252	213	1294	1307	1307	213
	AVER.	241	249	259	236	221	1302	1302		
	11	506	446	445	425	417	4276	4230	4276	417
	12	452	522	462	456	417	4285	4228	4285	417
20	13	462	489	484	417	423	4227	4253	4253	417
	14	490	498	531	453	393	4247	4248	4248	393
	15	448	472	551	422	437	4280	4242	4280	422
	AVER.	471	485	495	435	417	4263	4240		
	16	760	789	798	735	688	10685	10754	10754	688
	17	766	824	819	774	618	10742	10637	10742	618
25	18	695	761	749	747	698	10722	10663	10722	695
	19	858	716	798	726	700	10765	10676	10765	700
	20	707	821	791	653	695	10721	10785	10785	653
	AVER.	757	782	791	727	679	10727	10703		
	21	1085	1103	1088	1029	1170	22563	22622	22622	1029
	22	1072	1261	1168	1165	1037	22783	22799	22799	1037
30	23	1184	1142	1190	1086	1034	22702	22714	22714	1034
	24	1253	1119	1165	1145	1170	22636	22600	22636	1119
	25	1147	1122	1111	1045	1054	22758	22625	22758	1045
	AVER.	1148	1149	1144	1094	1093	22688	22672		
	26	3257	3460	3264	3529	3011			3529	3011
	27	3263	3145	3303	2954	3092			3303	2954
50	28	3169	3380	3091	3354	3012			3380	3012
	29	3357	3312	3304	3077	3175			3357	3077
	30	3160	3384	3328	3246	3038			3384	3038
	AVER.	3241	3336	3258	3232	3066				
	31	7624	7245	7692	7251	6852			7692	6852
	32	7847	7452	7672	7151	7248			7847	7151
75	33	7798	7394	7541	7299	6693			7798	6693
	34	7579	7790	8372	7263	7401			8372	7263
	35	7568	7892	7743	7470	6574			7892	6574
	AVER.	7683	7555	7804	7287	6954				
	36	10695	11251	10989	11195	10355			11251	10355
	37	11781	12368	11295	10747	10317			12368	10317
100	38	11291	12306	11349	10635	10245			12306	10245
	39	10864	12057	11496	11253	10422			12057	10422
	40	11221	11116	11018	11191	10065			11221	10065
	AVER.	11170	11820	11229	11004	10281				

Table 4.3

Construction Times (mil.sec) of MPG Heuristics

PROBLEM		HEURISTICS								
SIZE	NO.	HWWO	HWHG	HWHC	RDHG	RDHC	HTHG	HTHC	MAX	MIN
	1	161	141	168	191	186	299	406	406	141
	2	242	253	244	194	164	402	399	402	164
10	3	190	146	148	162	172	362	286	362	146
	4	166	158	219	153	220	290	329	329	153
	5	151	167	178	148	206	346	298	346	148
	AVER.	182	173	191	170	190	340	344		
	6	476	461	599	450	341	1516	1435	1516	341
	7	452	396	376	494	410	1434	1569	1569	376
15	8	496	444	392	357	431	1480	1440	1480	357
	9	463	383	411	447	544	1442	1459	1459	383
	10	344	456	430	365	490	1511	1560	1560	344
	AVER.	446	428	442	423	443	1477	1493		
	11	808	809	693	706	714	4479	4537	4537	693
	12	693	928	859	743	852	4579	4705	4705	693
20	13	650	793	671	736	548	4693	4693	4693	548
	14	912	775	715	821	779	4585	4559	4585	715
	15	946	785	1155	855	748	4603	4553	4603	748
	AVER.	801	818	818	772	728	4588	4609		
	16	1606	1104	1928	2105	1662	10964	11307	11307	1104
	17	1517	1326	1110	1006	1067	11413	11136	11413	1006
25	18	973	1635	1253	1278	1089	11220	11083	11220	973
	19	1649	1611	1053	1300	1121	11004	10907	11004	1053
	20	1224	1444	1183	1072	1276	11116	11274	11274	1072
	AVER.	1394	1424	1305	1305	1243	11143	11141		
	21	2481	1771	1967	2222	1823	23666	23199	23666	1771
	22	1632	1978	2216	1739	2491	23416	24040	24040	1632
30	23	2652	2259	2084	1835	1572	23369	23303	23369	1572
	24	2121	2043	1861	2082	1809	23458	23374	23458	1809
	25	2926	1691	1975	1701	2102	23990	23436	23990	1691
	AVER.	2362	1948	2021	1916	1959	23580	23470		
	26	5915	5490	4761	4989	7532			7532	4761
	27	6643	4623	5466	4696	6900			6900	4623
50	28	6739	4882	4608	5569	4925			6739	4608
	29	5562	5122	4862	5454	5335			5562	4862
	30	7633	5119	5489	4879	5556			7633	4879
	AVER.	6498	5047	5037	5117	6050				
	31	14648	12657	13317	13675	11524			14648	11524
	32	14299	12386	13893	12205	14455			14455	12205
75	33	15354	12322	13185	14611	14713			15354	12322
	34	10797	15426	14927	12402	16339			16339	10797
	35	12830	13089	13184	12151	12098			13184	12098
	AVER.	13586	13176	13701	13009	13826				
	36	17599	20632	20237	22925	17564			22925	17564
	37	27041	20435	19494	20256	19436			27041	19436
100	38	22411	17371	19811	16327	18143			22411	16327
	39	19167	17413	17097	18584	16544			19167	16544
	40	23798	20578	19432	20721	24715			24715	19432
	AVER.	22003	19286	19214	19763	19280				

Table 4.4
Total Runtimes (mil.sec) of MPG Heuristics

heuristics.

4.6.2 Analysis of the Experimental Results

The task of analysing the empirical results of various heuristics raises an important theoretical issue, namely the nature of the scale of measurement of the results. One school of thought treats the results as metric data, hence the use of elaborate statistical techniques are justified (Golden & Stewart, 1981; Golden & Assad, 1982; King & Spachis, 1980; Spachis, 1978). This approach is acceptable only when the problems tested are of similar complexities, *ie* roughly of the same sizes. When the problem size varies greatly, the metric property of the results is required to be justified explicitly. This is due to a well known general phenomenon of combinatorial problems: that it is far more difficult to get within a certain range of an optimun solution in a larger problem than it is for a smaller one. The larger the difference in size, the greater the difference in computation efforts; to obtain a solution within one percent of the optimal solution for a 30 vertex problem does not imply the same effectiveness as obtaining a solution within the same percentage range for a 100 vertex problem.

The second school of thought, and it is the one adopted here, is that the data are only ordinal and performance analyses should rely on nonparametric tests (Parker, 1976; Abdel Barr, 1978). The average values of the results in the Tables 4.1-4.4 are used only as *rough guides*, and play no part in the analysis of performance as such. The sign test and the run test are the two main procedures used.

The performances of various heuristics on the test problems are tabulated in the Tables 4.1-4.4.

The results of the sign tests for the solutions of the construction procedures are summarised in Table 4.5. The first figure of each pair is the number of times the row-label heuristic provided higher (in this case better) solutions than the column-label heuristic. The second figure is the number of times the reverse occurred. The number of ties can be deduced from the difference of the numbers of test problems and the sum of the two figures in the table. If the HWWO heuristic is omitted from the table, it would represent a two level factorial design, and hence the effect of a class of strategies (level) can be studied by comparing the results of the heuristics while keeping the other level constant.

HEURISTICS						
HWHG	HWHC	RDHG	RDHC	HTHG	HTHC	
2, 38	12, 27	16, 24	28, 12	3, 22	10, 13	HWWO
	32, 7	34, 6	37, 3	8, 14	19, 6	HWHG
		21, 19	33, 7	7, 18	12, 11	HWHC
			32, 8	0, 25	7, 18	RDHG
				0, 25	2, 23	RDHC
					20, 5	HTHG

Table 4.5
Construction Cost Sign Tests

The effect of the initial tetrahedron strategies is considered by comparing the results of the HWHG, RDHG, and HTHG heuristics, and then comparing the results of the HWHC, RDHC and HTHC heuristics. There are some indications that the heaviest tetrehedron (HT) strategy produces better solutions at the end of the construction phase than the highest weight order (HW) strategy although the result is not statistically significant. Both strategies perform better (statistically significant at 5% or less) than the random strategy, which is to be expected. Similar analysis for the insertion strategies shows that the weight order (WO) insertion is significantly poorer (at 5% or less level) than the other two insertion methods, thus justifing the decision to test this strategy in a less comprehensive manner. The highest gain (HG) strategy performs statistically better (at 5% or less level) than the highest cost (HC) strategy. This is an unexpected outcome, as it is usually the case that the highest cost strategy gives better results, as in the case of the transportation problem or the travelling salesman problem. The run tests on the results in Table 4.6 show two significant results; between RDHG and HWWO test (less than 4% level) and between RDHG and HWHC test (less than 0.1% level). The RDHG heuristic shows significantly poorer results for the smaller problems, and significantly better results for the larger problems than the results produced by the HWWO and HWHC heuristics. It should be noted that the straight-forward sign tests on both sets of results are not statistically significant. A possible explanation is that the RD strategy provides a poorer starting condition than the one produced by the HW strategy. However, if the HG insertion strategy is allowed to take its full effect, by using it in larger problems, the initial disadvantage will in most cases be overcome. This interpretation is consistent with the earlier conclusion regarding the performance of various strategies during the construction phase.

HEURISTICS						
HWHG	HWHC	RDHG	RDHC	HTHG	HTHC	
9, 28	17, 20	14, 24	29, 11	9, 14	16, 8	HWWO
	30, 8	26, 13	36, 4	12, 9	20, 4	HWHG
		15, 23	32, 8	8, 16	18, 6	HWHC
			33, 7	6, 18	12, 12	RDHG
				1, 23	3, 22	RDHC
					16, 8	HTHG

Table 4.6
Final Cost Sign Tests

The final solution sign tests (Table 4.6) provide a similar picture to the Table 4.5, in spite of the higher benefit during the improvement phase by the poorer construction solutions. The run test also detects the previous pairs found during the construction phase with even more pronounced patterns. An additional pair between the HTHG and HWHG heuristics (less than 3% level) is also detected; the HWHG produces better results for smaller problems. This is also consistent with the earlier results which suggest that the HW strategy produces a good starting condition for smaller problems, and the highest gain provides a good insertion strategy in general.

Taking the overall effect into account, the heuristics can be ranked according to the quality of the final solutions as follows:

1 HWHG, HTHG
2 RDHG, HTHC
3 HWWO, HWHC
4 RDHC

Figures 4.9-4.10 show the average construction and final solutions achieved by the HWHG heuristic for all the test problems.

HEURISTICS

HWHG	HWHC	RDHG	RDHC	HTHG	HTHC	
14, 26	11, 29	28, 12	37, 3	0, 25	0, 25	HWWO
	21, 19	33, 7	38, 2	0, 25	0, 25	HWHG
		35, 5	38, 2	0, 25	0, 25	HWHC
			27, 13	0, 25	0, 25	RDHG
				0, 25	0, 25	RDHC
					14, 11	HTHG

Table 4.7
Construction Time Sign Tests

HEURISTICS

HWHG	HWHC	RDHG	RDHC	HTHG	HTHC	
28, 12	25, 15	30, 10	25, 15	0, 25	0, 25	HWWO
	22, 18	22, 18	18, 22	0, 25	0, 25	HWHG
		19, 21	17, 23	0, 25	0, 25	HWHC
			19, 21	0, 25	0, 25	RDHG
				0, 25	0, 25	RDHC
					10, 14	HTHG

Table 4.8
Final Time Sign Tests

The runtime sign test analyses are shown in Tables 4.7-4.8 and the average run times for the construction phase and the average total run times are shown in Figures 4.11-4.12. The construction results conform to the theoretical prediction. The algorithms split into two groups, namely the $O(n^4)$ and $O(n^2)$ groups, *eg* the empirical complexities of the HTHG and HWHG heuristics during the construction phase are $0.02n^{4.09}$ and $0.87n^{2.09}$ respectively. The improvement time, roughly the same as the construction time of the $O(n^2)$ heuristic of the same problem size, has $O(n^2)$ time complexity as expected, consequently the total runtime is $0.40n^{3.87}$ for the HTHG heuristic and $1.63n^{2.06}$ for the HWHG heuristics. The difference in time performances of the two $O(n^4)$ heuristics is negligible. In the other group, the random tetrahedron strategy runs slightly faster than the highest weight strategy during the construction phase. The weight order insertion strategy, although producing a relatively fast solution during the construction phase, requires considerably more execution time during the improvement phase than the rest in the group, and overall runtime of the WO strategy is the highest among the $O(n^2)$ group. The remaining heuristics have very similar runtime performances. There is no significant result for the run tests carried out on the results in Table 4.7-4.8.

4.7 INTERACTIVE ASPECTS

Interactions with the heuristics can be done in two ways; firstly, by artificially manipulating the input data to ensure that certain effects are obtained; and secondly, by imposing additional rules of manipulation. As the input for the MPG is likely to contain certain subjective evaluations, the use of additional rules may be more desirable. One such additional rule, that can be implemented readily, is the restriction of maximum valences of particular nodes to correspond to the physical limitations of the objects being represented. Alternative solutions can be quickly generated by varying the maximum permitted valences.

4.8 CONCLUSIONS

It has been demonstrated that construction and improvement heuristics for the MPG can be implemented effectively using an algorithmic language. Pascal was chosen because the language has data structuring facilities that allow adequate data abstractions. The codes are fast and compact, and they can be used to solve problems with several hundred vertices.

The comparative test results indicate that the use of the heaviest tetrahedron as a starting point does not provide the expected benefit. Moreover with hindsight, it becomes clear why the highest gain insertion strategy during the insertion phase provides better results than those achieved by the highest shadow cost strategy: in other similar combinatorial problems, the assignment of an arc usually results in the total exclusion of the other competing candidates, but this is not usually the case in the MPG.

5 Group Technology: Literature survey

5.1 INTRODUCTION

In the past decade, the emphasis in the literature on Group Technology has slowly shifted away from classification schemes *per se* to the problem of developing methods for grouping components and associated machines. This has led to a variety of approaches which may, for the purposes of this survey, be classified as (i) similarity coefficient (ii) set theoretic (iii) evaluative and (iv) other analytical methods, although it should be pointed out that there is a considerable overlap and interrelationship between these methods.

5.2 SIMILARITY COEFFICIENT METHODS

The similarity coefficient approach is drawn directly from the field of numerical taxonomy and was first suggested by McAuley (1972). The basis of this method is to measure the *similarity* between each pair of machines and then to group the machines into families based on their similarity measurements. In most cases, the similarity measurement used is the coefficient of Jaccard (Sneath & Sokal 1973, p131) which is defined for any pair of machines as: the number of components which visit both machines, divided by the number of components which visit at least one of the machines.

The consequence of defining the similarity coefficient in this way is that equal weightings are given to the requirements and nonrequirements of a particular component insofar as the machines are concerned. As de Beer & de Witte (1978) point out, this may lead to very low values of the coefficient even in cases where a large number of components may require both machines. Another situation where the Jaccard similarity coefficient may not perform satisfactorily is when some machines are required by a large number of components and duplications of these machines are needed. This can, depending on the treatment, result in multiple values of the coefficients. None of the papers reviewed discuss this problem explicitly.

The second problem associated with the similarity coefficient approach is the use of a *threshold value* such that if a coefficient is less than this limiting value the coefficient will be ignored in the next stage of the algorithm. There is however, a large degree of arbitrariness involved in this. Rajagopalan & Batra (1975) suggest a more systematic way of finding the threshold value, but in spite of this, the arbitrary nature of the selection still persists, as evidenced by the final choice of the threshold value in their paper.

In grouping machines, McAuley (1972) uses *Single Linkage Cluster Analysis* (SLCA). "This method first clusters together those machines mutually related with the highest possible similarity coefficient, then it successively lowers the level of admission by steps of predetermined equal magnitude. The admission of a machine or groups of machines into another group is by a criterion of single linkage." However, as McAuley points out "the main disadvantage of this method is that while two clusters may be linked by this technique on the basis of a single bond, many of the members of the two clusters may be quite far removed from each other in terms of similarity." To overcome this problem, various methods have been suggested by McAuley and Sneath & Sokal, but at the cost of having to define more limiting values.

Carrie (1974) has used McAuley's method in an actual case involving additional problem constraints, such as, for example, a requirement of a minimum number of machines per group. However, no detailed results of the implementation are reported.

Rajagopalan & Batra (1975) developed a graph-theoretic method which uses cliques of the *machine-graph* as a means of classification. The vertices of this graph are the machines, the arcs are the Jaccard similarity coefficients and a clique is a maximal collection of vertices, every pair of which is connected by an edge of the graph. The main disadvantage of this approach is that because of the high density of the graph, a very large number of cliques is usually involved and many of the cliques are not vertex disjointed. To reduce the number of groups and to incorporate the machines which are not included in the cliques, graph partitioning is used, and it is at this stage that the allocation of components, in accordance with a number of heuristic rules, is also carried out.

As the number of cliques varies exponentially with the number of vertices (Moon & Moser 1965), the clique approach may be acceptable for a few machine types, however the complicated and time consuming nature of the allocation procedure means that application to a large problem would be very difficult.

de Beer *et al* (1976) and (1978) describe a modified form of Burbidge's *Production Flow Analysis.* An important aspect of this approach is the development of a method of cell formation based on an analysis of operation routings and the *divisibility* of operations between machines, and hence between cells. This divisibility is governed by the numbers of machines of the required types that are available for undertaking specific operations. Three categories of machine types are defined: primary or key, where only one such machine is available; secondary, where several machines are available; and tertiary, where there are sufficient machines available to be able to assign to each cell if required. de Witte (1979), in a further extension of this approach, suggested the use of three similarity coefficients which are different from Jaccard's and are specifically designed to indicate the interdependence of machine types within the three categories mentioned above. The subsequent clustering of machine types into cells is carried out using the SLCA method, not the clique method as suggested in the paper. In addition, it is not clear how de Witte's method could cope with the situation where not all the machines available are required, or alternatively, where additional machines may economically be justified. Lastly, it is arguable whether there is any need to include the tertiary machines in the process, since by definition they are available for inclusion in every cell. Capacity considerations alone should be adequate for determining how these machines should be allocated.

None of the above papers considers the sensitivity of the solution in relation to the procedure used in the formation of the cells and, in particular, the form of the similarity coefficients used. By their very nature, similarity coefficients are aggregate measures and hence during their manipulation information losses are inevitable, and the significance of these losses ought to be clearly established before the procedures described can be used with confidence.

5.3 SET-THEORETIC METHODS

In spite of various titles given to his papers, Purcheck(1974, 1975*a*, 1975*b*) has adopted throughout a common set-theoretic approach to the problem. The earliest paper describes a systematic way of using union operation on the sets of machines required for various components, in order to arrive at the supersets (termed *hosts* and *superhosts*) which progressively include more and more components. The process of building up these supersets can be represented as a path along the edge of a lattice diagram. This method significantly reduces the total number of possible solutions. The process is fundamentally similar to those described by Burbidge (1971, 1973) and El-Essawy (1972), but is specified in a much more explicit manner.

The lattice diagram is at best only useful as a general illustrative device. The lattice diagrams actually drawn by Purcheck (1974, 1975*a*), complicated as they are, represent the combinations of only 6 machines. It is true that not all the possible points in the lattice need to be represented in practice. However, the exponential growth in the number of lattice points with increasing number of machines means that a stage is soon reached where the lattice diagram becomes virtually unintelligible.

Purcheck (1975*a*) also develops a classification scheme which combines machine requirements and sequences by codifying them respectively in the form of long strings of letters and digits. In the example given in which 19 machines are involved, code lengths of 15 or more are not uncommon. The code length requirement is a crucial limitation and dashes any real hope of applying the scheme to problems with large numbers of machines. It is also difficult to see why such packing of information would improve the efficiency of grouping the machines. Mathematical programming (linear, combinatoric) is suggested as a means of carrying out the grouping process. There is, however, insufficient description in the paper to show how the constraint matrices could actually be constructed and there is no specification of the objective function to be used.

The use of a set partitioning technique to solve an LP formulation of the problem is advocated by

Purcheck (1975*b*). The cost function however, is not, in general, stated explicitly. In the worked example, the cost function is the total capital costs of the machines involved. In actual practical application, most of the machines, if not all, would already be available. The main benefits of group production, shorter throughput time, and hence reduced work-in-progress etc., are not included. As in the previous paper (1975*a*), the constraint matrices are not explicitly given. How various cells would constrain the problem is not at all clear, and the problems of machine utilization and duplicated machines are not defined. It is difficult to see how the LP problem as formulated could represent any real group layout problem.

It is not clear how optimisation methods in general, and mathematical programming in particular, can be applied successfully to this problem; at least in the near future. A satisfactory definition of the objective function to include only quantifiable aspects of the problem would be lengthy, complex and unlikely to be linear. The constraint matrices would necessarily be large in order to define the whole problem adequately. Even the much simpler quadratic assignment problem (QAP) is notoriously difficult to solve, as discussed in the previous chapters. The QAP considers only the material handling costs, whereas the group layout problem involves a large number of interacting factors, many of which are highly dynamic. Fifteen machines is the present limit of most optimization procedures for the QAP, though sub-optimal procedures are able to solve somewhat larger problems.

5.4 EVALUATIVE METHODS

The concept of *Production Flow Analysis* (PFA) was first introduced by Burbidge (1963). The aim of the technique was stated by Burbidge (1971) as that of "finding the families of components and associated groups of machines for group layout... by a progressive analysis of the information contained in route cards...". PFA has since been developed, extended and given various names. The main feature of the evaluative approach to PFA is that it involves the systematic listing of the components in various ways, in the expectation that groups of machines and components may be found by careful inspection. As de Beer & de Witte (1978) point out, the procedure requires "a series of evaluations to be made by (the) designer, more or less calling upon his ability to recognize patterns". Burbidge's approach to PFA consists of three levels of analysis. *Factory Flow Analysis*, the first stage, makes use of *Process Route Numbers* (PRNs), in order to get an overall picture of the present state of material flows. Machines are divided into departments, and each department is given a number (in the example quoted, one digit figures are used). The PRN of a component is defined as the sequence of the numbers of the departments visited. A flow chart showing the interaction of various departments based on PRNs is then drawn. Burbidge gives various suggestions as to how this chart can be simplified and once this is done, each department is analysed in turn. This constitutes the second step, called Group Analysis. With the information obtained by sorting components into packs, according to the machines required, the designer then proceeds to form families of machines and components mainly by reordering the rows and columns of the *Component-Machine Chart* to create as near a block diagonal form as possible (the significance of this block diagonal structure is considered in more detail later in this chapter). Burbidge (1971) does not explain explicitly how the outcomes were achieved. The difficulty was discussed in Burbidge (1973), in which the author states: "Fifteen different methods were tried before a reliable solution was obtained." The "best" method, called *Nuclear Synthesis*, is based on selecting machines used by few components as starting points for various cells, or nuclei, as Burbidge terms them. The next machine is allocated on the basis that it has the smallest number of components left unassigned to a group. Once Nuclear Synthesis is completed, these nuclei are modified and subject to certain special reservations, combined in a manner similar to that of Purcheck's superset approach, until the required number of groups is formed. Burbidge (1977) describes how the process can be carried out manually. The third stage, *Line Analysis*, is a procedure to find a layout in each group which will give the nearest approximation to line flow.

Burbidge's approach consists of a series of subjective evaluations, which require substantial local knowledge in order to make any well-informed judgements. It is not surprising, as has been discussed by Edwards (1972) and El-Essawy (1972), that most of the attempts to apply the procedures have not been entirely satisfactory. Admittedly, most of the critical comment had been made before Burbidge introduced the method of Nuclear Synthesis, but it is not clear how well this works in practice and whether it has overcome the earlier criticism. The process of modification and combination of nuclei is artificially restricted by the predefined number of groups. The number of groups is in part determined by what is deemed to be a "*sociologically acceptable size*" which Burbidge considers to be from 6 to 12 workers; in his example Burbidge uses the mean value of 9. However, the number of groups would have changed by as much as 50% either way, if instead of choosing the mean value, Burbidge had chosen the lower limit of 6 or the upper limit of 12 for the "sociologically acceptable size".

In spite of various difficulties, Burbidge's approach highlights the importance of partitioning the problem into subproblems of manageable size. Without partitioning, the effort required to solve larger problems would be excessive. Perhaps the most important conclusion that can be drawn from Burbidge's work is that there is a large number of factors which cannot, at least for the time being, be formulated explicitly but which could crucially affect the final outcome.

Component Flow Analysis (CFA) was first used in 1971 and distinguished as being different to PFA (El-Essawy, 1971; El-Essawy & Torrance, 1972), and in spite of various claims and counter claims, the similarity of the two approaches is apparent. CFA is made up of 3 stages of analyses. The objective of the first stage is "to consider the total component mix of the company and to identify and sort components into categories according to their manufacturing requirements". In essence, this stage consists primarily of sorting the components in the order of machine requirements and printing out the sorted list in two ways, firstly in the order of the number of machines required and secondly in the order of the smallest machine numbers involved, ready to be manually analysed in the second stage. The aim of the second stage is to obtain groupings of the machines using the lists of sorted components and taking into account various local constraints. Rough groups are formed by using the combinations with the highest number of machines as the cores (cf Burbidge's nucleus, Purcheck's host), to which other machines and components are successively added. The third stage involves a detailed analysis of the loadings and flow pattern of the cells with appropriate adjustments to ensure that an acceptable design is achieved.

In some respects, the methodology of CFA does differ from that of PFA. For example, PFA first partitions the problem, whereas CFA does not. The manner in which the cells are built up is also different in the two methods. CFA also relies less on the subjective evaluation, since the way in which problems can be tackled is described more precisely. Both methods, however, stress the importance of local factors which it is not easy to formulate explicitly, and the need for careful analysis of data both before and after group formation.

An attempt has been made by de Beer & de Witte (1978) to extend the basic approach of PFA to explicitly consider both the question of machine duplication and different characteristics of the machines. This method has been termed *Production Flow Synthesis* (PFS). One major difference between PFS and the other methods discussed in this section is that the number of components that require more than one cell is quite substantial. In the case study described, only 46% of components could be accommodated in single cells. There is also no detailed account of how various cells are formed, a process which is crucial to both PFA and CFA.

5.5 OTHER ANALYTICAL METHODS

As Gallagher & Knight (1973) have pointed out: "The crux of the problem of introducing group technology is the identification, from the large variety and total number of components, of the families requiring similar manufacturing operations on similar machine tools". Unfortunately, as Burbidge (1973, p7) states "It has proven to be surprisingly difficult to find a method suitable for the computer". El-Essawy & Torrance (1972, p167) came to a similar conclusion: "... the use of a computerised method to decide on these 'rough' groupings requires an unjustifiably sophisticated procedure".

The processing requirements of components on machines can be represented in graph theoretic terminology as a bipartite graph $G(V_m, V_c, A)$ where V_m and V_c are the two sets of vertices of the graph which correspond respectively to the machines and components. A is a set of arcs of the graph such that:

1 If an arc exists between machine vertex i and component vertex j $(a_{ij}=1)$ then component j requires processing on machine i

2 If an arc does not exist between machine vertex i and component vertex j $(a_{ij}=O)$ then component j does not require processing on machine i.

Each vertex of the graph can be viewed as a compound element if so desired and components which require exactly the same set of machines may be depicted as a single vertex. Similarly machines of the same type can, if required, be represented as a single vertex. Such devices can be used to reduce the overall size of the graph.

The processing requirements of the components on the machines are also specified by the incidence matrix representation of the bipartite graph. It is easy to see that in this form the problem of allocating machines to groups and components to associated families reduces to that of finding a block diagonal form of the $a_{ij}=1$ entries in the incidence matrix by appropriately rearranging the order of rows and columns. An example of a machine component incidence matrix is shown in Figure 5.1.1 (where it should be noted that all $a_{ij}=0$ values are shown as blank entries). Figure 5.1.3 shows a block diagonal arrangement achieved by row and column changes that produces a solution of the two machine groups with two associated component families.

There are many algorithms which would readily identify a block diagonal form, if one exists. With the exception of the ROC algorithm, the methods to be outlined have not been specifically tailored or designed for the group formation problem in Group Technology. Iri (1968) suggests one of the simplest methods, using a masking technique. This may be described briefly as follows: Starting from any row, mask all the columns which have an entry in this row, then proceed to mask all rows which have entries in these columns. Repeat the process until the numbers of masked rows and columns stop increasing. The masked rows and columns constitute a block. If none exists, the entire matrix is masked as one group. It is not, however, possible to modify this procedure to take account of the case where there might be, say, a few non-conforming elements in what would otherwise be a pure block diagonal problem.

McCormick *et al* (1972) have developed a matrix clustering technique which they call the *Bond Energy Algorithm* (BEA). The BEA is applicable to any matrix in which non-negative integer values of an element in the matrix express a measure of the degree of association of the corresponding row and column entities. What the BEA seeks to determine is a permutation of the rows and columns in which the sum of the products of adjacent elements is maximized. This is a restricted form of the quadratic assignment problem. The BEA is a sub-optimising procedure which uses a single pass heuristic applied to both rows and columns. The algorithm will reveal a block diagonal form if one exists. However, it is more difficult to predict the behaviour of the algorithm in cases where there exist a few exceptional elements that cannot be fitted into such an arrangement.

King (1979) shows that if the patterns of row entries are read as binary words they can be ranked in reducing binary value order. This then permits the rows to be rearranged in accordance with this rank order. The same procedure can be repeated on the columns. This process may be repeated for rows and columns alternately until no further rearranging of rows and columns is possible, at which point a block diagonal form will be produced if one exists.

This process is illustrated in relation to an example problem with the machine-component incidence matrix shown in Figure 5.1.1. Binary ranking by row leads to the rearrangement of rows to form the matrix shown in Figure 5.1.2. Binary ranking of the columns of Figure 5.1.2 leads in turn to a rearrangement of columns to form the matrix of Figure 5.1.3. The latter cannot be rearranged further and, as will be seen, constitutes a block diagonal form.

This particular procedure of reading the entries as binary words presents some computational difficulties. Since the largest integer representation in most computers is 2^{48}-1 or less, the maximum number of rows or columns that could be dealt with in this way would be 47. To overcome this limitation, element by element comparisons for carrying out row or column ranking are used. For example, row 1 (*0101110*) and row 4 (*0101010*) of the matrix in Figure 5.1.1 are compared successively digit by digit from left to right. Five comparisons are needed to conclude that the index of row 1 is larger than that of row 4, as the first four pairs of digits are the same. The process is repeated for the other rows until the complete row ranking is obtained. The procedure is applicable to column ranking as well and it is the basis of the iterative *Rank Order Clustering* (ROC) algorithm developed by King (1979, 1980). This procedure has a computational complexity of cubic order, namely $O(mn(m+n))$, where m and n are the numbers of rows and columns respectively.

The block diagonal structure illustrated in Figure 5.1.3 is the exception rather than the rule. If it exists then the ROC algorithm will generate it. More commonly the elements in the matrix are such that they cannot be divided into mutually exclusive diagonal groups. This case presents no real problem since the ROC algorithm can still be used to generate a diagonal structure which may contain one or more elements that do not conform to the block form. These elements can be considered as *exceptional elements* comprising machine-component combinations that would not form part of the the machine-component groups represented by the remaining pure diagonal blocks. As a simple illustration, if the matrix of Figure 5.1.1 had contained an additional *1* element, say (3,6), then the ROC algorithm would have produced, after two iterations, the final result shown in Figure 5.2. It will be seen that this contains exactly the same groupings as the result shown in Figure 5.1.3, except that now (3,6) is an exceptional element.

The formal procedure for dealing with the exceptional elements adopted by King may be described as follows: (i) Use the ROC algorithm to generate a diagonal structure (with probably one or more overlapping groups). (ii) Identify the exceptional elements (those elements in overlapping groups whose removal would allow a separation of the group to be achieved). (iii) Temporarily ignore the exceptional elements so that the ROC algorithm can be continued to enable a block diagonal form to be produced. (iv) Reinstate in this final matrix the previously ignored exceptional elements designating them by asterisks instead of 1's.

The explicit identification of exceptional elements in this way allows us to concentrate on only a small part of a matrix at a time; namely the potential overlap between any two groups. Consequently, even in cases where there are a large number of exceptional elements, this procedure can still be used to deal step by step with the exceptional elements in all the potential overlaps.

By way of illustration the original matrix in Figure 5.1.1 is modified to include additional elements (3,6) and (5,5): In this case stage (i) of the procedure would generate the matrix shown in Figure 5.3.1. Stage(ii) would identify (3,6) and (5,5) as exceptional elements. Stage(iii) would generate the block diagonal groups of *1*'s shown in Figure 5.3.2 and stage(iv) would insert the asterisks indicating that (3,6) and (5,5) are the exceptional elements.

Where particular types of machines are required by a large number of components, King(1980) suggests a relaxation procedure which determines the number of duplicated machines required to eliminate the bottleneck, as well as their disposition in the block diagonal structure produced. This procedure, however, greatly increases the dimension of the matrix because it begins by assuming a relaxation of one machine to one component. As the computational complexity of the ROC algorithm is of cubic order, this is a severe practical limitation on the use of this procedure for problems of anything other than modest size.

There is another approach similar to the ROC algorithm for clustering data where, instead of weighting the positions of the rows or columns in an exponential manner, the weights are increased linearly (Graham *et al*, 1976). In the specific archaeological application described by Graham *et al* the i^{th} row is given a weighting of $m-i+1$, where m is the total number of rows, and the priority ranking value is determined as the mean of the weightings of the non-zero entries. Ranking values calculated this way can be found and sorted very quickly and the requirement of a very large integer representation does not arise. In practice, the clustering algorithm is used to compress the entries into a band along the major diagonal of the matrix. If a block diagonal form exists the procedure will determine it. If this occurs then the attempt to determine a time seriation of archaeological evidence has failed: thus, in complete contrast to machine and component grouping, the hoped for result in any archaeological application is that the data will not break down into a block diagonal form. The major disadvantages of this linear weighting algorithm are the complicated and very confusing patterns of the intermediate results together with the difficulty in predicting the behaviour of the procedure.

BINARY WEIGHTS		2^6	2^5	2^4	2^3	2^2	2^1	2^0	
				COMPONENTS					BINARY
		1	2	3	4	5	6	7	RANKING
	1		1		1	1	1		4
	2	1		1					2
MACHINES	3	1		1				1	1
	4		1		1		1		5
	5	1						1	3

Figure 5.1.1

					COMPONENTS				
BINARY WEIGHTS			1	2	3	4	5	6	7
2^4		3	1		1				1
2^3		2	1		1				
2^2	MACHINES	5	1						1
2^1		1		1		1	1	1	
2^0		4		1		1		1	
BINARY RANKING			1	4	2	4	7	4	3

Figure 5.1.2

				COMPONENTS					BINARY
		1	3	7	2	4	6	5	RANKING
	3	1	1	1					1
	2	1	1						2
MACHINES	5	1		1					3
	1				1	1	1	1	4
	4				1	1	1		5
BINARY RANKING		1	2	3	4	4	4	7	

Figure 5.1.3

Figure 5.1
Matrix sorting using the ROC algorithm

COMPONENTS

		1	3	7	6	2	4	5
	3	1	1	1	1			
	2	1	1					
MACHINES	5	1		1				
	1				1	1	1	1
	4				1	1	1	

Figure 5.2

Figure 5.1.1 with an additional element

COMPONENTS

		1	3	6	7	2	5	4
	3	1	1	①	1			
	2	1	1					
MACHINES	5	1			1		①	
	1			1		1	1	1
	4			1		1		1

Figure 5.3.1

COMPONENTS

		1	3	7	6	2	5	4
	3	1	1	1	*			
	2	1	1					
MACHINES	5	1		1			*	
	1				1	1	1	1
	4				1	1		1

Figure 5.3.2

Figure 5.3

Sorting matrix with exceptional elements

6 The Design and Applications of the ROC2 Algorithm

6.1 INTRODUCTION

Of the papers reviewed in the last chapter, most tend to favour either similarity coefficient or evaluative methods. As has been discussed in chapter 5, these approaches exhibit certain weaknesses: the more important ones being firstly, the fact that the clustering techniques used in the similarity coefficient methods are either too weak (in the case of SLCA) or too rigorous (in the case of cliques), and secondly, the limitation on the size of problem that can be handled by evaluative methods. The explicitness of the similarity coefficient and the flexibility associated with evaluative methods are highly desirable characteristics. It is perhaps worth noting that explicitness and flexibility are combined features of the improved and extended ROC procedure to be described later.

The ROC algorithm at its previous stage of development by King (1980) has a number of major limitations. Firstly, the storage of the incidence matrix as a two dimensional array puts a severe limit on the size of the problem that can be tackled. A moderate problem with 50 machines and 2000 components, together with the program, would require core storage in excess of 120 *K* words. Secondly, because the sorting procedure has a complexity of cubic order, efficient implementation is not possible for very large problems. The situation is exacerbated if the relaxation procedure mentioned in the last chapter is included, since this significantly increases the dimensionality of the problem.

By sorting with several rows or columns at the same time, instead of element by element, the efficiency of the sorting procedure can be improved, even though this requires additional calculation to find the priority ranking values for these rows and columns. By this device, and in conjunction with an efficient computer sorting procedure, such as *Quicksort* or *Mergesort*, the overall complexity may be reduced to $O(mn\log(mn))$, compared with $O(mn(m+n))$ achieved previously. Considerable improvement in the computational efficiency can thus be achieved by this process, which has particular relevance where problems involving large machine-component incidence matrices are concerned.

An even faster sorting procedure that can be used in conjunction with a linked data structure to be described is *Least Significant Digit Radix Sort*. Radix Sort does not incur the overhead of ranking value calculations and the way in which the data are stored also means that part of the radix procedure is already carried out, so that the overall effect is to provide an algorithm with a complexity of $O(k)$, where k is the number of non-zero entries. The whole sorting procedure is thus reduced to that of shifting the order of rows and columns which is designated ROC2, to distinguish it from the earlier ROC algorithm described by King (1979, 1980).

6.2 DESIGN OF THE ROC2 ALGORITHM

The first major restriction that needs to be overcome by the new algorithm is the storage requirement of the original implementation. Without a better storage scheme, only moderate sized problems can be solved in this way. Since incidence matrices of the kind involved in Group Technology problems are usually very sparse, with densities unlikely to be higher than 5-10%, an elaborate system of linked list structures would in general be economical. Various structures can be found in the literature (Pooch & Nieder 1973; Berztiss 1975; Horowitz & Sahni 1976). The use of a list structure brings two kinds of advantage. Firstly, by storing only the non-zero elements the algorithm would only operate on the non-zero elements, which form a very small proportion of all the elements of the matrix. Secondly, in appropriate cases, list structure can be treated as analogous to the grouping together of numbers with the same radix in the Least Significant Radix Sorting procedure. The operation of Radix Sort can be illustrated by the following example. Consider the sequence of numbers 11, 32, 13 and 21. This sequence may be divided into three groups, as there are three radices *1*, *2* and *3* involved, according to the last (*i.e.* least significant) digit. As 21 has 1 as the last digit, it is entered into radix band *1*, 13 has 3 as the last digit and is therefore put into radix band *3* and so on, as illustrated in Table 6.1.1. At the end of this process the intermediate sequence is 13, 32, 11 and 21. If the process is repeated on this sequence but with the division being made in accordance with the next significant digit (*i.e.* so that 21 is entered into radix band *2* and 11 into radix band *1*, and so on) then the final sequence, as illustrated in Table 6.1.2, will be 32, 21, 13 and 11.

```
               .  .   .  .   .  .              .  .   .  .   .  .
               .  .   .  .   .21.              .  .   .  .   .11.
               .13.   .32.   .11.              .32.   .21.   .13.
               . . . . . . . . .               . . . . . . . . .
RADIX BAND     . 3.   . 2.   . 1.              .3 .   .2 .   .1 .
               . . . . . . . . .               . . . . . . . . .
INTERMEDIATE                               FINAL
    SEQUENCE    13   32   11   21      SEQUENCE 32   21   13   11
```

Table 6.1.1 Table 6.1.2

In the case of binary numbers the number of the radix bands is essentially reduced to one, as any number not assigned to the band of digit one, is assumed to have digit zero for that particular

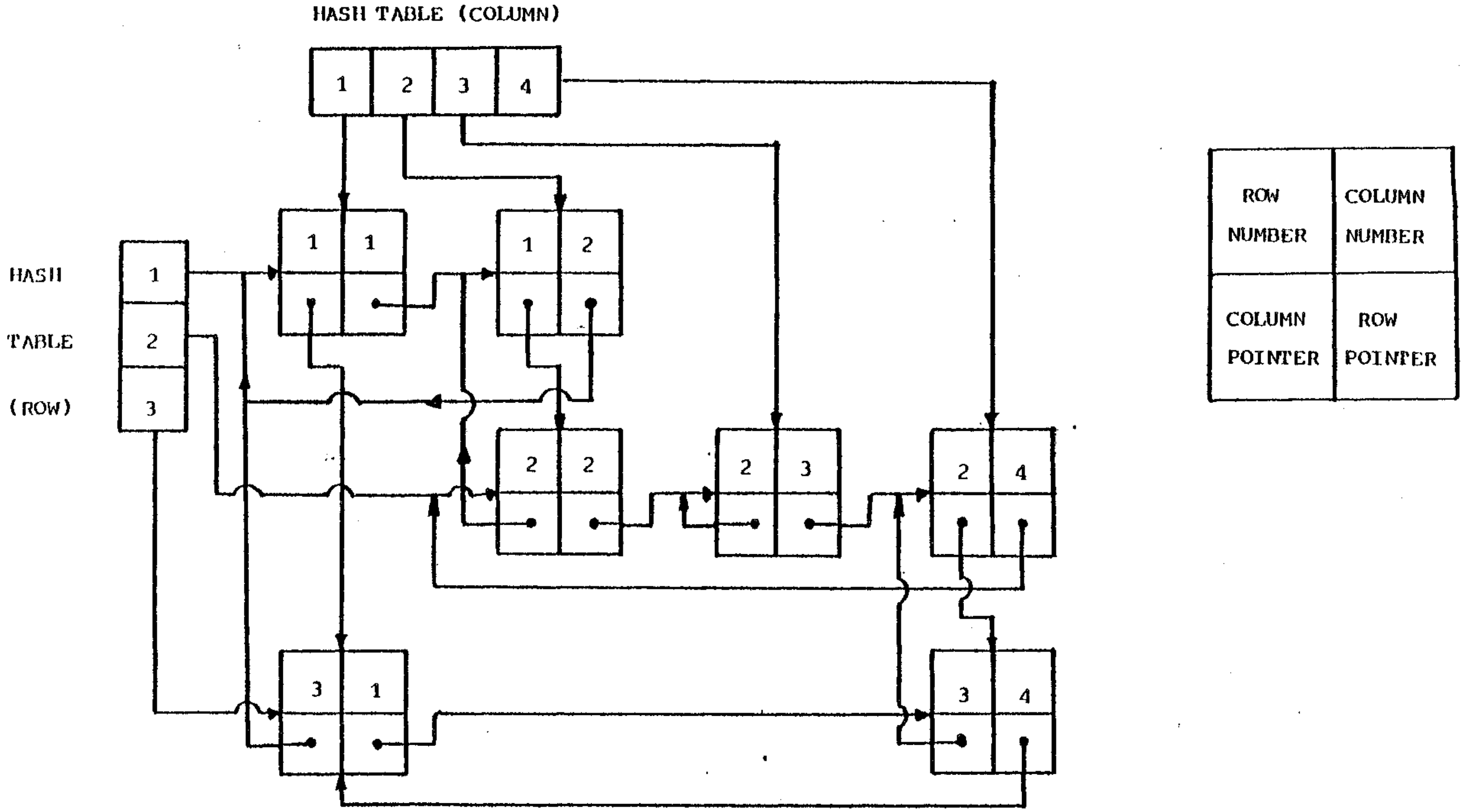


Figure 6.1
A diagram of a storage scheme for the ROC2 algorithm

band. In the case of sorting a binary matrix the radix bands are, in effect, the rows or columns of the matrix. List structure thus readily divides the entries into appropriate subgroups. In order that both the rows and the columns may be easily accessed, a double list structure is required. Circular lists may be appropriate in some applications. An example of such a structure with two hash tables is represented diagramatically in Figure 6.1. Two hash tables are used to allow convenient random access of any row or column.

Figures 6.2.1 - 6.2.5 illustrate how the radix sorting procedure can be applied to the sorting of a matrix. In the case of row sorting, columns become radix bands, and in column sorting rows become radix bands. As rows *2* and *3* have 1's in the fourth column, row *2* and *3* are moved to the first and second positions respectively in front of row *1*. The process is repeated with all the remaining columns. The process can be reproduced using the list structure. The non-zero elements in the fourth column can be found by accessing the data structure via the hash table (column). In this case, rows *2* and *3* could be identified readily as shown in Figure 6.2.1. To indicate this fact, 2 and 3 in Figure 6.2.1 in the row order are underlined. The identified rows are moved to the head of the queue to form an intermediate sequence, to be sorted again according to the next radix. As can be seen, the matrix can be sorted by manipulating the row or the column order, without having actually to move parts of the matrix around.

```
                          RADIX          STARTING
                            !            ROW ORDER
(1)    1    1    0    0
(2)    0    1    1    1                  1   2   3
(3)    1    0    0    1
```

Initial matrix
Figure 6.2.1

```
                    RADIX                  INTERMEDIATE
                      !                    ROW ORDER
(2)    0    1    1    1
(3)    1    0    0    1                    2    3    1
(1)    1    1    0    0
```

Matrix after the first pass
Figure 6.2.2

```
               RADIX                       INTERMEDIATE
                 !                         ROW ORDER
(2)    0    1    1    1
(3)    1    0    0    1                    2    3    1
(1)    1    1    0    0
```

Matrix after the second pass
Figure 6.2.3

```
     RADIX                             INTERMEDIATE
       !                               ROW ORDER
(2)    0      1      1      1
(1)    1      1      0      0          2    1    3
(3)    1      0      0      1
```

Matrix after the third pass
Figure 6.2.4

```
                                       ROW ORDER
(1)    1      1      0      0
(3)    1      0      0      1          1    3    2
(2)    0      1      1      1
```

Matrix after the first iteration.
Figure 6.2.5

In order that the removal of exceptional elements, assignments of components to duplicated machines, and the transfer of components between machines of the same types may be carried out quickly in the ROC algorithm without a major disruption of the entire structure, the data structure of the incidence matrix may be rearranged so that it comprises four main cells for each entry and two hash tables. The two hash tables, one for the rows and one for the columns, are simply efficient programming devices that allow the computer quick access to any row or column. The four cells represent the row and the column of the entry, together with pointers to the next elements along the same row and column. These pointers are part of the circular, double-linked list structure. Circular lists are chosen because they allow better access in the removal or reassignment of an entry.

The algorithm can be summarized as follows:

ROC2 Algorithm:

```
REPEAT
     FROM the last column TO the first column
     DO{row reordering}
          locate the rows {machines} with entries;
          move the rows with entries to the head of the row list,
               maintaining the previous order of the entries
     END DO{row reordering};
     FROM the last row TO the first row
     DO{column reordering}
          locate the columns {components} with entries;
          move the columns with entries to the head of the column list,
               maintaining the previous order of the entries
     END DO{column reordering}
UNTIL (no change OR inspection required)
```

6.3 ILLUSTRATION OF THE ROC2 ALGORITHM IN USE

Consider again the example problem represented by the matrix shown in Figure 5.1.1 but this time using the ROC2 algorithm. The stages involved in row reordering of the matrix are shown as successive lines in Table 6.2.1. The first line shows the initial row list in which, for the last column, column *7*, the underlined entries *3* and *5* are the machines in this column and are moved in this order to the front of the list, as indicated in line *2* of Table 6.2.1. For the next column of the matrix, column *6*, the machine entries are *1* and *4* and are indicated by underlining in line *2* of Table 6.2.1. These entries are moved to the front of the list to form line *3* of Table 6.2.1 where, in the next column, column *5*, of the matrix, machine *1* is the only entry and is already at the head of the list so that no change is necessary in this case. This process is repeated for each of the remaining columns of the matrix of Figure 5.1.1, and finally results, as indicated in the last line of of Table 6.2.1, in the new row order of *3,2,5,1,4* being determined.

		Row list				
For column no.	7	1	2	3	4	5
	6	3	5	1	2	4
	5	1	4	3	5	2
	4	1	4	3	5	2
	3	1	4	3	5	2
	2	3	2	1	4	5
	1	1	4	3	2	5
New row order		3	2	5	1	4

Table 6.2.1
Stages in row reordering using the ROC2 algorithm

Column reordering is carried out in a similar way but starting with the current column order *1, 2, 3, 4, 5, 6, 7* and the current row order *3, 2, 5, 1, 4* (this is equivalent to Figure 5.1.2), and the stages involved are shown as the successive lines of Table 6.2.2, where the new column order is determined as *1, 3, 7, 2, 4, 6* and *5.*

		Column list						
For row no.	5	1	2	3	4	5	6	7
	4	2	4	6	1	3	5	7
	3	2	4	6	5	1	3	7
	2	1	7	2	4	6	5	3
	1	1	3	7	2	4	6	5
New column order		1	3	7	2	4	6	5

Table 6.2.2
Stages in column reordering using the ROC2 algorithm.

It will be seen that the final row and column orders are the same as those in Figure 5.1.3.

6.4 A NEW RELAXATION PROCEDURE

One of the most difficult problems in using the algorithms to group machines and components is that some machines are required by a large number of components. Most algorithms discussed have not contained any effective means of dealing with this problem at all. Yet, if there is to be any hope of applying such an algorithm in practice, this problem must be overcome.

If these machines are treated in the normal way, they will dominate the results in such a way that no effective grouping could be deduced. By giving them a high priority as in King's (1980) relaxation procedure, the side effect, namely the very large increase in the dimensionality of the problem, becomes unacceptable.

The method proposed here is to give these machines less emphasis. By their nature, they tend to be either simple machines or highly sophisticated ones. In cases where they are fairly simple, like centre lathes, they tend to exist in large numbers and hence will be available in more than one cell. If they are highly complicated machines which are capable of a large range of operations, they would need to be treated separately. In either case, by disregarding them during certain stages of grouping in order to remove their dominant effects, and reinstating them at a later stage, it is possible to find the underlying pattern which otherwise might not be found.

Hence, a new relaxation procedure for the bottleneck machines is simply to ignore those machines (rows) during the shifting process. This has the effect of slightly reducing the size of the problem instead of greatly increasing it as was the case in King's relaxation method mentioned earlier. The operation of this new procedure can be best illustrated by considering the example shown in Figures 6.3.1 to 6.3.4. The ROC2 algorithm was applied to the original incidence matrix of Figure 6.3.1, in the manner already described. It is clear, as shown in Figure 6.3.2 (the result generated after the two iterations of the algorithm), that machines *8* and *6* are required by a large proportion of the components and may thus be considered to be bottleneck machines. Two further iterations of the ROC2 were therefore carried out, but ignoring the bottleneck machines *8* and *6*. The result, as shown in Figure 6.3.3, is that a general but incomplete pattern of a block diagonal form begins to take shape. At this stage, various block diagonal combinations are possible, depending upon the numbers of machines *8* and *6* that can be provided. For example, if there are two of each of these machines available, then only two distinct machine-component blocks are feasible. Reference to Figure 6.3.3, however, shows that there are three possible alternative band mergings, namely (i) *1* and *2*, *3* and *4*, (ii) *1* and *3*, *2* and *4*, (iii) *1* and *4*, *2* and *3*. After merging, the ROC2 algorithm must be applied again to carry out the required regrouping. Figure 6.3.4 shows a combination which requires four machines *8* and three machines *6*, with one exceptional element. This was achieved by simply allowing each band (except band *4*) naturally to form a block with the machines *8* and *6*, and since there was only one component (*no. 3,4*) requiring machine *6*, it was decided to assign this component to machine *6* in band *2*. The result compares favourably with King's (1980) previous solution (four *8*'s, four *6*'s and two exceptional elements) and Burbidge's (1973) solution (four *8*'s, four *6*'s and three exceptional elements).

```
     FLOW MATRIX AFTER     0 ITERATION(S)

           LOCATIONS
           0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4
           1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
           COMPONENTS
           0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4
           1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
(  1)   1                                                                          1         1
(  2)   2    1               1                                   1       1         1 1   1   1
(  3)   3              1                   1                                 1 1 1
(  4)   4          1       1         1         1   1   1           1
(  5)   5          1     1 1         1 1 1     1   1   1           1       1               1   1
(  6)   6  1 1       1 1 1       1 1 1     1   1       1                 1 1 1     1   1 1   1 1
(  7)   7  1                       1                       1
(  8)   8  1 1 1         1 1   1 1     1       1 1 1   1 1     1 1     1           1 1     1   1
(  9)   9    1   1           1               1                   1       1         1 1   1   1
( 10)  10  1                     1 1                       1 1         1               1
( 11)  11      1           1                     1       1     1     1
( 12)  12                    1                     1   1     1     1
( 13)  13      1                                         1
( 14)  14    1       1                     1                                   1
( 15)  15          1                 1         1   1                       1               1   1
( 16)  16    1         1     1               1                           1         1 1       1
```

Figure 6.3.1

```
      FLOW MATRIX AFTER     2 ITERATION(S)

          LOCATIONS
          0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4
          1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
          COMPONENTS
          0 1 0 3 1 2 4 0 3 3 2 2 0 0 2 2 2 4 1 1 1 3 4 3 4 1 0 0 1 3 3 2 2 1 1 0 3 3 0 2 1 3 2
          1 2 2 7 9 3 3 8 1 8 8 4 3 9 7 0 1 1 5 1 3 9 2 2 0 7 6 7 4 3 4 5 6 0 8 4 5 0 5 9 6 6 2
(  1)   8 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
(  2)   6 1 1 1 1 1 1 1 1                         1 1 1 1 1 1 1 1 1 1 1
(  3)  10 1 1             1                       1 1                   1 1
(  4)   7 1                                       1                     1
(  5)   9     1 1           1 1                       1 1 1                 1 1 1
(  6)  14     1                                             1 1                   1
(  7)  16     1 1           1                         1 1       1           1 1
(  8)   2     1 1           1 1                       1 1 1                 1
(  9)  11                       1 1 1 1 1                                           1
( 10)  13                       1 1
( 11)   5         1 1 1 1           1     1 1 1                   1 1                 1 1 1
( 12)   4         1 1               1     1                       1                   1 1
( 13)  15         1   1                   1 1                     1 1                 1
( 14)   3                                                   1   1     1         1         1
( 15)  12                       1     1         1                                   1         1
( 16)   1       1                                   1
```

Figure 6.3.2

```
     FLOW MATRIX AFTER     4 ITERATION(S)

          LOCATIONS
          0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4
          1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
          COMPONENTS
          0 1 2 1 3 3 2:0 3 4 3 3 1 2 4 1 0 0 1 3 0:1 2 1 0 0 2 2 4 4 3 0 1 1:2 0 2 3 2 1 2 3 3
          1 3 5 2 1 9 6:2 7 2 8 2 0 8 0 8 4 7 7 5 6:9 1 4 5 9 3 9 3 1 3 8 5 6:4 3 7 0 0 1 2 4 6
(  1)   8 1     1 1    :1 1   1     1              :1 1     1 1   1 1   1 1   :1 1 1   1 1
(  2)   6 1 1   1   1  :1 1 1   1     1     1 1   1:1   1     1   1   1 1    :              1
(  3)  10 1 1 1 1 1 1 1:                           :                         :
(  4)   7 1 1 1        :                           :                         :
(  5)   9              :1 1 1 1 1 1 1 1 1 1        :                         :
(  6)   2              :1 1 1 1 1 1 1 1            :                         :
(  7)  16              :1 1 1 1 1 1     1   1      :                         :
(  8)  14              :1                     1 1 1:                         :
(  9)   1              :  1 1                      :                         :
( 10)   5              :                           :1 1 1 1 1 1 1 1 1 1 1 1 1:
( 11)   4              :                           :1 1 1 1 1 1 1            :
( 12)  15              :                           :1 1 1 1       1 1 1      :
( 13)  11              :                           :        1                :1 1 1 1 1
( 14)  13              :                           :                         :1 1
( 15)  12              :                           :                         :1   1 1   1 1
( 16)   3              :                    1 1 1  :                         :              1 1
                1                   2                           3                      4
```

Figure 6.3.3

```
FLOW MATRIX AFTER     6 ITERATION(S)

          LOCATIONS
          0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4
          1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
          COMPONENTS
          0 1 2 1 3 3 2 0 3 3 4 3 1 2 4 1 0 0 1 3 0 3 3 1 1 2 0 2 0 2 4 3 4 0 1 1 2 0 2 2 3 1 2
          1 3 5 2 9 1 6 2 7 8 2 2 0 8 0 8 4 7 7 5 6 4 6 9 4 1 5 3 9 9 3 3 1 8 5 6 4 3 7 0 0 1 2
(  1)  10 1 1 1 1 1 1 1
(  2)   7 1 1 1
(  3)   6 1 1   1 1
(  4)   8 1     1   1
(  5)   9               1 1 1 1 1 1 1 1 1 1
(  6)   2               1 1 1 1 1 1 1 1
(  7)  16               1 1 1 1 1 1     1   1
(  8)   6               1 1   1 1     1     1 1   1 1
(  9)   8               1 1 1       1
( 10)  14               1                     1 1 1
( 11)   1                 1   1
( 12)   3                                   1 1 1   1 1
( 13)   5                                               1 1 1 1 1 1 1 1 1 1 1 1 1
( 14)   4                                               1 1 1 1 1 1 1
( 15)  15                                               1 1 1 1       1 1 1
( 16)   8                                               1   1   1 1   1   1 1 1
( 17)   6                                               1 1     1     1 1   1
( 18)  11                                                         *               1 1 1 1 1
( 19)   8                                                                         1 1 1 1   1
( 20)  13                                                                         1 1
( 21)  12                                                                         1   1   1 1 1
```

Figure 6.3.4

Figure 6.3
Illustration of the use of the new relaxation procedure

6.5 INTERACTIVE ROC2 ALGORITHM

In order that the new relaxation procedure could be implemented efficiently, an interactive program is extremely useful, though not absolutely vital. However, an interactive algorithm would allow the analyst to use more information which has largely been left out or cannot be handled directly by any algorithm. The analyst would be able to use his insight and local knowledge to ensure that the suggested groupings are meaningful in the local context.

By implementing ROC2 as an interactive routine, it is possible to utilise our sophisticated visual perception in helping to find a pattern. (It is well known that the human brain has extensive capabilities in searching for and processing even very complicated visual patterns.) By way of an illustration, consider the problem stated by de Witte (1979). The original matrix is shown in Figure 6.4.1. It can be seen that the components could be divided into two groups if machines *F*, *G* and *J* can be duplicated, which is the case in this instance. Figure 6.4.2 shows the grouping after the duplications are carried out. This solution is almost identical to the one derived by de Witte after a labourious process.

	M/Cs	A	B	C	D	E	F	G	H	I	J	K	L
No of	M/Cs	2	1	1	2	1	4	5	1	2	7	3	1
	1	1			1				1	1			
	2	1	1		1		1	1	1				
	3	1	1		1			1	1	1			
	4	1			1			1		1			
	5	1					1	1		1	1		
C	6						1	1	1	1	1		
O	7				1		1		1	1			
M	8		1	1	1	1	1		1	1			
P	9			1	1	1	1		1	1			
O	10			1	1		1		1				
N	11						1						1
E	12							1				1	1
N	13							1			1	1	1
T	14							1			1	1	
S	15										1	1	
	16											1	1
	17							1				1	1
	18						1	1			1		
	19							1			1		

Figure 6.4.1
de Witte's original machine-component matrix

	M/Cs	A	B	C	D	E	F	G	H	I	J	F	G	J	K	L
	1	1			1				1	1						
	2	1	1		1		1	1	1							
	3	1	1		1			1	1	1						
	4	1			1			1		1						
	5	1					1	1		1	1					
C	6						1	1	1	1	1					
O	7				1		1		1	1						
M	8		1	1	1	1	1		1	1						
P	9			1	1	1	1		1	1						
O	10			1	1		1		1							
N	11											1				1
E	12												1		1	1
N	13												1	1	1	1
T	14												1	1	1	
S	15													1	1	
	16														1	1
	17												1		1	1
	18											1	1	1		
	19												1	1		

Figure 6.4.2
de Witte's matrix after duplication process.

The extended ROC2 procedure is implemented as an interactive program with various facilities to rearrange the data in the manner required. It is this mechanism that makes possible the experimentation of alternative mergings and groupings of the kind outlined above, as well as taking account of the various practical constraints in determining an appropriate feasible solution to the problem. The main program can be summarised by the following procedure.

```
IF(start afresh)
    THEN read data from original file
    ELSE read data from continuation file
END IF;
REPEAT {the whole loop}
    IF(information about machines and components required)
        THEN print as much as requested
    END IF;
    REPEAT {interaction}
    CASE
        1: zoom a selected part of the matrix for detailed inspection;
        2: specify exceptional elements;
        3: return exceptional elements to normal status;
        4: specify or remove bottleneck status of machines;
        5: increase the number of machines of specific type;
        6: merge machines of the same type;
    END CASE
    UNTIL(no further action required);
    {end of interaction}
    implement ROC2;
    print current matrix and other data as requested
UNTIL(block diagonal form OR time off to consider next move);
{end of the whole loop}
IF(a final answer)
    THEN print the final matrix and lists of machines and components
    ELSE copy all the data to continuation file
END IF
```

Figure 6.5.1 shows the initial machine-component incidence matrix reported by Burbidge (1973) and resulting from a practical study at Black and Decker Ltd. The extended ROC2 procedure just outlined was applied to this data and the matrix in Figure 6.5.2 was obtained in the ninth iteration of the second trial. The first trial, reaching 23 iterations before being terminated, arrived at a similar result with a higher number of exceptional elements. The objective of these trials was to show that even with a fairly complex matrix such as that shown in Figure 6.5.1, block diagonal structure can still be achieved within moderate limits of computing (approximately 0.25 *CDC Cyber 174* sec per iteration and 20*K* of memory) and human resources. The computations were carried out without specific data about the numbers of the various machine types available, since information of this kind was not published in Burbidge's paper. (Had it been available, it could have been readily incorporated into the analysis.)

The ROC2 algorithm will provide a pure block diagonal form if one exists, in just two iterations. This means that in a very complicated matrix, various trial assignments of the exceptional elements

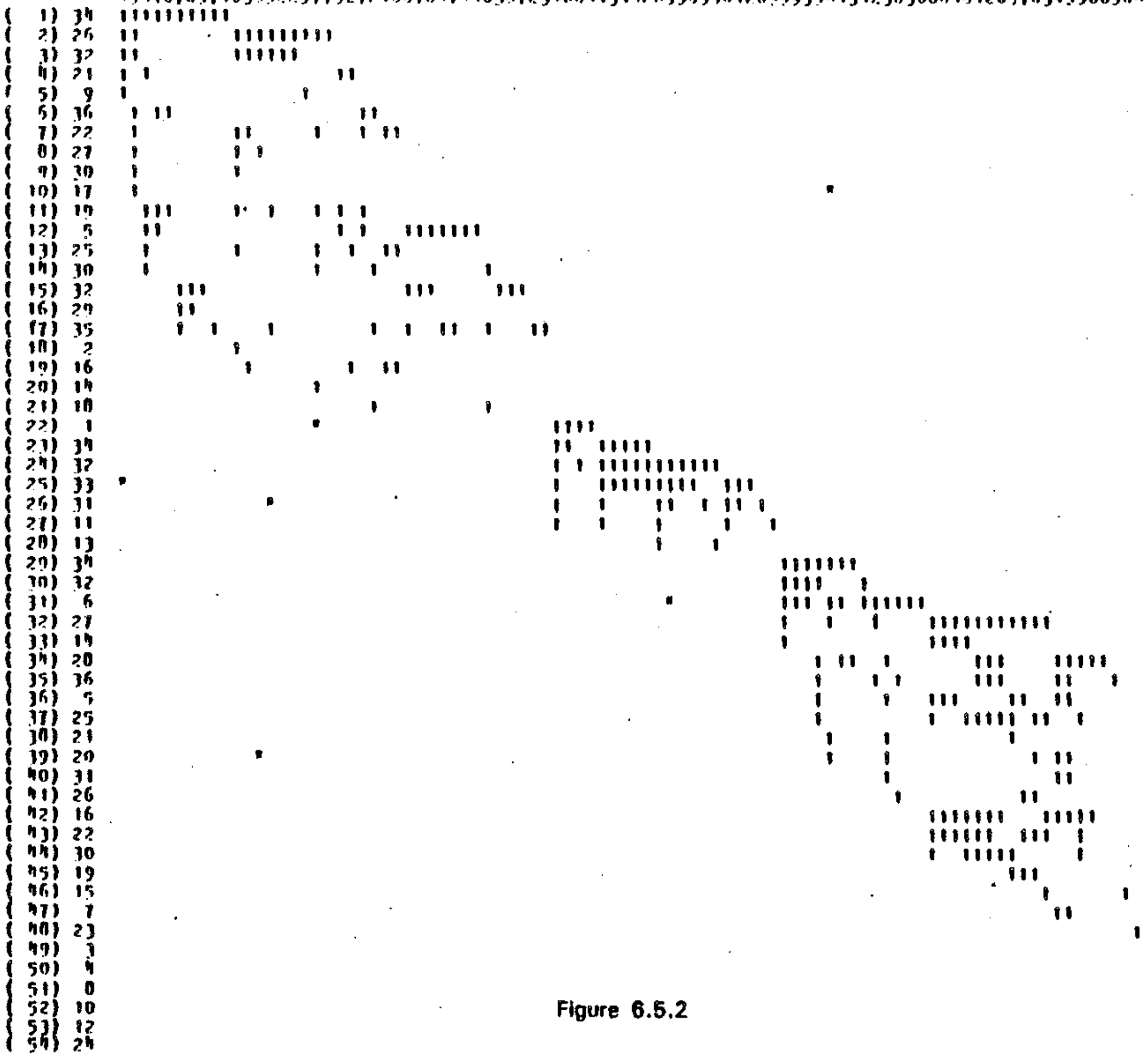
```
      FLOW MATRIX AFTER     0 ITERATION(S)

          LOCATIONS
          000000000111111111122222222223333333333444444444455555555556666666666777777777788888888889
          123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
          COMPONENTS
          000000000111111111122222222223333333333444444444455555555556666666666777777777788888888889
          123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
(  1)  1                 1                 1      1        1                          1
(  2)  2                                                                                           1
(  3)  3
(  4)  4
(  5)  5          1    1         1            1  1   1  1        1 11    1111   1         1 1 1 1 1
(  6)  6     1                  1     1        1    1    11  1                1    1        1  1
(  7)  7                                                                                  1   1
(  8)  8
(  9)  9                                    1                                                1
( 10) 10
( 11) 11            1                1       1  1 1
( 12) 12
( 13) 13                                     1                                      1
( 14) 14                                         1               1       1    1      11
( 15) 15                   1                                                      1
( 16) 16         1      1     11       1         1          1    1     111   1    1  1    1   1
( 17) 17                                                                                1      1
( 18) 18                                                                                 1 1
( 19) 19                                                                1   1     1 1  11     1         1111
( 20) 20                                                                                       1  1 1 11 1
( 21) 21                                                                    1  1  1                 11 1  1
( 22) 22                1     11                 1          1    1      11  11    1  11 1       11 1
( 23) 23                           1
( 24) 24
( 25) 25         11     1     11                 1          1          1  1  1    1  11          111
( 26) 26   1              1                 1      1     1      1       1             111    1  11 1
( 27) 27         1            11                 1           1   1       1    1   1 1 11      11111
( 28) 28         11     1    111       1  1                                        1      1 1 1
( 29) 29                                    1                                1
( 30) 30         1      1     11                 1                        1          11 11 1      11
( 31) 31           11       1        1       1    1       1     1     1                   1 1 1
( 32) 32   111111 1 1 1     1     1 1 1 1    11   111  1  11   11  1 1  1  1  1  1  1  11    1     1
( 33) 33      1     111              1  1    1    1    1  1    1      1          1           1
( 34) 34 1       1 1 1  1        1 1 1 1 1       1 1          1  1  1    1   1111 1    1    1 1
( 35) 35               1            1    1              1     1 1 1        1             1 1
( 36) 36         11           11                      1  1   1              1  11       1 11  1
```

Figure 6.5.1

```
FLOW MATRIX AFTER      9 ITERATION(S)

     LOCATIONS
     000000000111111111122222222223333333333444444444455555555556666666666777777777788888888889
     123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890
     COMPONENTS
     0767726250967504371556056635142400003543531135734041726113624007305042345672208807801132412
     4951070571030522577927210670474403672316041314269569501209993964342303006461289703155005 06.
```



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

Figure 6.5.2

Figure 6.5

Burbidge's problem and an alternative solution

```
          MATRIX AFTER      0 ITERATION(S)

          LOCATIONS
          0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
          1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7
          CONTROL VARIABLES
          0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
          1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7
(  1)  1  3  3     2  2     1  1              2              1           3
(  2)  2  3  3     3        1  1              2        1        1        3
(  3)  3        3  3  1     1  1                 3     1        1
(  4)  4  2  3  3  3                          1  1        1
(  5)  5  2     1     3     2  2  1  1                 3        1     3  1
(  6)  6                 3        2  1                    1        1        3        1  2
(  7)  7  1  1  1     2     3  2  2  3  1     1  1     1                 1
(  8)  8  1  1  1     2     2  3  2  2                 1
(  9)  9              1  2  2  2  3  3
( 10) 10              1  1  3  2  3  3  1  1  2  2              1
( 11) 11                    2        1  3  3                       1              1
( 12) 12                             1  3  3  1  1              1
( 13) 13  2  2     1        1        2     1  3        1        1        2
( 14) 14        3  1        1        2     1     3     1        1
( 15) 15                                            3  1  1  1        3  1              1
( 16) 16     1  1     3     1  1              1  1  1  3  1  1  1     3  2
( 17) 17           1     1                          1  1  3        1  1     1           3
( 18) 18  1                                         1  1     3           1        2
( 19) 19     1  1     1              1     1  1  1     1        3        1
( 20) 20                 1              1                 1        3  2        3     1
( 21) 21              3                             3  3  1        2  3        1     1  1
( 22) 22  3  3        1     1                 2     1  2     1  1        3
( 23) 23                 3                                1                 3           1
( 24) 24                                                           3  1        3     2  1
( 25) 25                                1                    2                    3     3
( 26) 26                 1                                         1  1        2     3
( 27) 27                 2                          1     3           1     1  1  3     3
```

LOCATIONS

CONTROL VARIABLES

Figure 6.6.1

```
RUN NUMBER         1


        MATRIX AFTER     4 ITERATION(S)

        LOCATIONS
          0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
          1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7
        CONTROL VARIABLES
          2  1  0  1  2  0  2  0  0  0  1  1  0  0  1  0  1  1  1  2  0  2  1  2  2  2  1
          1  6  5  5  6  2  2  1  4  3  4  0  9  7  3  8  9  1  2  3  6  7  7  5  4  0  8
LOCATIONS CONTROL VARIABLES
(  1) 21  3  3  3  3  1                                                  1  1     1  2
(  2) 16  3  3  3  1     1  2        1  1        1  1  1  1                 1           1
(  3)  5  3  3  3           1  2     1     1  1  2     2  1
(  4) 15  3  1     3        1                                            1  1           1
(  5) 26  1           3                                               1           2  1
(  6)  2     1           3  3  3  3              1  2  1  1
(  7) 22     2  1  1     3  3  3                 1  2     1                             1
(  8)  1        2        3  3  3  2              1  2  1                                1
(  9)  4                 3     2  3  3  1           1                       1
( 10)  3     1  1                 3  3  3        1     1  1
( 11) 14     1                    1  3  3  2     1        1     1
( 12) 10        1                       2  3  3  3  2  2  1  1  1     1
( 13)  9        1                          3  3  2     2              2
( 14)  7     1  2        1  1  1     1  1  3  2  3  1  2     1
( 15) 13     1           2  2  2  1        2     1  3     1     1
( 16)  8     1  2        1     1     1     2  2  2     3
( 17) 19     1  1        1  1        1  1  1        1     3     1
( 18) 11                                   1     2           3  3              1     1
( 19) 12                                1  1        1     1  3  3
( 20) 23                                                           3  3  1  1
( 21)  6              1                    1  2                    3  3  2  1        1
( 22) 27  1        1                                               1  2  3  3  3  1
( 23) 17  1  1     1              1                                1  1  3  3        1
( 24) 25                                                     1           3     3        2
( 25) 24  1           2                                                  1        3  3
( 26) 20  2           1                                      1        1     1     3  3
( 27) 18     1     1        1  1                                               2        3
```

Figure 6.6.2

```
         MATRIX AFTER      4 ITERATION(S)

         LOCATIONS
           0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
           1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7
         CONTROL VARIABLES
           0  0  2  0  0  1  0  1  1  2  0  0  0  1  1  1  1  1  0  2  1  2  2  2  2  2  1
           1  2  2  3  4  4  5  5  6  1  8  7  9  0  9  3  1  2  6  3  7  7  5  6  0  4  8
(  1)  1   9  9  9  8  8  8
(  2)  2   9  9  9  8  8  8
(  3) 22   9  9  9  8  8  8
(  4)  3   8  8  8  9  9  9
(  5)  4   8  8  8  9  9  9
(  6) 14   8  8  8  9  9  9
(  7)  5                     9  9  9  9  3
(  8) 15                     9  9  9  9  3
(  9) 16                     9  9  9  9  3
( 10) 21                     9  9  9  9  3
( 11)  8                     3  3  3  3  9  4  4  4     3
( 12)  7                                 4  9  9  9  4  3  3  3
( 13)  9                                 4  9  9  9  4  3  3  3
( 14) 10                                 4  9  9  9  4  3  3  3
( 15) 19                                    4  4  4  9  3
( 16) 13                                 3  3  3  3  3  9
( 17) 11                                    3  3  3        9  9
( 18) 12                                    3  3  3        9  9
( 19)  6                                                         9  9  4  4  3
( 20) 23                                                         9  9  4  4  3
( 21) 17                                                         4  4  9  9  8  3
( 22) 27                                                         4  4  9  9  8  3
( 23) 25                                                         3  3  8  8  9  3  3  3
( 24) 26                                                               3  3  3  9
( 25) 20                                                                     3     9  9
( 26) 24                                                                     3     9  9
( 27) 18                                                                                 9
```

LOCATIONS

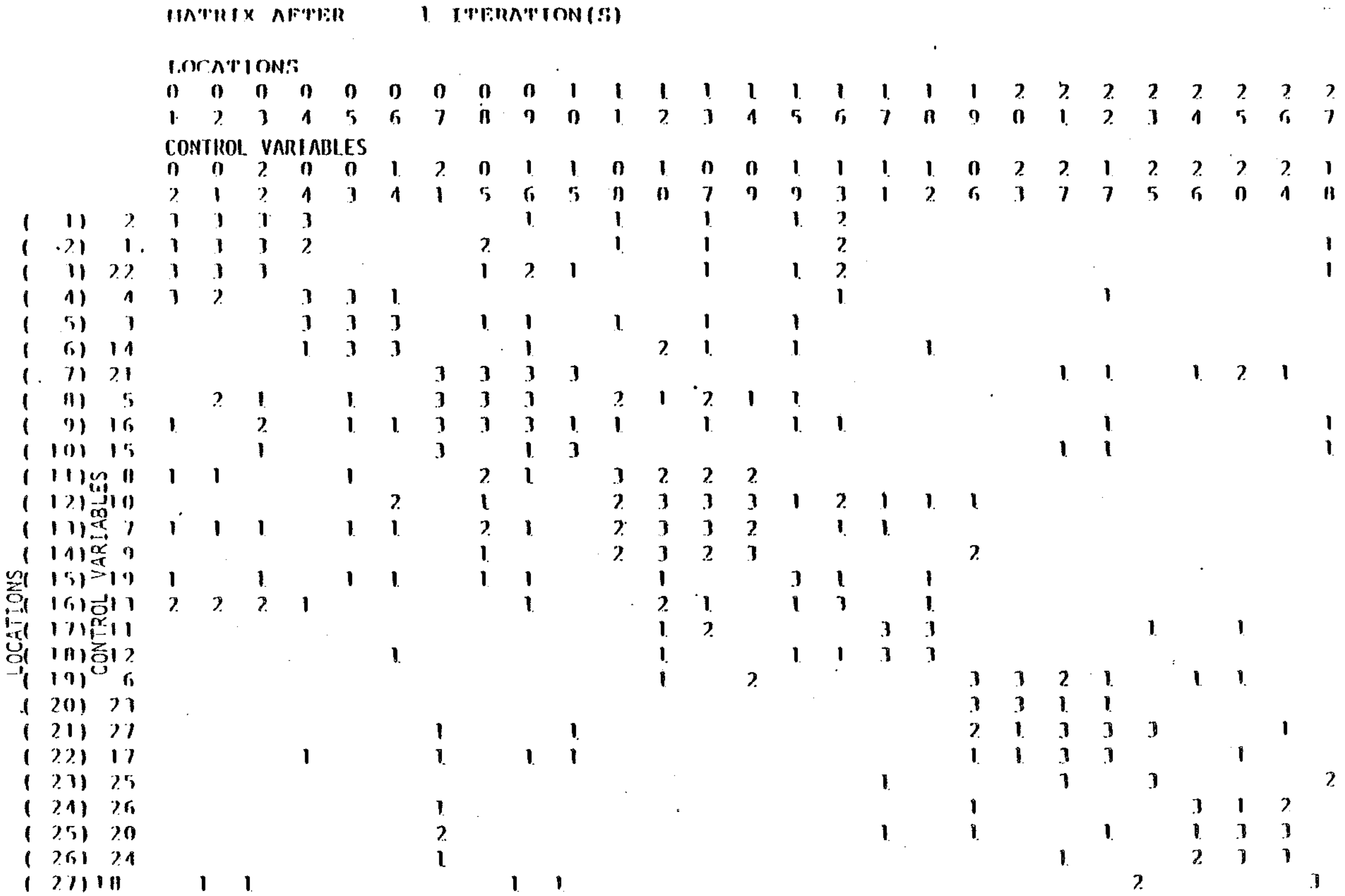
CONTROL VARIABLES

Figure 6.6.3

```
           MATRIX AFTER      0 ITERATION(S)

           LOCATIONS
           0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1  2  2  2  2  2  2  2  2
           1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5  6  7
           CONTROL VARIABLES
           0  0  2  0  0  1  0  1  1  2  0  0  0  1  1  1  1  1  0  2  1  2  2  2  2  2  1
           1  2  2  3  4  4  5  5  6  1  8  7  9  0  9  3  1  2  6  3  7  7  5  6  0  4  8
LOCATIONS  CONTROL VARIABLES
(  1)   1  3  3  3     2     2           1  1           2                                1
(  2)   2  3  3  3     3           1     1  1        1  2
(  3)  22  3  3  3           1  1  2        1        1  2                                1
(  4)   3           3  3  3  1     1     1  1        1
(  5)   4  2  3     3  3  1                             1              1
(  6)  14           3  1  3        1        1     2  1        1
(  7)   5  2     1  1        3     3  3  2  2  1  1  1
(  8)  15        1              3  1  3                                1  1              1
(  9)  16     1  2  1     1  3  1  3  3  1  1        1  1              1                 1
( 10)  21                    3  3  3  3                                1  1     1  2  1
( 11)   8  1  1     1        2     1     3  2  2  2
( 12)   7  1  1  1  1     1  2     1     2  3  2  3     1  1
( 13)   9                    1           2  2  3  3              2
( 14)  10                 2  1           2  3  3  3  1  2  1  1  1
( 15)  19     1  1  1     1  1     1              1  3  1     1
( 16)  13  2  2  2     1           1        1     2  1  3     1
( 17)  11                                   2     1        3  3              1     1
( 18)  12                 1                       1  1  1  3  3
( 19)   6                                      2  1              3  3  1  2     1  1
( 20)  23                                                        3  3  1  1
( 21)  17              1        1  1  1                          1  1  3  3        1
( 22)  27                       1     1                          2  1  3  3  3        1
( 23)  25                                               1              3  3           2
( 24)  26                             1                          1              3  1  2
( 25)  20                             2                    1     1     1        1  3  3
( 26)  24                             1                                   1     2  3  3
( 27)  18  1     1              1                                            2           3
```

Figure 6.6.4



```
          MATRIX AFTER       1 ITERATION(S)

          LOCATIONS
          0   0   0   0   0   0   0   0   0   1   1   1   1   1   1   1   1   1   1   2   2   2   2   2   2   2   2
          1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   6   7   8   9   0   1   2   3   4   5   6   7
          CONTROL VARIABLES
          0   0   2   0   0   1   2   0   1   1   0   1   0   0   1   1   1   1   0   2   2   1   2   2   2   2   1
          2   1   2   4   3   4   1   5   6   5   8   0   7   9   9   3   1   2   6   3   7   7   5   6   0   4   8
(  1)   2 3   3   3   3                   1       1       1       1   2
(  2)   1 3   3   3   2               2           1       1           2                                           1
(  3)  22 3   3   3                   1   2   1           1       1   2                                           1
(  4)   4 3   2       3   3   1                                       1                       1
(  5)   3             3   3   3       1   1       1       1       1
(  6)  14             1   3   3           1           2   1       1           1
(  7)  21                         3   3   3   3                                           1   1       1   2   1
(  8)   5     2   1       1       3   3   3       2   1   2   1   1
(  9)  16 1       2       1   1   3   3   3   1   1       1       1   1                       1                   1
( 10)  15         1               3       1   3                                           1   1                   1
( 11)   8 1   1           1           2   1       3   2   2   2
( 12)  10                     2       1           2   3   3   3   1   2   1   1   1
( 13)   7 1   1   1       1   1       2   1       2   3   3   2       1   1
( 14)   9                             1           2   3   2   3                   2
( 15)  19 1       1       1   1       1   1           1           3   1       1
( 16)  13 2   2   2   1                   1           2   1       1   3       1
( 17)  11                                             1   2               3   3                   1       1
( 18)  12                     1                       1           1   1   3   3
( 19)   6                                             1       2                   3   3   2   1       1   1
( 20)  23                                                                         3   3   1   1
( 21)  27                         1           1                                   2   1   3   3   3           1
( 22)  17             1           1       1   1                                   1   1   3   3           1
( 23)  25                                                             1               3       3               2
( 24)  26                         1                                               1                   3   1   2
( 25)  20                         2                                       1       1           1       1   3   3
( 26)  24                         1                                                       1           2   3   3
( 27)  18     1   1                       1   1                                                   2               3
```

LOCATIONS

CONTROL VARIABLES

Figure 6.6.5

Figure 6.6

An airport design problem

and transfers of components between machines of the same type can be made and the results of the effects can be quickly determined within two iterations. If the outcome is not as expected or desired, a quick return to the previous stage can be achieved, followed by another trial run. This interactive approach, and the ability of the ROC algorithm quickly to pick out any emerging pattern, allows the designer to experiment with various alternatives. It also allows the designer to take account, during the process of interaction, of other factors, some of which may be neither quantifiable nor easy to formulate in a very precise manner.

6.6 OTHER APPLICATIONS OF THE ROC2 ALGORITHM

There are many other situations in which the use of the ROC2 algorithm is also appropriate. In loading components for a highly sophisticated numerically controlled machine, where the changing time of the tools for various operations become significant, the ROC2 algorithm has been used to group the tools and the components appropriately. By loading the components of the same group in sequence, the amount of tool changing time can be significantly reduced, without having to resort to more complicated techniques. This problem is solved in less than 2 *Cyber 174* seconds. An earlier attempt to solve it using the SLCA required so much computing time that the job could only be run at the weekend, and even then failed to provide any clear grouping. The use of SLCA also requires access to a graph plotter.

The ROC2 algorithm can be used in the case of non 0-1 matrices by sorting the entries in accordance with their values during the shifting process of the radix procedure. The airport design problem of McCormick *et al* (1972) is used as an example to illustrate the procedure. The initial matrix is shown in Figure 6.6.1 in which the machines and components of the production problem are replaced by airport design variables that are under the control of the designers. The degree of dependency between the variables is designated as nil, weak, moderate or strong and represented in the matrix by the value 0, 1, 2 and 3 respectively. The problem as outlined by McCormick *et al* reduces to that of determining a decomposition of the matrix elements into groups with minimal interdependency. This is equivalent to the creation of a block diagonal clustering if possible.

A straightforward application of the ROC2 algorithm does not highlight the relationships between the control variables adequately. However if the matrix is further processed using only entries higher than 1, clearer relationships begin to emerge. It is also possible to experiment further by considering only the strong elements of value 3 (Figure 6.6.2). As the grouping of the control variables may be affected by the starting condition, nine random starting solutions were generated. The ROC2 algorithm was applied to the 3 entries. Figure 6.6.3 shows the numbers of times particular pairs of variables were found within the same group. (Frequencies less than three out of nine are deleted for clarity). In most cases, stable relationships emerge. The few elements that are unstable may be assigned to the block in which they most frequently appear.

Although the final matrix using the ROC2 algorithm (Figure 6.6.4) may not look as neat as the solution generated by McCormick *et al* (Figure 6.6.5), the final groupings are very similar. The ROC2 algorithm does not require the data to be metric, (they obviously are not in the case of the airport design problem); it provides an approach for grouping ordinal data as no objective function is required.

Grigoriadis (1980) suggests that most large scale LP problems can be formulated or permuted into a block diagonal structure with a few connecting rows and columns. The bottleneck machines example shows how such connecting rows can be identified. The same procedure applied to the columns will identify the connecting columns. The ROC2 algorithm can also be used to investigate the possible partitioning of the set covering problem (Hey 1980). The preliminary result of an investigation into the use of the ROC2 algorithm in conjunction with the State Space Relaxation method to solve the Set Covering Problem was encouraging. A problem which could not be solved in less than 35 *Cyber 174* seconds, was solved in less than 5 seconds using the partition generated by the ROC2 algorithm. The lower bounds generated by partitionings using the ROC2 algorithm also appear to have higher values than those generated by random partitioning (Paixao 1982).

6.7 CONCLUSIONS

A practical solution to the problem of machine-component group formation requires a compromise between an objective, explicit and repeatable algorithm on the one hand, and the flexibility of *ad hoc* facilities to cater for specific considerations or constraints on the other hand. Similarity coefficient methods are perhaps more explicit and hence more repeatable than most, but there is still much more work to be done both on the sensitivity aspects of the various weightings that have been advocated, and on the development of an efficient method for selecting one specific set of clusters out of all the possible ones which can be generated. Evaluation methods *per se* are useful in smaller problems. The method advocated in this chapter has an explicit and repeatable algorithm (ROC2) and provides interactive procedures for *ad hoc* treatments. As described here, the method does not explicitly include other considerations such as machine capacity constraints; these can however, be incorporated quite easily within the existing data structure.

It would be unrealistic to hope that procedures such as the ROC2 algorithm will overcome all the difficulties associated with machine-component group formation. This problem can be relaxed into a well known Graph Theory problem called *minimum k-connected*, with extra constraints. The basic minimum *k*-connected problem alone is NP-complete (Garey & Johnson 1979, *GT31*), which implies that it has no known polynomial-time algorithm. The determination of a grouping of machines and components that would minimise the total material handling costs between cells would constitute an even harder problem. For the moment, therefore, we must be content with procedures which provide us with a *good* feasible solution and allow us to concentrate on more complicated and not easily quantifiable issues in an *ad hoc* and interactive manner.

As far as using the ROC2 algorithm as a clustering method is concerned, the main advantages are that very few assumptions are made concerning the nature of the data. Another feature is that there is no necessity for a prior specification of the number of clusters required. The ROC2 algorithm is also neither a hierarchical nor an optimizing procedure. As the algorithm is very fast and no loss of information of any kind results from the processing, it is ideally suited to exploratory data analysis or data reduction on a large set of input, where other methods (such as the Bond Energy Method of McCormick *et al*) may necessitate an unacceptable amount of computing time.

7 Sequence-Dependent Setup Time Scheduling Problems

7.1 INTRODUCTION

Sequence-dependent setup time scheduling problems (SDSTSPs) are commonly found among the cases where single facilities are used in the manufacture of several products. This is more pronounced in the process industry where some amount of cleaning may be required between the production of various batches, such as in the making of paints and detergents. Other examples can be found among the usages of automated multi-purpose machinery, where the setup time between various jobs can be very expensive, or in certain assembly lines where retooling and rearrangement of work stations represent the setup activity. In practice, even though many scheduling problems are strictly sequence dependent in their setup times, it is only beneficial to consider the problems as such if the setup constraints are a predominant factor, either in absolute terms or relative to the operational cost (time).

7.2 THE TRAVELLING SALESMAN PROBLEM

The SDSTSP can be formulated as an *asymmetric travelling salesman problem* (ATSP). The travelling salesman problem (TSP) is one of the most studied combinatorial problems, since many problems that arise in practical situations involving sequencing and routing can be formulated as TSPs. The TSP can be described as: given an *n* by *n* distance matrix between *n* cities, find a minimum length circuit that passes through each city once and only once. The problem can be formalized as:

$$\text{Minimize } \sum_{i\in N} \sum_{j\in N} c_{ij}x_{ij} \tag{7.1}$$

$$\text{subject to } \sum_{i\in N} x_{ij} = 1 \tag{7.2}$$

$$\sum_{j\in N} x_{ij} = 1 \tag{7.3}$$

$$x_{ij} = 1 \text{ if arc} ij \text{ is in the tour; } x_{ij} = 0 \text{ otherwise} \tag{7.4}$$

$$x_{ij} \text{ must form a tour} \tag{7.5}$$

There are various ways to express the constraint (7.5) explicitly (Gavish & Graves 1979). It is, however, easy to implement a subtour elimination procedure in a heuristic and hence constraint (7.5) will not be elaborated.

7.3 SOME THEORETICAL CONSIDERATIONS FOR THE TRAVELLING SALESMAN PROBLEM

The TSP, like certain problems investigated in this thesis, is an NP-complete problem (Garey *et al*, 1976). It is, however, easier than the problems considered in earlier chapters, as the size of TSP problems that can be solved in a reasonable time is considerably larger. This is achieved by imposing certain restrictions on the distance matrix. The two main restrictions are that the matrix is symmetric and that the distances are Euclidean. The symmetric property reduces the solution spaces by half. The Euclidean constraint, also known as the triangularity constraint, implies that for any *i j* and *k* the following condition holds true:

$$c_{ik} + c_{kj} \geq c_{ij} \tag{7.6}$$

This constraint provides many useful properties which can be used in the search for the solution. One of the more important ones is that the order of vertices in the convex hull of the distance matrix is the same order in which these vertices appear in the optimal tour (Gonzales, 1962).

In the case of the SDSTSP, the distance matrix is usually not symmetric and more importantly the distances are quite often non-Euclidean. The asymmetric matrix increases the solution spaces by 100% over the symmetric case. The non-Euclidean property implies that no heuristic can be guaranteed to provide a solution within a fixed bound. It is generally recognised that the non-Euclidean TSPs are significantly more difficult than their Euclidean cousins (Papadimitriou & Steiglitz, 1978).

7.4 LITERATURE SURVEY

The majority of the papers dealing with the TSP are confined to symmetric Euclidean distances. Some of the techniques described in these papers can be applied directly or with minor modifications to the asymmetric and non-Euclidean cases. The approach of using various Linear Programming relaxations (*eg* Crowder & Padberg, 1980; Miliotis, 1976) will not be discussed as this necessitates access to an efficient LP package. Furthermore, the approach is not competitive with other branch and bound methods for the asymmetric case (Christofides, 1979).

An optimal procedure for TSPs is generally based on a relaxation of the original TSP problem either into a *shortest spanning tree* (SST) problem or into an *assignment problem* (AP). The examples of the earlier approach were suggested by Held & Karp (1970, 1971) and Hansen & Krarup (1974). The underlying idea of the SST relaxation is that, if a vertex and its two associated arcs are removed from a tour, the remaining arcs form a spanning tree. Hence the cost of the shortest spanning tree together with the two shortest arcs associated with the removed vertex provides a lower bound for the TSP. By using the Lagrangean relaxation technique, the bounds can be updated until all but two of the vertices of the spanning tree have degree 2. At this stage a feasible solution is found. The AP relaxation is intuitively related to the TSP since the AP is the TSP without the constraint (7.5). The solution is obtained by successively solving the problem as an AP with penalty functions associated with the violations of the constraint (7.5). Recent results suggest that the AP relaxations are more useful in the asymmetric case than other forms of relaxation (Carpaneto & Toth, 1980; Balas & Christofides, 1981).

Heuristic approaches to the asymmetric TSP can be divided into two classes; construction heuristics and improvement heuristics. The construction heuristics can be divided further into two subclasses; tour building and tour patching methods. A tour building method iteratively selects a small number of arcs, usually one, by a certain set of criteria until a tour is formed. A typical example is the nearest unvisited city heuristic (Eilon *et al*, 1971). In this heuristic, an arc is selected if it forms the shortest arc to an assigned city without creating a subtour. Van Der Cryssen-Rijckaert (1978) heuristic is based on a concept of shadow cost, namely a potential loss if an arc is not assigned at a particular stage of the iteration. A shadow cost heuristic will select the arc with the highest associated shadow cost for an assignment. Both heuristics have the time complexity of $O(n^2)$, and in both cases when an arc is assigned it remains part of the tour permanently. In a tour insertion heuristic, an assigned arc can be removed in a subsequent iteration. Given a starting point, a subtour is created by iteratively inserting a node into the subtour according to a set of criteria, until all the nodes are included and a feasible tour is formed. The time complexity of a tour insertion procedure is $O(n^3)$. The criterion often used in the tour insertion heuristic is the minimization of the increase in the subtour cost.

A tour patching heuristic solves a relaxed problem in the same manner as the optimum procedures. The difference is that the relaxed problem is solved only once in a patching heuristic. If the solution is a feasible tour, then the optimum solution is achieved. More often, the solution is not feasible, and ways have to be found to change the solution into a feasible one. Alk (1980) suggested a heuristic based on the SST relaxation where the patching algorithm is carried out by solving an associated transportation problem. Karp's (1979) heuristic is based on the AP relaxation, and the subtour elimination is also formulated as another assignment problem.

Improvement heuristics for the asymmetric case are largely extensions of the approaches adopted for the symmetric case (Kanelakis & Papadimitriou, 1980). These include the variable depth search and *n-opt* heuristics.

The only paper found on the interactive approach to TSP problem is by Krolak *et al* (1971). It is a cumbersome manual implementation involving intensive human effort in the interpretation of the intermediate solutions in a graphical manner. The visual aspect of the implementation limits the sizes of the problems to relatively small ones. The non-Euclidean distances would reduce the potential benefit of visual interaction even further. It is unlikely that interaction with the TSP in this manner would be beneficial.

7.5 A FRAMEWORK FOR EMPIRICAL STUDIES OF SOME HEURISTICS

One of the results of the Euclidean restriction is that the worst case behaviours of many heuristics can be analysed in advance. For example, the nearest neighbour heuristic is guaranteed to produce a tour within a factor of *log*(*n*) of the optimal value in the symmetric case (Rosenkantz *et al*, 1977) and within a factor of $n/2$ in the asymmetric case (Frieze *et al*, 1982). In the non-Euclidean case, it cannot be so analysed. To illustrate the difficulty, consider a transformation of a non-Euclidean distance matrix to satisfy the triangularity constraint by adding a number M, which may be arbitarily large, to all distances. This would lead to the overall increase of the final tour length by nM. Hence, the bound guaranteed by the nearest neighbour routine is $log(n)(nM +$ previous optimum). Since M may be arbitarily large, there can be no effective guarantee of the bound. Performances of various heuristics can only be compared empirically.

Four construction heuristics are studied. The first is based on the bounding calculations suggested by Little *et al* (1963). Although the bounds calculated are not as tight as the ones generated by the use of AP or SST relaxation, Little's method always considers only feasible solutions and hence does not require further patching procedure, as is the case of AP or SST relaxation. The heuristic can be summarised as follows:

```
REPEAT
   for every row $i$, reduce cost $c_{ij}$ by $c_{i.}$,
      where $c_{i.}$ is the minimum of row $i$;
   for every column $j$, reduce cost $c_{ij}$ by $c_{.j}$,
      where $c_{.j}$ is the minimum of column $j$;
   for every $c_{ij} = 0$, calculate the increase in the
      lower bound $b_{ij} = p(i) + q(j)$,
      where
         $p(i) = \min c_{ik} \quad k \neq i$,
         $q(i) = \min c_{kj} \quad k \neq j$;
   assign arc $a_{ij}$ to the solution for the maximum $b_{ij}$;
   update the matrix to prevent subtour formation;
UNTIL a tour is assigned
```

The value of b_{ij} is the potential increase of the lower bound of the TSP if the arc a_{ij} is excluded from the tour (Little *et al*, 1963). At any stage of the iteration, an arc is included if its exclusion results in the highest increase of the lower bound, b_{ij}. The second heuristic tested is the standard nearest unvisited city adapted for the asymmetric case. The third heuristic is based on a shadow cost method and the final one is the nearest tour insertion heuristic.

A shadow cost of an arc can be defined in many different ways. In this chapter, two definitions of shadow costs are studied. The more comprehensive one, to be called *shadow1*, is similar to the one suggested by Van Der Cryssen & Rijckaert (1978). The second definition, *shadow2*, takes a simplistic approach. In the shadow1 definition, the shadow cost of an arc is defined as the difference between the cost of the best local assignment if the arc is excluded from consideration, and the best local assignment if the arc is included. A local assignment is an allocation of an arc entering or leaving a node if the node has already been assigned as leaving or entering the node respectively. In the case where no arc has been assigned to the node, the combined cost of arc entering and leaving the node will be considered in the calculation of the shadow cost. In the Van Der Cryssen-Rijckaert heuristic, the shadow cost is not used in a consistent manner. This leads to some different assignment criteria to the ones used in the shadow1 heuristic. Some of these differences will be indicated in the next section.

7.5.1 Shadow1 Heuristic for the Asymmetric Travelling Salesman Problem

A shadow cost heuristic essentially considers assigning an arc if a penalty associated with the alternative assignment is highest. In order that the discussion regarding a local arrangement can be conveniently carried out, the following notations are adopted:

i: node under consideration;

x_1, x_2, x_3: the shortest, the second shortest
and the third shortest arcs into node i respectively;

y_1, y_2, y_3: the shortest, the second shortest
and the third shortest arcs leaving node i respectively;

$TX1$, $TX2$, $TX3$: the nodes associated with the three shortest
arcs into node i such that $c(TX1, i) = x_1$,
$c(TX2, i) = x_2$, and $c(TX3, i) = x_3$;

$TY1$, $TY2$, $TY3$: the nodes associated with the three shortest
arcs leaving node i such that $c(i, TY1) = y_1$,
$c(i, TY2) = y_2$, and $c(i, TY3) = y_3$;

A representation of the above description is shown in Figure 7.1.3. It should be noted that x_3 and y_3 are not represented in the following diagrams as their relative locations do not affect the shadow cost consideration.

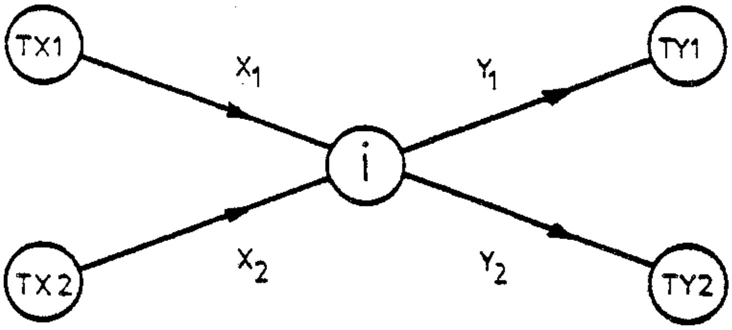


Figure 7.1.1
Case 1 of an active node under consideration

In a shadow cost heuristic, an arc is assigned at each iteration by considering all the nodes. A node can be in one of the following states: A node is nonactive when an arc entering and an arc leaving the node have already been assigned. A node is partially active if an arc entering or leaving the node is assigned. Finally, a node is active is there is no assigned arc entering or leaving the node. If a node is nonactive, it is not processed. If the node is partially active and the arc leaving the node has already been assigned, the shadow cost of the arc (*TX*1, *i*) is x_2 - x_1. Similarly the shadow cost of the arc(*i*, *TY*1) is y_2 - y_1 when the arc entering node *i* has already been assigned. In the case of a fully active node, there are seven possible configurations regarding the locations of nodes *TX*1, *TX*2, *TY*1 and *TY*2. The first and second configurations are shown in Figures 7.1.1-7.1.2.

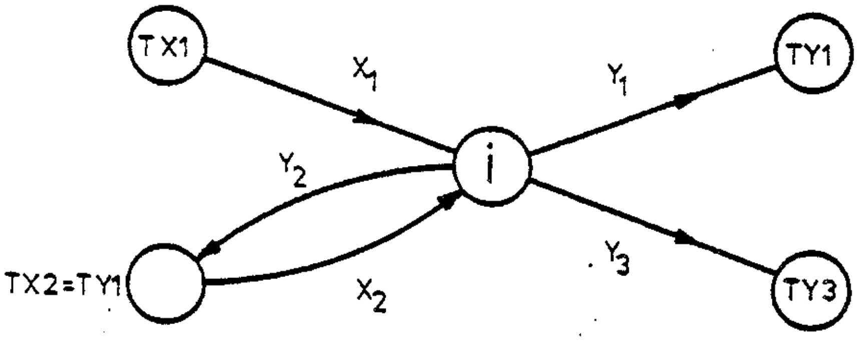


Figure 7.1.2
Case 2 of an active node under consideration

It will be seen that in cases 1 to 5 the cheapest pair of incident arcs of a node are arcs (*TX*1, *i*) and (*i*, *TY*1), for a cost of $x_1 + y_1$. In both cases 1 and 2 the least cost combination excluding the arc (*TX*1, *i*) is arc (*TX*2, *i*) and (*i*, *TY*1) at the cost of $x_2 + y_1$. Hence the shadow cost of arc (*TX*1, *i*) is x_2 - x_1. Similarly it can be shown that the shadow cost of arc (*i*, *TY*1) is y_2 - y_1. The shadow cost with respect to node i is

$$\text{Max}(x_2 - x_1,\ y_2 - y_1) \tag{7.7}$$

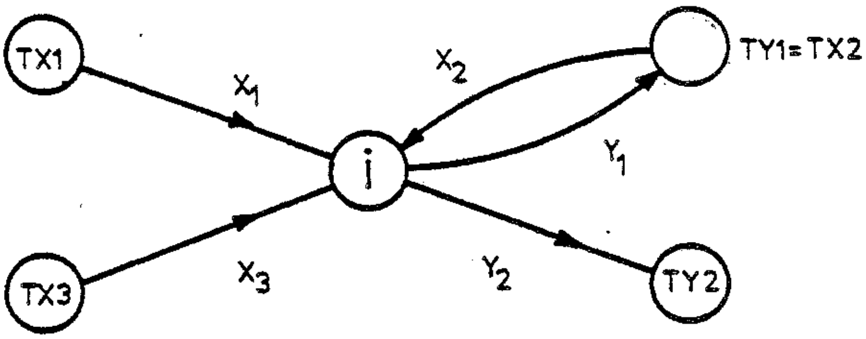


Figure 7.1.3
Case 3 of an active node under consideration

In case 3, if the arc ($TX1$, i) is excluded, there are two possible candidates for the least cost combinations; arc ($TX2$, i) together with arc (i, $TY2$), or arc ($TX3$, i) together with arc (i, $TY1$). (It should be noted that Van Der Cryssen-Rijckaert heuristic only considers the latter combination). The shadow cost of the arc ($TX1$, i) is

$$\text{Min}\ [(x_2 + y_2) - (x_1 + y_1),\ x_3 - x_1]$$

The shadow cost of the arc (i, $TY1$) is the same as in cases 1 and 2. The shadow cost with respect to node i in case 3 is

$$\text{Max}\ [\ \text{Min}((x_2 + y_2) - (x_1 + y_1),\ x_3 - x_1),\ y_2 - y_1] \tag{7.8}$$

Similarly, it can be shown that the shadow cost in case 4 is

$$\text{Max}\ [\ x_2 - x_1,\ \ \text{Min}((x_2 + y_2) - (x_1 + y_1),\ y_3 - y_1)] \tag{7.9}$$

and the shadow cost in case 5 is

$$\text{Max}\ [\ \text{Min}((x_2 + y_2) - (x_1 + y_1),\ x_3 - x_1),\ \ \text{Min}((x_2 + y_2) - (x_1 + y_1),\ y_3 - y_1)] \tag{7.10}$$

In cases 6 and 7, Figures 7.1.6-7.1.7, there are two main candidates, namely arc ($TX1$, i) together with arc (i, $TY2$) or arc($TX2$, i) together with arc (i, $TY1$). The shadow cost is

$$\text{Abs}[(x_1 + y_2) - (x_2 + y_1)] \tag{7.11}$$

7.5.2 Shadow2 Heuristic for the Asymmetric Travelling Salesman Problem

The shadow2 heuristic is a simplified version of the shadow1 procedure. In the case of the partially active nodes, the shadow cost calculations are exactly the same. In the case of the active nodes the shadow cost function is the same as the cases 1 and 2 of the shadow1 heuristic. Both shadow

Figure 7.1.4

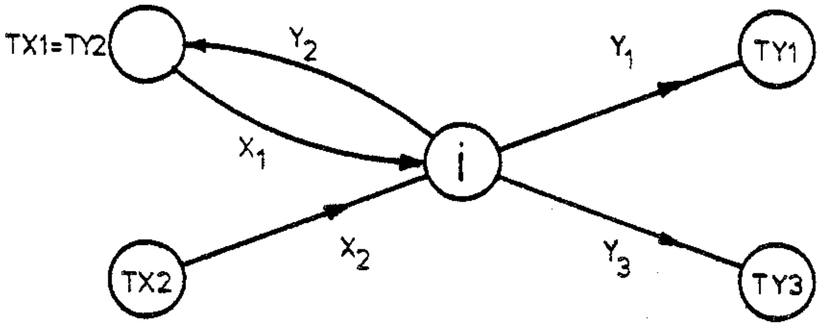


Figure 7.1.5

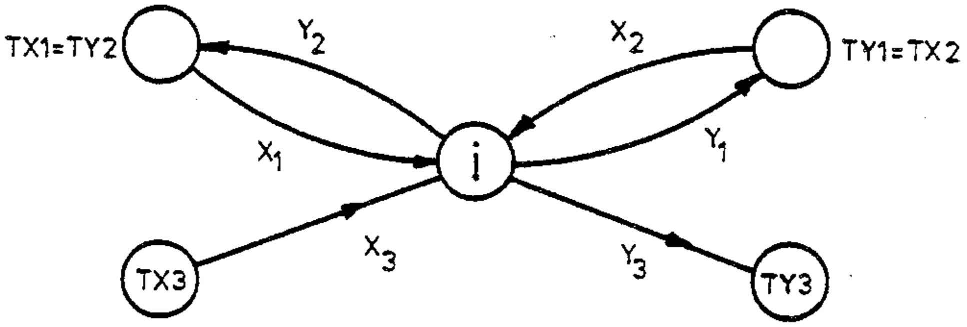


Figure 7.1.6

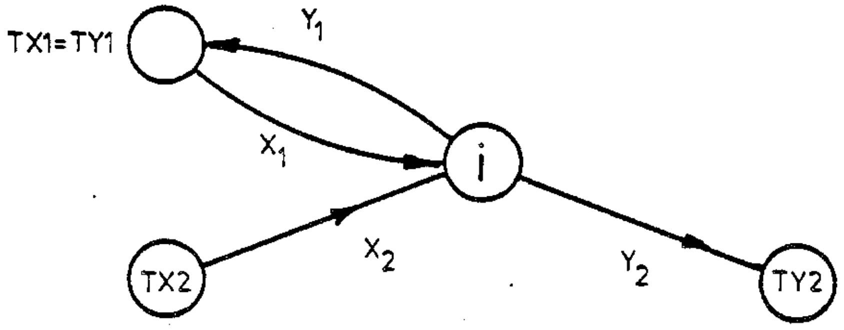


Figure 7.1.7

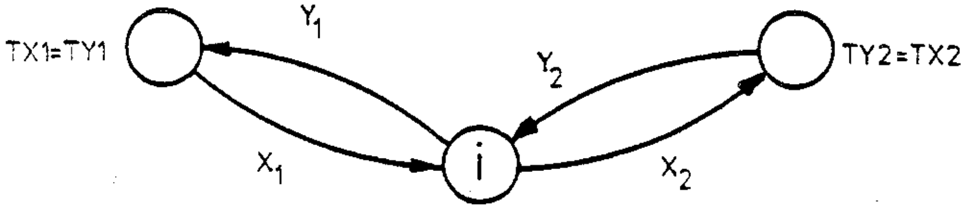


Figure 7.1
Cases of active nodes under consideration

cost heuristics can be summarised as:

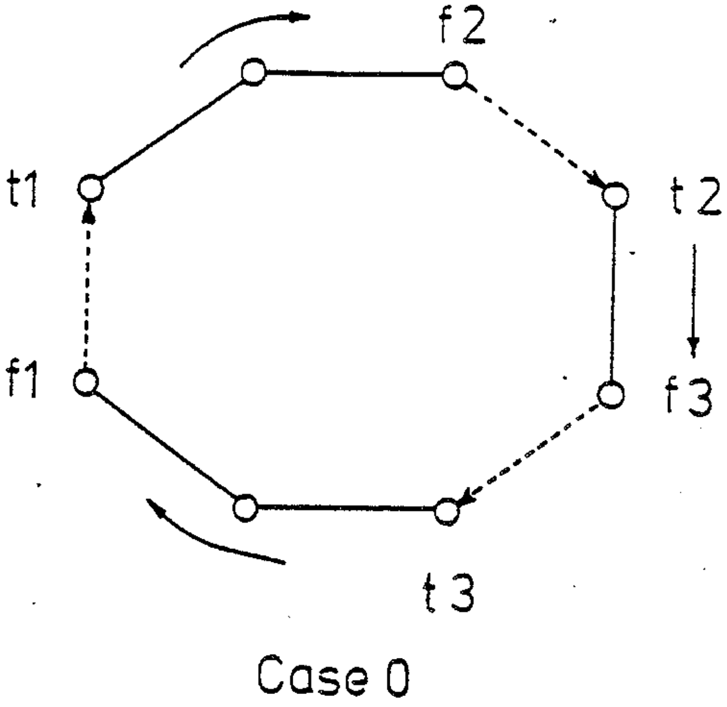
```
REPEAT
    FOR $i$ = 1 TO $n$ DO calculate the shadow cost;
    select the arc with the highest shadow cost;
    assign the arc and update the matrix;
UNTIL a tour is formed;
```

7.5.3 Implementations of 3-Opt and 4-Opt Improvement Heuristics

Improvement heuristics considered in this chapter are limited to the *3-opt* and *4-opt* versions for the asymmetric case only. An *n-opt* improvement heuristic considers removing *n* existing arcs, to be replaced by *n* new ones. The *3-opt* heuristic for the symmetric case involves seven extra alternatives (Eilon *et al*, 1971). In the asymmetric case, there is only one extra option as shown in Figure 7.2. In the other six cases, the asymmetric counterparts require parts of the original tour to have the direction of traversal reversed. Although this may lead to alternative tours, it is considered unlikely that such changes in the tour would result in the lowering of the tour length. The *3-opt* implementation will consider the case 1 in Figure 7.2 as the only alternative. The runtime complexity of the *3-opt* heuristic is $O(n^3)$.

The *4-opt* heuristic generates 5 extra alternatives as shown in Figures 7.3.1-7.3.2. (In the symmetric case, there are 46 extra alternatives). Closer inspection of these alternatives reveals that only case 4 in Figure 7.10 involves four new arcs. The remaining three cases involve only three new arcs, and as such, the implementation of the *4-opt* in a straightforward manner involves many repeated calculations of these four cases. The four cases can be efficiently implemented as *3-opt* exchanges. Kanellakis & Papadimitriou (1980) suggest a fast implementation of the *4-opt* exchange of case 4. This implementation, even though it still has a worst case behaviour of $O(n^4)$, should run somewhat faster than the direct implementation.

As the improvement heuristics are likely to be much slower than their construction counterparts, the steepest descent strategy may not always be appropriate. The steepest descent requires a complete search of all possible improvements, followed by the selection of the one with the largest reduction. The search procedure is then repeated until there is no further improvement. In order to study the effects of the selection strategies, two implementations of the *3-opt* and *4-opt* heuristics are tested. The first set, *greedy* strategy, exchanges arcs as soon as a beneficial exchange is found. Once the exchange has taken place, the search is restarted at the last unchanged condition. The second set implements the *steepest descent* strategy. In the greedy strategy, the solutions of the *3-opt* heuristic are used as starting solutions for the *4-opt* searches. Improvement strategies are implemented independently in the implementation of the steepest descent strategy. There are some other exchange strategies, all of which will be faster than the steepest descent strategy and most will be slower than the greedy strategy. The results from the two selected implementations provide bench marks for other *3-opt* and *4-opt* exchange strategies.



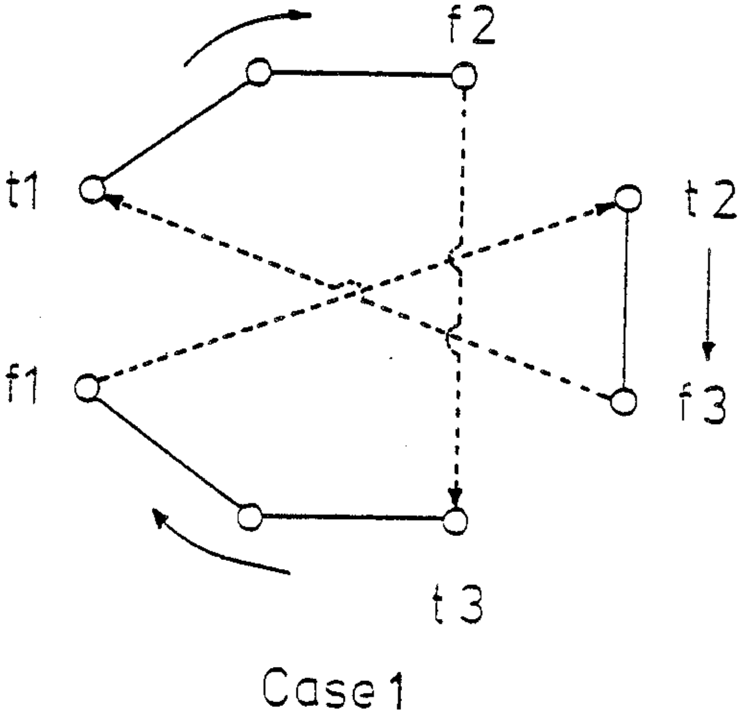
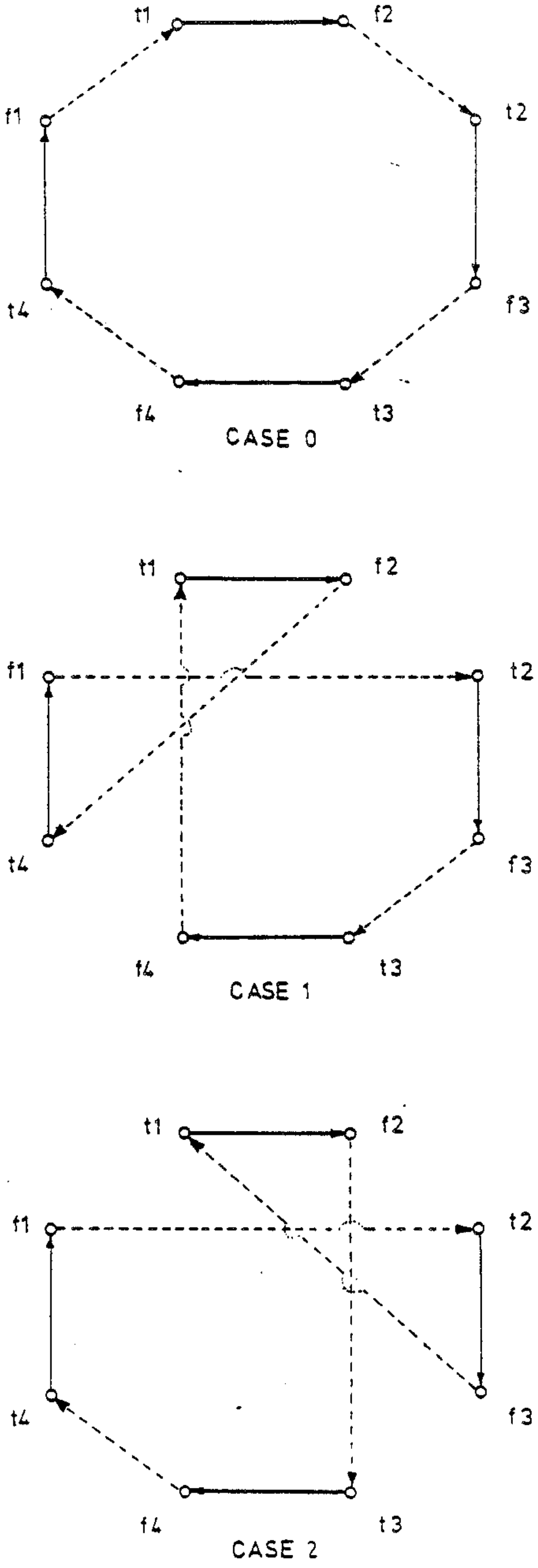


Figure 7.2
3-opt arc exchange

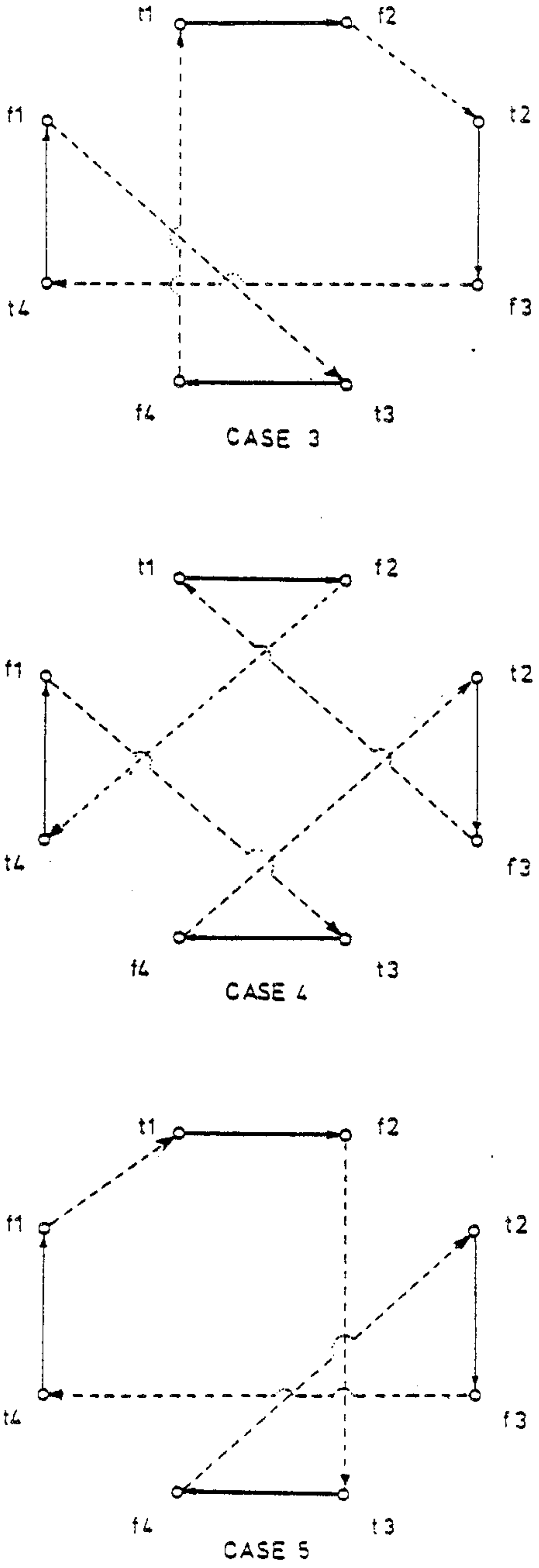


CASE 0

CASE 1

CASE 2

Figure 7.3.1



CASE 3

CASE 4

CASE 5

Figure 7.3.2

Figure 7.3

4-opt arc exchange

	SDW1	SDW2		SDW1	SDW2
1	84	69	21	67	55
2	91	99	22	71	63
3	87	87	23	56	51
4	91	91	24	65	64
5	115	117	25	70	112
6	88	88	26	81	79
7	84	69	27	75	65
8	88	89	28	81	85
9	91	87	29	69	73
10	99	103	30	59	50
11	82	91	31	63	69
12	79	89	32	76	61
13	97	102	33	84	62
14	85	73	34	54	73
15	97	103	35	68	61
16	79	73	36	103	62
17	80	74	37	49	57
18	75	76	38	54	51
19	86	103	39	45	53
20	114	80	40	53	72

Cost range 0-50

	SDW1	SDW2		SWD1	SDW2
1	177	146	21	165	161
2	150	150	22	171	172
3	185	185	23	119	134
4	189	189	24	168	152
5	285	247	25	169	200
6	208	173	26	203	183
7	187	187	27	140	192
8	192	192	28	151	179
9	189	241	29	181	170
10	237	232	30	152	160
11	216	216	31	219	178
12	175	224	32	163	160
13	169	163	33	239	162
14	151	167	34	160	157
15	197	199	35	203	186
16	166	164	36	157	169
17	171	197	37	126	123
18	193	181	38	125	125
19	204	189	39	218	137
20	164	197	40	151	143

Cost range 0-99

Construction solutions of Shadow1 and Shadow2 heurisics
Table 7.1

7.6 SHADOW COST HEURISTICS IN COMPARISONS

The two versions of the shadow cost heuristics, shadow1 and shadow2, are tested by comparing their solutions to randomly generated problems. The sizes of the test problems vary from 20 to 90 cities and the distances between cities vary from 0 to 50 in the first set of 40 problems, and 0 to 99 in the second set of 40 problems. The results of the tests are shown in Table 7.1. In the first set of problems (cost range 0-50) the two heuristics performed equally well; the shadow1 heuristic provides better construction solutions for 18 problems and the shadow2 heuristic provide better solutions on 19 occasions. However, when the cost ranges from 0 to 99, there are some indications, though not statistically significant, that the shadow2 heuristic performed better than the more elaborate shadow1 (shadow2 was better on 20 occasions and shadow1 was better on 13 occasions). As the shadow2 heuristic seems to be more robust than the shadow1 heuristic, the implementations of the shadow cost heuristic in subsequent tests are restricted to the shadow2 formulation only. In addition, *any further reference to the shadow cost heuristic refers to the shadow2 heuristic*, unless stated otherwise.

7.7 COMPARATIVE RESULTS FOR VARIOUS HEURISTICS FOR THE ATSP

In the testing of the heuristics for the ATSP, various practices adopted earlier in the testing of the MPG are also observed. A notable one is that the codes are designed primarily to be both efficient and compact; faster execution times can be achieved if less compact data structures are used. The program, approximately 1600 lines long, is written in Pascal and run on a *Cyber 174* using the *Pascal 6000* compiler, with runtime checking suppressed. The forty test problems are randomly generated with the size ranging from 20 to 90 cities and the cost ranging from 0 to 99.

7.7.1 Comparisons of the Construction Heuristics

Construction solutions by various heuristics are shown in Table 7.2. It is obvious that the Little heuristic is distinctly better than others being tested; the lowest level for the significant tests is 96%. The shadow cost heuristic performs better than the nearest unvisited city heuristic, which in turn is better than the nearest tour insertion routine. A general impression that the nearest tour insertion heuristic performs poorly in larger problems is confirmed by the run test.

Table 7.3 shows the runtime of construction heuristics. The empirical complexity of the Little heuristic is

$$t = 0.37\ n^{2.29} \tag{7.12}$$

and the complexity of the shadow cost heuristic is

$$t = 0.41\ n^{1.85} \tag{7.13}$$

The empirical complexities of both heuristics are less than the theoretical values, $O(n^3)$ and $O(n^2)$; the faster executions were achieved by the use of fast matrix updating procedures which only recalculate the affected elements and employ efficient use of flags. The empirical complexity of the nearest unvisited city heuristic is marginally less than that of the shadow cost heuristic. The empirical complexity of the shortest tour insertion heuristic ($0.16\ n^{2.88}$) is close to the theoretical bound, $O(n^3)$, which is due to the lack of suitable features for fast updating in the algorithm.

7.7.2 Improvement Strategies and Their Consequences

The final results of the combined effort of the construction and improvement heuristics are shown in Tables 7.4-7.7. It is clear from the tables that the relative merits of the construction heuristics are not affected by the use of the improvement heuristics. The only exception is that the shadow cost and *4-opt* heuristics combined to produce results of roughly the same merit as the results produced by the Little and *3-opt* heuristics. The dominant role of the construction heuristics in the ATSP is similar to that found in the MPG.

As mentioned earlier in Section 7.5.2, the overall theoretical time complexities of both improvement heuristics and their possible interactions necessitate some experimentation. Tables 7.4 and 7.5 show the costs and execution times of the final solutions of the greedy strategy, which exchanges arcs as soon as beneficial ones are found. Similarly, Tables 7.6 and 7.7 show the costs and times of the steepest descent strategy. Only 25 smaller problems were examined in the second test as times required for the larger problems were deemed to be excessive.

The effects of the improvement strategies on the Little construction heuristic seem to be minor. They are no obvious gains in applying the steepest descent strategy as far as the *3-opt* heuristic is concerned. For the *4-opt* heuristic, there are some indications, though statistically not significant, that the steepest descent strategy provided better solutions. The relatively small impact may be due to the fact that the Little heuristic provides solutions close to local optimal values, and hence more extensive searches are not always more productive. The expected benefit of the more extensive searches in the improvement strategies is confirmed in the cases where poorer construction heuristics are used. The solutions are significantly poorer in the case where the greedy strategy is used compared to the ones achieved by the use of the steepest descent strategy. The poorer the construction solutions, the larger are the benefits.

The combined performances of the construction and improvement heuristics can be ranked as follows:

Little + *4-opt*
Little + *3-opt*, shadow cost + *4-opt*
shadow cost + *3-opt*
nearest unvisited city + *4-opt*
nearest unvisited city + *3-opt*
shortest tour insertion + *4-opt*
shortest tour insertion + *3-opt*

The complexity implication of the combined heuristics is clear: the steepest descent strategy is very time consuming to execute. For the Little and *3-opt* methods, the empirical complexity of the total runtime is 0.13 $n^{2.74}$ and 0.05 $n^{3.04}$ in the cases of the greedy and steepest descent methods respectively. The time requirement is exacerbated in the case of the *4-opt* heuristic, rising from 0.16 $n^{2.71}$ in the case of greedy strategy to 0.05 $n^{3.20}$ in the case of the steepest descent method. The poorer the initial construction heuristic is, the larger the difference in the two methods.

7.7.3 Implementation Implications

From all the tests carried out, it is evident that the Little construction heuristic provides a cost effective method for obtaining a "good" solution for the ATSP. Approximately 30% of the solutions provided by the Little heuristic cannot be improved by the uses of *3-opt* and *4-opt* heuristics. In the cases where improvements are possible, only one or two iterations are usually needed to reach the local optima. The use of the steepest descent strategy may not be suitable in many cases; it can be argued that for very large problems, say 300 vertices, the difference between the execution times required is too large (27 minutes against 14 minutes). It may be more beneficial to try to obtain additional solutions using alternative construction heuristics. The shadow cost heuristic is a possible alternative, as it has an approximately 30% chance of providing better solutions than those achieved by the Little heuristic. The nearest unvisited city and the shortest tour insertion heuristics generally provide poorer results.

7.8 INTERACTIVE ASPECTS

It is unlikely that an interactive, graphical representation of the results of a large problem will be more useful than a more conventional representation. A possible method of interaction is the manipulation of the distance matrix. As the selection of an arc results in the total exclusion of other contending candidates, it is relatively easy, by changing some elements of the distance matrix, to represent certain operating requirements such as priority jobs and precedence requirements.

7.9 CONCLUSIONS

The comparative solutions and runtimes on the randomly generated problems indicate the clear advantage of the Little construction heuristic over other construction strategies tested. The solutions from the Little construction procedure are usually near or at local optima. The dominance of the construction technique over the improvement procedure is also clear and hence the use of an effective construction heuristic is crucial in obtaining a good result. The excecution times of the steepest descent strategy during the improvement phase for larger problems are found to be prohibitive, and consequently this strategy is not suitable for general use.

PROBLEM			HEURISTICS				
SIZE	NO	LIT	NUC	SDW	NTU	MAX	MIN
20	1	123	197	146	247	247	123
	2	145	255	150	251	255	145
	3	195	291	185	248	291	185
	4	171	304	189	188	304	171
	5	193	293	247	373	373	193
30	6	156	227	173	252	252	156
	7	153	278	187	382	382	153
	8	194	303	192	311	311	192
	9	162	371	241	300	371	162
	10	185	331	232	334	334	185
40	11	179	402	216	373	402	179
	12	179	434	224	369	434	179
	13	167	363	163	341	363	163
	14	153	347	167	328	347	153
	15	198	372	199	373	373	198
50	16	175	365	164	393	393	164
	17	194	338	197	343	343	194
	18	188	386	181	439	439	181
	19	157	399	189	299	399	157
	20	170	376	197	483	483	170
60	21	233	309	161	358	358	161
	22	140	381	172	443	443	140
	23	185	294	134	453	453	134
	24	198	389	152	457	457	152
	25	150	365	200	481	481	150
70	26	177	362	183	445	445	177
	27	225	310	192	525	525	192
	28	273	368	179	418	418	179
	29	152	418	170	473	473	152
	30	129	361	160	465	465	129
80	31	134	351	178	481	481	134
	32	165	347	162	552	552	162
	33	155	363	162	514	514	155
	34	143	309	157	557	557	143
	35	143	387	186	460	460	143
90	36	131	404	169	513	513	131
	37	125	336	123	525	525	123
	38	133	355	125	496	496	125
	39	141	331	137	486	486	137
	40	130	348	143	526	526	130

Table 7.2
Construction costs of ATSP heuristics

PROBLEM		HEURISTICS					
SIZE	NO	LIT	NUC	SHW	STI	MAX	MIN
	1	332	48	105	99	332	48
	2	317	54	101	95	317	54
20	3	302	50	100	95	302	50
	4	318	55	104	91	318	55
	5	325	54	118	93	325	54
	6	692	99	218	293	692	99
	7	709	110	229	279	709	110
30	8	727	102	229	294	727	102
	9	742	111	225	290	742	111
	10	744	105	229	292	744	105
	11	1220	180	384	653	1220	180
	12	1337	181	381	655	1337	181
40	13	1303	177	375	662	1303	177
	14	1317	176	373	677	1317	176
	15	1303	176	389	636	1303	176
	16	2290	267	532	1247	2290	267
	17	2223	268	585	1246	2223	268
50	18	2351	278	559	1222	2351	278
	19	2228	282	553	1260	2228	282
	20	2261	295	557	1272	2261	295
	21	3198	338	742	2119	3198	338
	22	3520	378	778	2131	3520	378
60	23	3191	373	785	2101	3191	373
	24	3376	380	786	2158	3376	380
	25	3332	383	763	2108	3332	383
	26	5358	516	1106	3442	5358	516
	27	5290	477	1065	3412	5290	477
70	28	5203	514	1071	3450	5203	514
	29	5087	539	1066	3411	5087	539
	30	5075	495	1095	3409	5075	495
	31	7318	652	1398	5064	7318	652
	32	7381	639	1387	5043	7381	639
80	33	7248	665	1418	5047	7248	665
	34	7568	615	1393	5085	7568	615
	35	7221	684	1412	5087	7221	684
	36	9614	804	1748	7133	9614	804
	37	9986	810	1626	7045	9986	810
90	38	8909	819	1734	7076	8909	819
	39	9304	770	1704	7106	9304	770
	40	10449	806	1728	7182	10449	806

Table 7.3
Construction time (mil-sec) of ATSP heuristics

		HEURISTICS									
PROBLEM		LIT		NUC		SHW		STI			
SIZE	NO.	30PT	40PT	30PT	40PT	30PT	40PT	30PT	40PT	MAX	MIN
20	1	123	123	163	163	117	117	170	170	170	117
	2	145	145	174	174	150	145	182	176	182	145
	3	193	193	230	189	180	169	233	224	233	169
	4	171	171	189	183	171	171	175	175	189	171
	5	193	193	227	227	211	211	244	244	244	193
30	6	153	153	156	156	157	152	200	169	200	152
	7	145	145	193	176	181	155	209	197	209	145
	8	189	189	204	190	170	167	247	214	247	167
	9	162	162	212	211	193	188	279	254	279	162
	10	185	185	237	237	204	204	226	214	237	185
40	11	173	173	234	223	206	187	217	206	234	173
	12	171	161	195	195	189	167	243	243	243	161
	13	167	141	200	193	161	161	205	192	205	141
	14	149	149	191	154	160	160	182	158	191	149
	15	194	188	240	224	187	186	295	277	295	186
50	16	152	152	203	203	164	164	228	197	228	152
	17	184	168	191	191	166	164	255	240	255	164
	18	162	162	222	209	174	174	208	208	222	162
	19	157	157	190	186	157	155	201	199	201	155
	20	167	167	236	233	173	173	215	206	236	167
60	21	163	163	185	181	158	158	201	186	201	158
	22	140	140	206	192	161	159	258	225	258	140
	23	135	135	175	167	133	131	218	196	218	131
	24	160	151	218	208	152	150	243	240	243	150
	25	150	150	200	196	159	159	237	218	237	150
70	26	167	165	211	194	168	168	211	198	211	165
	27	140	127	198	182	136	136	233	197	233	127
	28	147	147	208	199	165	162	242	233	242	147
	29	152	152	231	208	166	166	245	210	245	152
	30	129	127	190	174	155	155	252	221	252	127
80	31	131	131	209	206	143	143	265	231	265	131
	32	157	154	222	212	154	153	238	225	238	153
	33	140	140	210	210	160	160	215	202	215	140
	34	142	142	179	178	150	150	290	258	290	142
	35	139	139	205	204	159	159	212	212	212	139
90	36	131	131	207	194	153	153	241	230	241	131
	37	125	121	188	188	123	120	240	237	240	120
	38	125	119	165	160	121	121	233	211	233	119
	39	127	126	179	179	137	136	273	246	273	126
	40	130	130	221	202	143	143	224	219	224	130

Table 7.4
Final solutions of ATSP heuristics
(Greedy exchange strategy)

		HEURISTICS									
PROBLEM		LIT		NUC		SDW		STI			
SIZE	NO.	30PT	40PT	30PT	40PT	30PT	40PT	30PT	40PT	MAX	MIN
	1	123	123	163	148	117	117	140	123	163	117
	2	145	145	167	158	150	145	209	145	209	145
20	3	193	184	170	170	165	165	165	165	193	165
	4	171	171	194	178	171	171	175	175	194	171
	5	193	193	227	227	205	205	235	236	236	193
	6	153	153	156	165	157	152	187	177	187	152
	7	145	145	161	145	181	149	151	219	219	145
30	8	182	182	186	186	170	164	189	179	189	164
	9	162	162	203	173	188	167	168	165	203	162
	10	185	185	211	201	200	188	200	193	211	185
	11	173	173	201	164	206	177	208	208	208	164
	12	171	161	180	193	185	181	243	184	243	161
40	13	167	153	165	180	140	140	171	186	186	140
	14	147	147	140	140	152	152	146	129	152	129
	15	194	185	217	204	187	186	257	242	257	185
	16	152	155	173	187	164	164	206	202	206	152
	17	187	168	199	185	166	177	223	217	223	166
50	18	162	162	185	210	172	172	216	195	216	162
	19	157	157	171	167	157	154	186	186	186	154
	20	167	167	175	201	173	184	198	182	201	167
	21	145	145	194	176	158	153	196	216	216	145
	22	140	140	175	156	161	149	178	186	186	140
60	23	137	131	158	132	133	131	185	145	185	131
	24	144	149	189	175	152	150	188	217	217	144
	25	150	150	184	183	148	148	221	192	221	148

Table 7.5
Final solutions of ATSP heuristics
(Steepest descent strategy)

		HEURISTICS									
PROBLEM		LIT		NUC		SDW		STI			
SIZE	NO.	30PT	40PT	30PT	40PT	30PT	40PT	30PT	40PT	MAX	MIN
20	1	520	569	252	295	338	383	355	397	569	252
	2	517	559	260	300	300	369	335	390	559	260
	3	492	543	283	389	324	389	299	346	543	283
	4	517	569	269	332	301	351	285	324	569	269
	5	524	566	280	323	348	390	343	385	566	280
30	6	1407	1501	933	1033	944	1067	1052	1252	1501	933
	7	1536	1662	950	1035	918	1066	1393	1534	1662	918
	8	1496	1621	846	990	966	1092	1276	1548	1621	846
	9	1438	1541	1159	1297	1016	1152	1048	1185	1541	1016
	10	1433	1533	821	919	1008	1126	1082	1226	1533	821
40	11	2963	3144	2463	2791	2021	2426	3070	3293	3293	2021
	12	3106	3364	2639	2861	2325	2599	2828	3096	3364	2325
	13	2923	3185	2388	2628	2058	2268	2889	3341	3341	2058
	14	3062	3238	2108	2375	2122	2306	2722	3032	3238	2108
	15	3078	3284	2333	2619	2199	2451	2826	3108	3284	2199
50	16	6315	6725	4232	4501	3870	4186	5752	6280	6725	3870
	17	5636	6083	4044	4425	3989	4371	5057	5684	6083	3989
	18	5981	6266	4333	4678	4160	4431	6582	6867	6867	4160
	19	5590	5875	4979	5332	4221	4634	5180	5618	5875	4221
	20	5660	6015	4536	4862	4282	4640	6430	6822	6822	4282
60	21	9920	10384	6944	7639	6703	7188	10345	10858	10858	6703
	22	9312	9742	8472	9055	7077	7791	10503	11472	11472	7077
	23	9577	10082	7357	7952	6579	7276	11048	12015	12015	6579
	24	9460	9944	7539	8125	6625	7100	10259	10801	10801	6625
	25	9208	9652	7609	8126	6947	7338	11157	11718	11718	6947
70	26	15595	16286	12246	13650	10993	11565	18847	20225	20225	10993
	27	16403	17172	11585	12388	11936	12509	18062	19186	19186	11585
	28	16038	16633	12689	13568	11368	12049	17757	18827	18827	11368
	29	14875	15504	13339	14443	11204	11776	16817	17959	17959	11204
	30	14845	15438	12297	13021	11238	11807	17739	19100	19100	11238
80	31	22406	23279	19303	20592	17310	18060	24494	26367	26367	17310
	32	22259	23010	18455	19536	17128	18167	26873	27996	27996	17128
	33	22319	23069	17405	18358	16406	17157	27873	28906	28906	16406
	34	22194	22904	17937	19151	16535	17288	26282	27746	27746	16535
	35	22103	22849	18293	19182	16710	17462	26318	27116	27116	16710
90	36	30613	31618	27281	28637	24027	25197	35295	36956	36956	24027
	37	30897	32357	23739	24645	22671	23669	35733	37421	37421	22671
	38	30918	32117	26986	27878	22832	23803	39471	41007	41007	22832
	39	31216	32918	25698	26834	22593	23723	36743	38280	38280	22593
	40	31250	32204	26028	27607	22825	23768	43044	44501	44501	22825

Table 7.6
Total runtimes (mil-sec) of ATSP heuristics
(Greedy exchange strategy)

		HEURISTICS									
PROBLEM		LIT		NUC		SDW		STI			
SIZE	NO.	3OPT	4OPT	3OPT	4OPT	3OPT	4OPT	3OPT	4OPT	MAX	MIN
	1	391	572	326	1036	365	720	792	1895	1895	326
	2	390	572	473	1065	218	574	358	1642	1642	218
20	3	530	1282	1056	2401	371	729	789	1704	2401	371
	4	387	572	477	1463	364	722	506	1041	1463	364
	5	384	560	483	1043	375	737	1078	2016	2016	375
	6	1582	2770	3158	4990	2200	5204	3782	6850	6850	1582
	7	1605	2841	2644	5279	1198	3672	7870	12527	12527	1198
30	8	1598	2817	1615	3545	1719	4716	5348	11476	11476	1598
	9	1107	1712	4699	9268	1718	4878	5356	11932	11932	1107
	10	1095	1682	2637	7666	2220	6385	3793	8256	8256	1095
	11	4765	8990	7613	24223	2759	11199	11631	24592	24592	2759
	12	4879	9273	13782	24678	5314	10990	12888	31428	31428	4879
40	13	2307	5180	12577	25600	3996	8139	21528	34704	34704	2307
	14	4788	8968	18771	36600	4022	8219	16592	35932	36600	4022
	15	3665	6562	10101	20516	4040	8297	12966	23184	23184	3665
	16	11813	25875	29996	55947	2872	5600	35682	75290	75290	2872
	17	6897	18376	22714	34889	10440	16058	28095	44933	44933	6897
50	18	9249	17390	39863	66041	5391	10828	40640	89154	89154	5391
	19	4149	6833	22586	52186	5325	13592	23156	47868	52186	4149
	20	6654	12073	42471	76934	17765	28735	43344	92380	92380	6654
	21	15424	30080	21765	51148	13417	31822	70058	131896	131896	13417
	22	6956	11645	34489	79357	17674	49976	108363	201363	201363	6956
60	23	11208	20836	38726	90900	9151	23283	87493	208613	208613	9151
	24	15484	29839	51847	110109	4883	14104	104620	183913	183913	4883
	25	7143	11984	43294	91659	17798	32040	83478	188712	188712	7143

Table 7.7
Total runtimes (mil-sec) of ATSP heuristics
(Steepest descent strategy)

8 Conclusions and recommendations

The three classes of mathematically-related problems selected are the principal ones that need to be solved if effective decentralisation of decision making within a factory is to take place. The continuing reduction in the cost of microprocessors and the advances made in the area of computer networking have greatly reduced the difficulties imposed by hardware on the realisation of this objective. The main aim of the thesis has been to solve some of the software problems that may arise in the decentralisation process.

One of the more obvious routes to decentralisation is to have group layout instead of the more usual functional layout. The rank order clustering algorithm, (ROC), has been adapted and developed into a fast and compact interactive scheme, called the ROC2 algorithm, for the purpose of grouping components and machines. Problems which require weeks of manual effort or which cannot be solved by other methods are solved by the ROC2 algorithm with modest human and computing resources, and solutions produced for known test problems are as good as or better than, those generated by other methods. As a general clustering technique, the ROC2 algorithm has been shown to be an effective partitioning scheme for the set covering problem.

Following the grouping of machines, the question of their layout must be solved. Two models for layout, the quadratic assignment problem, (QAP), and the maximal planar graph problem, (MPG), are investigated. A short experiment on the QAP model has highlighted the potential benefit of using the ROC2 algorithm in generating an initial layout. For the MPG, various construction and improvement heuristics, which do not require planarity testing procedures, are studied. This is believed to be the first report on computer implemented heuristics for the MPG. The final part of the thesis is concerned with scheduling, which can be made more effective in many environments if properly decentralised. A class of scheduling problem, the sequence-dependent setup time scheduling problem, (SDSTSP), is selected for study, and various construction and improvement heuristics were tested.

A general conclusion that can be drawn from the various heuristics tested is the dominant role of the construction over the improvement heuristics. On the interactive aspect, it seems clear that where a problem can only be partially defined quantitatively, and the solution provided by the algorithm alone may therefore not be satisfactory, interaction can play a useful complementary role to the algorithm. In cases where the problem is well defined, such as some scheduling problems, interaction is less important, although it can still be useful in dealing with exceptional circumstances.

Two further pieces of work could usefully be carried out in the future; firstly a data collection routine could be developed as an interface between the ROC2 algorithm and real life problems; secondly the ROC2 algorithm and plant layout routines could be combined into one package. These steps could help to reduce further the practical difficulties in implementing group layout.

Abdel Barr,S.E.Z. (1978) A Computerised Approach to Facility Layout, PhD Thesis, University of Nottingham.

Abdel Barr,S.E.Z. & O'Brien,C. (1976) A Procedure for Solving the Facility Layout Problem Using a Multi-Pairwise Exchange, 2nd Annual Operations Research Conference, **2/2**, Egypt, 1976.

Alk,S.G. (1980) The Minimum Directed Spanning Graph for Combinatorial Optimization, *The Australian Computer Journal*, **12/4**(132-136).

Apple,J.M. (1977) *Plant Layout and Material Handling*, 3rd ed., John Wiley & Sons, New York.

Armour,G.C. & Buffa,E.S. (1963) A Heuristic Algorithm and Simulation Approach to Relative Location of Facilities, *Man.Sc.*, **9/1**(294-303).

Balas,E. & Christofides,N. (1981) A Restricted Lagrangean Approach to the Travelling Salesman Problem, *Math.Prog*, **21**(19-46).

Baybars,I. (1979) Characterization of Maximal Planar Graphs, Generating Planar Graphs and 0-1 Program for Determining the Optimal Spanning Subgraph of a Weighted Graph, Carnegie-Mellon University, Pittsburg.

de Beer,C. & de Witte,J. (1978) Production Flow Synthesis, *Annals of the CIRP*, **27/1**(389-392).

de Beer,C., van Gerwen,R. & de Witte,J. (1976) Analysis of Engineering Production Systems as a Base for Production-Oriented Reconstruction, *Annals of the CIRP*, **25/1**(439-441).

Berztiss,A.T. (1975) *Data Structure Theory and Practice*, 2nd. ed, Academic Press, New York.

Block,T.E. (1977) A Note on 'Comparison of Computer Algorithms and Visual Based Methods for Plant Layout' by M. Scriabin and R.C. Vergin, *Man.Sc.*, **24/2**(235-237).

——— (1979) On the Complexity of Facilities Layout Problems, *Man.Sc.*, **25/3**(280-285).

Buffa,E.S. (1955) Sequencing Analysis for Functional Layout, *J.Ind.Eng.*, **6/2**(12-16).

——— (1976) On a Paper by Scriabin and Vergin, *Man.Sc.*, **32/1**(104).

Buffa,E.S., Ammour,C.G. & Vollmann,T.E. (1964) Allocating Facilities with CRAFT, *Harvard Business Review*, **42/2**(136-158).

Burbidge,J.L. (1963) Production Flow Analysis, *Prod.Engnr.*, **42**(742-752), Dec 63.

——— (1971) Production Flow Analysis, *Prod.Engnr.*, **50**(139-152), Apr/May 71.

——— (1973) Production Flow Analysis on the Computer, 3rd. Annual Conf., Inst. of Prod. Eng., Nov 73.

——— (1977) A Manual Method of Production Flow Analysis, *Prod.Engnr.*, **56**(34-38), Oct 77.

Burkard,R.E. & Stratmann,K-H. (1978) Numerical Investigation on Quadratic Assignment Problems, *Naval Research Logistics Quarterly*, **25/1**(129-148), March 78.

Cameron,D.C. (1952) Travel Charts- A Tool for Analyzing Material Movement Problem, *Modern Material Handling*, **8/1**.

Carpaneto,G. & Toth,P. (1980) Some New Branching and Bounding Criteria for the Asymmetric Travelling Salesman Problem, *Man.Sc.*, **26/7**(736-743).

Carrie,A.S. (1974) Numerical Taxonomy Applied to Group Technology and Plant Layout, *Proc. 2nd. Int. Conf. on Prod. Res.*, Copenhagen, Aug 73, 337-354.

Christofides,N. (1977) Lecture Notes.

——— (1979) The Travelling Salesman Problems, in *Combinatorial Optimization*, edited by Christofides *et al*, 131-149, John Wiley, Chichester.

Christofides,N., Gailiani,G. & Stefanini,L. (1980) An Algorithm for the Maximal Planar Graph Problem Based on Lagrangean Relaxation, Department of Management Science, Imperial College, Research Paper IC.OR.80-21.

Christofides,N. & Gerrard,M (1976) Special Cases of the Quadratic Assignment Problems, Management Science Research Report No 391, Graduate School of Industrial Administration, Carnegie-Mellon University.

Christofides,N., Mingozzi,A. & Toth,P. (1980) Contributions to the Quadratic Assignment Problem, *Euro.J.Ops.Res.*, **4**(243-247).

Cook,S.A. (1971) The Complexity of Theorem-Proving Procedures, *Proc. 3rd ACM Symposium on Theory of Computing* (151-158).

Crowder,H & Padberg,M.W. (1980) Large-Scale Symmetrical Travelling Salesman Problems. *Man.Sc.*, **26/5**(495-509).

Edwards,C.S. (1977) The Derivation of a Greedy Approximation for the Koopmans-Beckmann Quadratic Assignment Problem, *Proc. Combinatorial Programming 77*, University of Liverpool, 13-15 Sept 77.

——— (1980) A Branch and Bound Algorithm for Koopmans-Beckmann Quadratic Assignment Problem, in *Combinatorial Optimization II*, edited by Rayward-Smith,V.J., North Holland.

Edwards,G.A.B. (1972) Correspondence, *Prod.Engnr.*, **51**(278), Jul/Aug 72.

Edwards,H.K., Gillet,B.E. & Hale,M.E. (1970) Modular Allocation Technique, *Man.Sc.*, **17/3**(161-169).

Eilon,S., Watson-Gandy,C.D.T. & Christofides,N. (1971) *Distribution Management*, Griffin, London.

El-Essawy,I.F.K. (1971) The Development of Component Flow Analysis in Production Systems' Design for Multi-Product Engineering Companies, PhD Thesis, UMIST.

——— (1972) Correspondence, *Prod.Engnr.*, **51**(278), Jul/Aug 72.

El-Essawy,I.F.K. & Torrance,J. (1972) Component Flow Analysis an Effective Approach to Production Systems' Design, *Prod.Engnr.*, **51**(165-170), May 72.

El-Rayah,T.E. & Hollier,R.H. (1970) A Review of Plant Design Techniques, *Int.J.Prod.Res.*, **8/3**(263-279).

Foulds,L.R. & Robinson,D.E. (1976) A Strategy for Solving the Plant Layout Problem, *Opl.Res.Q.*, **27/4,i**(845-855).

——— (1978) Graph Theorectical Heuristics for the Plant Layout Problem, *Int.J.Prod.Res.*, **16/1**(27-37).

Francis,R.L. & White,J.A. (1974) *Facility Layout and Location*, Prentice-Hall, New Jersey.

Frieze,A.M., Galbiati,G. & Maffioili,F. (1982) On the Worst-Case Performance of Some Algorithms for the Asymmetrical Travelling Salesman Problem, *Network*, **12/1**(23-39).

Gallagher,C.C. & Knight,W.A. (1973) *Group Technology*, Butterworths, London.

Garey,M.R., Graham,R.L. & Johnson,D.S. (1976) Some NP-complete Geometric Problems, *Proc 8th ACM Sym. on Theory of Computing* 1976.

Garey,M.R. & Johnson,D.S. (1979) *Computers and Intractability*, W.H. Freeman, San Francisco.

Gavett,J.W. & Plyter,N.V. (1966) The Optimal Assignment of Facilities to Locations by Branch and Bound, *Ops.Res.*, **14/2**(210-232).

Gavish,B & Graves,S.C. (1979) The Travelling Salesman Problem and Related Problems, Working

Paper 7906, Graduate School of Management, U. of Rochester.

Gilmore,P.C. (1962) Optimal and Sub-optimal Algorithms for the Quadratic Assignment Problem, *J. SIAM*, **10/2**(305-313).

Golden,B.E. & Assad,A.A. (1982) An Analytical Framework for Comparing Heuristics, Working Paper MS/S 82-002, College of Business and Management, U. of Maryland.

Golden,B.L. & Stewart,W.R. (1981) The Empirical Analysis of TSP Heuristics, Working Paper MS/S 81-040, College of Business and Management, U. of Maryland.

Gonzales,R.H. (1962) Solutions to the Travelling Salesman Problem by Dynamic Programming on the Hypercube, Technical Report No. 18, OR Centre, MIT.

Graham,I., Galloway,P. & Scollar,I. (1976) Model Studies in Computer Seriation, *J.Archeological Sc.*, **3/1**(1-30).

Graves,G.W. & Whinston,A.B. (1970) An Algorithm for the Quadratic Assignment Problem, *Man.Sc.*, **17/7**(453-471).

Grigorriadis,M.D. (1980) Partitioning Methods for Block-Diagonal Linear Systems and Programs, A paper presented at the International Workshop on Advances in Linear Optimization Algorithms and Software, Pisa, Italy, July 1980.

Hansen,K.H. & Krarup,J. (1974) Improvements of the Held and Karp Algorithm for the Symmetrical Travelling Salesman Problem, *Math.Prog.*, **4**(87-98).

Harary,F. (1971) Sparse Matrices and Graph Theory, in *Large Sparse Set of Linear Equations*, Reid,J.K.(ed) (1971), 139-167, Academic Press, London.

Held,M. & Karp,R. (1970) The Travelling Salesman Problem and Minimum Spanning Trees, *Ops.Res.* **26/6**(1138-1162).

——————— (1971) The Travelling Salesman Problem and Minimum Spanning Trees II, *Math.Prog.*, **1**(6-25).

Hey,A.M. (1980) Algorithms for the Set Covering Problems, PhD Thesis, Department of Management Science, Imperial College, London.

Hillier,F.S. (1963) Quantitative Tools for Plant Layout Analysis, *J.Ind.Eng.*, **14/1**(34-40).

Hillier,F.S. & Michael,M.C. (1966) Quadratic Assignment Problem Algorithms and the Location of Indivisible Facilities, *Man.Sc.*, **13/1**(42-57).

Hiscox,W.J. (1948) *Factory Lay-out Planning and Progress*, 4th.ed., Pitman, London.

Hitchings,G.C. (1973) Analysis and Development of Techniques for Improving the Layout of Plant and Equipment, PhD Thesis, University of Wales, Cardiff.

Hitchings,G.C. & Cottam,M. (1976) An Effective Heuristic Procedure for Solving the Layout Design Problem, *Omega*, **4/2**(205-214).

Hopcroft,J. & Tarjan,R. (1974) Efficient Planarity Testing, *Journal ACM*, **21/4**(549-568).

Horowitz,E. & Sanhi,S. (1976) *Fundamentals of Data Structures*, 134-140, Pitman, London.

Iri,M. (1968) On the Synthesis of Loop and Cutset Matrices and the Related Problems, *SAAG Memoirs*, **4**(376-410), A-XIII.

Kanellakis,P-C. & Papadimitriou,C.H. (1980) Local Search for the Asymmetric Travelling Salesman Problem, *Ops.Res.*, **28/5**(1086-1099).

Karp,R. (1972) Reduciblitiy among Combinatorial Problems, from *Complexity of Computer Computation*, edited Miller,R.E. & Thatcher,J.W. (85-103) Plenum Press, New York.

—— (1979) A Patching Algorithm for the Nonsymmetric Travelling Salesman Problem, *SIAM J.Computing*, **8/4**(561-573).

Kaufman,L. & Broeckx,F. (1978) An Algorithm for Quadratic Assignment Problem Using Bender's Decomposition, *Euro.J.Ops.Res*, **2/3**(207-211).

Krolak,P., Felts,W. & Marble,G. (1971) A Man-Machine Approach Towards Solving the Travelling Salesman Problem, *Comm. of the ACM*, **14/5**(327-334).

King,J.R. (1979) Machine-Component Group Formation in Group Technology, *Proc. Vth Int. Conf. on Prod. Res.*, Aug 79, 40-44, also *Omega*, **8/2**(193-199).

—— (1980) Machine-Component Grouping in Production Flow Analysis: An Approach Using A Rank Order Clustering Algorithm, *Int.J.Prod.Res*, **18/2**(213-232).

King,J.R. & Spachis,A.S. (1980) Heuristics for Flow Shop Scheduling, *Int.J.Prod.Res.*, **18/3**(345-357).

Koopmans,T.C. & Beckmann,M.J. (1957) Assignment Problems and Location of Economic Activities, *Econometrica*, **25**(52-76).

Krejcirik,M. (1968) *Computer Aided Building Layout*, Booklet I, 1968 IFIP Congress, Edinburgh.

—— (1969) Computer Aided Plant Layout, *Computer Aided Design*, Autumn 1969,(7-19).

Land,A.H. (1963) A Problem of Assignment with Inter-related Costs, *Opl.Res.Q.*, **14**(185-199).

Lawler,E.L. (1963) The Quadratic Assignment Problems, *Man.Sc.*, **9/4**(586-599).

—— (1975) The Quadratic Assignment Problem: A Brief Review, in *Combinatorial Programming: Methods and Applications*, edited by Roy, 351-360, D.Reidel Publishing, Dordrecht-Holland.

Liggett,R.S. (1981) The Quadratic Assignment Problem: An Experimental Evaluation of Solution Strategies, *Man.Sc.*, **27/4**(442-458).

Lin,S. (1965) Computer Solutions to the Travelling Salesman Problem, *Bell System Technical Journal*, **44/10**(2245-2269).

Little,J.D.C., Sweeny, D.W. & Karel,C. (1963) An Algorithm for the Travelling Salesman Problem, *Ops.Res.*, **11/6**(972-989).

Llewellyn,R.W. (1958) Travel Charting with Realistic Criteria, *J.Ind.Eng.*, **9/3**(217-220).

Los,M (1978) Comparison of Several Heuristic Algorithms to Solve Quadratic Assignment Problems of the Koopmans-Beckmann Type, a paper presented at the Int. Sym. on Locational Decision at Bann, Alberta. 24th-28th April 1978.

Lundy.J.L. (1955) A Reply to W.P. Smith's Article, *J.Ind.Eng.*, **6/3**(9).

McAuley,J. (1972) Machine Grouping for Efficient Production, *Prod.Engnr.*, **51**(53-57), Feb 72.

McCormick,W.T., Schweitzer,P.J & White,T.W. (1972) Problem Decomposition and Data Reorganization by a Clustering Technique, *Ops.Res.*, **20**(993-1009).

Miliotis,P. (1976) Integer Programming Approaches to the Travelling Salesman Problem, *Math.Prog.*, **10**(367-378).

Mojena,R., Vollmann,T.E. & Okamoto,V. (1976) On Predicting Computational Time of a Branch and Bound Algorithm for the Assignment of Facilities, *Decision Sc.*, **7**(856-867).

Moon,J.W. & Morse,L. (1965) On Cliques in Graphs, *Israel J.Maths.*, **3/1**(23-28).

Moore,J.M. (1962) *Plant Layout and Design*, Macmillan, New York.

—— (1973) Computer Aided Facilities Design: An International Survey, *Proc 2nd Int. Conf. on*

Prod. Res., edited by Gudnason,C.H. & Corlett,E.N. (1974), (479-502), Taylor Francis, London.

——— (1976) Facilities Design with Graph Theory and Strings, *Omega*, **4/3**(193-202).

——— (1977) Who Uses the Computer for Layout Planning, *Proc 4th Int. Conf. on Prod. Res.*, edited by Muramats,R. & Dudley,N.A. (1978), (829-844), Taylor Francis, London.

——— (1979) The Zone of Compromise for Evaluating Layout Arrangement, *Proc Vth Int. Conf. on Prod. Res.*, Aug. 1979(24-27).

Muther,R. (1961) *Systematic Layout Planning*, Industrial Education Institute.

Muther,R. & Wheeler,J.D. (1962) Simplified Systematic Layout Planning, *Factory*, **120/8**(68-77), **120/9**(111-119), **120/10**(101-113).

Muther,R. & McPherson,R. (1970) Four Approaches to Computerized Layout Planning, *Indstrl. Engnr.*, **2/2**(39-42)

Noy,P.C. (1957) Make the Right Plant Layout... Mathematically, *Amer.Mechanist*, **101/19**(121-125).

Nugent,C.E., Vollman,T.E. & Rulm,J. (1968) An Experimental Comparison of Techniques for the Assignment of Facilities to Locations, *Ops.Res.*, **16/1**(150-173).

Paixao,J.M.P. (1982) Private Communication.

Papadimitriou,C.H. & Steiglitz,K. (1978) Some Examples of Difficult Travelling Salesman Problem, *Ops.Res.*, **26/3**(434-443).

Parker,C.S. (1976) An Experimental Investigation of Some Heuristic Strategies for Component Placement, *Opl.Res.Q.*, **27/1**,i(71-81).

Pemberton,A.W. (1974) *Plant Layout and Material Handling*, Macmillan, London.

Pierce,J.F. & Crowston,W.E. (1971) Tree Search Algorithms for Quadratic Assignment Problems, *Naval Research Logistics Quarterly*, **18/1**(1-36).

Pooch,U.W. & Nieder,A. (1973) A Survey of Indexing Techniques for Sparse Matrices, *Computing Survey*, **5/2**(109-133), Jun 73.

Purcheck,G.F.K. (1974) Combinatorial Grouping - A Lattice-Theoretic Method for the Design of Manufacturing Systems, *J.Cybernatics*, **4/3**(27-60).

——— (1975*a*) A Mathematical Classification as a Basis for the Design of Group Technology Production Cells, *Prod.Engnr.*, **54**(35-48).

——— (1975*b*) A Linear Programming Method for the Combinatoric Grouping of an Incomplete Power Set, *J.Cybernatics*, **5/4**(51-76).

Rajagopalan,R. & Batra,J.L. (1975) Design of Cellular Production Systems. A Graph Theoretic Approach, *Int.J.Prod.Res.*, **13/6**(567-579).

Ritzman,L.P. (1972) The Effieciency of Computer Algorithms for Plant Layout, *Man.Sc.*, **18/5**(240-247).

Rosenkrantz,D.J., Stearns,E.S. & Lewis,P.M. (1977) An Analysis of Several Heuristics for the Travelling Salesman Problem, *SIAM J.Computing*, **6/3**(563-581).

Schneider,M. (1960) Cross Charting Techniques as a Basis for Plant Layout, *J.Ind.Eng.*, **11/6**(478-483).

Scriabin,M. & Vergin,R.C. (1975) Comparison of Computer Algorithms and Visual Based Methods for Plant Layout, *Man.Sc.*, **22/2**(172-181).

Seppanen,J.J. & Moore,J.M. (1970) Facilities Planning with Graph Theory, *Man.Sc.*, **17/4**(B242-

B253).

—————— (1975) String Processing Algorithms for Plant Layout Problems, *Int.J.Prod.Res.*, **13/3**(239-245).

Smith,W.P. (1955) Travel Charting- Firt Aid for Plant Layout, *J.Ind.Eng.*, **6/1**(13-15).

Sneath,P.H.A. & Sokal,R.R. (1973) *Numerical Taxonomy*, W.H.Freeman & Co., San Francisco.

Spachis,A.S (1979) Job-Shop Scheduling with Approximate Methods, PhD thesis, Department of Management Science, imperial College.

Tewason,R.P. (1973) *Sparse Matrices*, Academic Press, New York.

Trybus,T.W. & Hopkins,L.D. (1980) Human *vs* Computer Algorithms for the Plant Layout Problem, *Man.Sc.*, **26/6**(570-574).

Van Der Cruysen,P. & Rijckaert,M.J. (1978) Heuristic for the Asymmetric Travelling Salesman Problem, *J.Opl.Res.Soc.*, **29/7**(687-701).

Vollmann,T.E. (1964) An Investigation of the Bases for Relative Location of Facilities, Doctoral Thesis, University of California, Los Angeles.

Vollmann,T.E. & Buffa,E. (1966) The Facilities Layout Problem in Perspective, *Man.Sc.*, **12/10**(B450-B468).

Vollmann,T.E., Nugent,C.E. & Zartlet,R. (1968) A Computerized Model for Office Layout, *J.Ind.Eng*, **19/7**(321-327).

de Witte,J. (1979) The Use of Similarity Coefficients in Production Flow Analysis, *Proc. Vth Int. Conf. on Prod. Res.*, Aug 79, (36-39).

Locations

1									10										20				24		
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4		
0	1	2	3	4	5	1	2	3	4	5	6	2	3	4	5	6	7	3	4	5	6	7	8	1	
	0	1	2	3	4	2	1	2	3	4	5	3	2	3	4	5	6	4	3	4	5	6	7	2	
		0	1	2	3	3	2	1	2	3	4	4	3	2	3	4	5	5	4	3	4	5	6	3	
			0	1	2	4	3	2	1	2	3	5	4	3	2	3	4	6	5	4	3	4	5	4	
				0	1	5	4	3	2	1	2	6	5	4	3	2	3	7	6	5	4	3	4	5	
					0	6	5	4	3	2	1	7	6	5	4	3	2	8	7	6	5	4	3	6	
						0	1	2	3	4	5	1	2	3	4	5	6	2	3	4	5	6	7	7	
							0	1	2	3	4	2	1	2	3	4	5	3	2	3	4	5	6	8	
								0	1	2	3	3	2	1	2	3	4	4	3	2	3	4	5	9	L
									0	1	2	4	3	2	1	2	3	5	4	3	2	3	4	10	o
										0	1	5	4	3	2	1	2	6	5	4	3	2	3	11	c
											0	6	5	4	3	2	1	7	6	5	4	3	2	12	a
												0	1	2	3	4	5	1	2	3	4	5	6	13	t
													0	1	2	3	4	2	1	2	3	4	5	14	i
														0	1	2	3	3	2	1	2	3	4	15	o
															0	1	2	4	3	2	1	2	3	16	n
																0	1	5	4	3	2	1	2	17	s
																	0	6	5	4	3	2	1	18	
																		0	1	2	3	4	5	19	
																			0	1	2	3	4	20	
																				0	1	2	3	21	
																					0	1	2	22	
																						0	1	23	
																							0	24	

Distance matrix for the QAP

Machines

1									10										20				24		
1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4		
0	2	0	0	0	0	0	0	2	0	0	0	0	0	0	2	2	0	1	0	0	0	0	0	1	
	0	0	0	0	0	0	0	8	0	0	0	0	1	0	6	5	0	4	0	0	0	0	0	2	
		0	0	0	0	0	0	0	0	0	0	0	2	0	1	4	0	0	0	0	0	0	0	3	
			0	7	0	0	0	0	0	0	0	0	0	4	0	0	3	0	4	0	0	0	0	4	
				0	0	0	0	0	0	10	0	0	0	7	0	0	6	0	8	0	0	0	0	5	
					0	2	2	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	
						0	1	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7	
							0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	
								0	0	0	0	0	1	0	8	5	0	4	0	0	0	0	0	9	M
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	10	a
										0	3	2	0	0	0	0	0	0	0	4	0	0	0	11	c
											0	1	0	0	0	0	0	0	0	3	0	0	0	12	h
												0	0	0	0	0	0	0	0	2	0	0	0	13	i
													0	0	1	3	0	1	0	0	0	0	0	14	n
														0	0	0	4	0	4	0	0	0	0	15	e
															0	6	0	3	0	0	0	0	0	16	s
																0	0	2	0	0	0	0	0	17	
																	0	0	4	0	0	0	0	18	
																		0	0	0	0	0	0	19	
																			0	0	0	0	0	20	
																				0	0	0	0	21	
																					0	0	0	22	
																						0	0	23	
																							0	24	

Weight matrix for the QAP

PROBLEM IDEN.	FINAL COST	NO. OF ITERATION(S)	EXEC. TIME (CYBER174 SEC)
1	273	12	0.484
2	276	13	0.516
3	276	11	0.467
4	266	10	0.434
5	280	8	0.360
6	281	10	0.431
7	277	9	0.391
8	279	9	0.393
9	268	8	0.350
10	288	8	0.352

The solutions to the 16 location configuration

PROBLEM IDEN.	INITIAL LAYOUTS															
1	2	10	9	6	3	12	13	11	5	4	7	14	1	15	8	16
2	6	5	12	15	11	1	8	14	13	10	7	4	16	3	2	9
3	11	15	2	16	14	9	8	7	10	12	6	1	3	13	4	5
4	6	14	9	4	7	2	13	1	5	8	15	12	10	16	3	11
5	16	14	13	4	6	8	3	12	2	10	15	11	5	7	9	1
6	4	8	12	1	14	13	6	3	15	2	7	5	9	11	10	16
7	13	4	6	3	5	1	15	12	8	9	16	11	7	14	2	10
8	9	1	15	10	4	8	3	14	16	5	2	13	12	6	7	11
9	3	15	12	10	8	11	16	6	14	1	5	2	9	13	7	4
10	9	10	6	5	1	12	16	15	2	3	14	7	8	11	4	13

Random starting layouts for the 16 location configuration

```
PROGRAM layout3(data, output, input /);

CONST
   maxactivity = 30;
   maxlocation = 30;
   maxdistance = 100;
   maxweight = 100;
   infinity = 9999999;

TYPE
   activity = 1..maxactivity;
   location = 1..maxlocation;
   distance = 0..maxdistance;
   weight = 0..maxweight;
   arrayweight = ARRAY
      [activity, activity] OF weight;
   arraydistance = ARRAY
      [location, location] OF distance;
   arrayswitchcost = ARRAY
      [location, location] OF integer;
   arrayactinloc = ARRAY
      [location] OF activity;
   arraylacofact = ARRAY
      [activity] OF location;
   setoffixedlocations = SET OF location;

VAR
   data: text;
   w, weightsubprob: arrayweight;
   d, dsubprob: arraydistance;
   costofswitchmacinloc: arrayswitchcost;
   macinloc, tempmacinloc, oldmacinloc: arrayactinloc;
   locationsfixed: setoffixedlocations;
   locofmac, templocofmac: arraylacofact;
   oldmacname: ARRAY
       [activity] OF activity;
   oldlocname: ARRAY
       [location] OF location;
   initlayoutgiven, fixedlocgiven: boolean;
   n, iteration: integer;
   starttime, timeelapsed, timeused, costoflayout: integer;
   noofpartitions, sizeofsubproblem: integer;


PROCEDURE readcostanddistancematrices;

   VAR
      i, j: location;
      l, m: activity;
      nolocfixed: integer;

   BEGIN
      reset(data);
      read(data, n);
      FOR i := 1 TO n DO
         FOR j := i TO n DO
            read(data, d[i, j]);
      FOR l := 1 TO n DO
         FOR m := 1 TO n DO
            read(data, w[l, m]);
 {complete the lower half of the matrices}
      FOR i := 1 TO n - 1 DO
         FOR j := i + 1 TO n DO
```

```
            d[j, i] := d[i, j];
      FOR l := 1 TO n - 1 DO
         FOR m := l + 1 TO n DO
            w[m, l] := w[l, m];
      read(data, noofpartitions);
      IF noofpartitions = 1
      THEN
         BEGIN
            FOR i := 1 TO n DO
               read(data, macinloc[i]);
            FOR i := 1 TO n DO
               locofmac[macinloc[i]] := i;
            read(data, nolocfixed);
            IF nolocfixed > 0
            THEN
               BEGIN
                  fixedlocgiven := true;
                  locationsfixed := [];
                  FOR i := 1 TO nolocfixed DO
                     BEGIN
                        read(data, j);
                        locationsfixed := locationsfixed + [j];
                     END;
               END
            ELSE
               BEGIN
                  fixedlocgiven := false;
                  locationsfixed := [];
               END;
         END;
   END {readcostanddistancematrices} ;


PROCEDURE writeoutput;

   VAR
      i: location;
      j: integer;

   BEGIN
      writeln(' FINAL LAYOUT COST ', costoflayout: 8);
      writeln(' NO OF ITERATION(S) ', iteration: 5);
      writeln(' EXECUTION TIME ', timeused: 6, ' MIL-SEC');
      writeln(' THE LAYOUT :');
      FOR i := 1 TO 4 DO
         write('  LOC  MAC  ');
      writeln;
      j := 0;
      FOR i := 1 TO n DO
         BEGIN
            write(i: 5, macinloc[i]: 5, '  ');
            j := j + 1;
            IF j = 4 THEN
               BEGIN
                  writeln;
                  j := 0;
               END;
         END;
      writeln;
   END {writeoutput} ;


PROCEDURE craft(n: integer; w: arrayweight; d: arraydistance;
locationsfixed: setoffixedlocations; VAR macinloc: arrayactinloc; VAR
locofmac: arraylacofact; VAR iteration, timeused, costoflayout:
integer);

VAR
   starttime, timeelapsed: integer;
   costofswitchmacinloc: arrayswitchcost;
   oldmacinloc: arrayactinloc;


PROCEDURE dumpinformation;

   VAR
      i, j: location;
      k: activity;

   BEGIN
      writeln(' EXCHANGE INFORMATION');
      writeln(' ITERATION(S)', iteration: 4, '  LAYOUT COST ',
         costoflayout: 6);
      FOR i := 1 TO n DO
         write(i: 4);
      writeln;
      FOR i := 1 TO n DO
         write(macinloc[i]: 4);
      writeln;
      FOR k := 1 TO n DO
         write(locofmac[k]: 4);
      writeln;
      writeln('  LOC  LOC   COST');
      FOR i := 1 TO n - 1 DO
         FOR j := i + 1 TO n DO
            writeln(i: 5, j: 5, costofswitchmacinloc[i, j]: 7);
   END {dumpinformation} ;


FUNCTION overalllayoutcost: integer;

   VAR
      i, j: activity;
      cost: integer;
      locofi: location;

   BEGIN
      cost := 0;
      FOR i := 1 TO n - 1 DO
         BEGIN
            locofi := locofmac[i];
            FOR j := i + 1 TO n DO
               cost := cost + w[i, j] * d[locofi, locofmac[j]];
         END;
      overalllayoutcost := cost;
   END {overalllayoutcost} ;


FUNCTION xchangecostforloc(l, m: location): integer;

   VAR
      macinl, macinm, macink: activity;
      k: location;
      cost: integer;

   BEGIN
```

```
      macinl := macinloc[l];
      macinm := macinloc[m];
      cost := 0;
      FOR k := 1 TO n DO
         BEGIN
            macink := macinloc[k];
            cost := cost + (d[l, k] - d[m, k]) * (w[macink, macinm] -
               w[macink, macinl]);
         END;
      xchangecostforloc := cost + 2 * w[macinl, macinm] * d[l, m];
   END {xchangecostforloc} ;


PROCEDURE keepoldmacinloc;

   VAR
      i: location;

   BEGIN
      FOR i := 1 TO n DO
         oldmacinloc[i] := macinloc[i];
   END {keepoldmacinloc} ;


PROCEDURE initpairwiseexchangecosts;

   VAR
      l, m: location;

   BEGIN
      FOR l := 1 TO n - 1 DO
         FOR m := l + 1 TO n DO
            costofswitchingmacinloc[l, m] := xchangecostforloc(l, m);
   END {initpairwiseexchangecosts} ;


PROCEDURE bestpair(VAR bestl, bestm: location; VAR largegain: integer
   );

   VAR
      l, m: location;
      gain: integer;

   BEGIN
      gain := - infinity;
      FOR l := 1 TO n - 1 DO
         IF NOT (l IN locationsfixed)
         THEN
            FOR m := l + 1 TO n DO
               IF NOT (m IN locationsfixed) THEN
                  IF - costofswitchmacinloc[l, m] > gain THEN
                     BEGIN
                        gain := - costofswitchmacinloc[l, m];
                        bestl := l;
                        bestm := m;
                     END;
      largegain := gain;
   END {bestpair} ;


PROCEDURE updatelocation(bestl, bestm: location);

   VAR
```

```
      previousmacinl: activity;

   BEGIN
      previousmacinl := macinloc[bestl];
      macinloc[bestl] := macinloc[bestm];
      macinloc[bestm] := previousmacinl;
      locofmac[macinloc[bestl]] := bestl;
      locofmac[macinloc[bestm]] := bestm;
   END {updatelocation} ;


PROCEDURE updatemarclos(i, j: location);

   VAR
      l, m: location;
      updatecost: integer;
      macini, macinj, macinl, macinm: activity;

   BEGIN
      macini := oldmacinloc[i];
      macinj := oldmacinloc[j];
      FOR l := 1 TO n - 1 DO
         IF NOT (l IN locationsfixed)
         THEN
            FOR m := l + 1 TO n DO
               IF NOT (m IN locationsfixed)
               THEN
                  IF (l = i) AND (m = j)
                  THEN
                     costofswitchmacinloc[l, m] := -
                        costofswitchmacinloc[l, m]
                  ELSE
                     IF ((l = i) OR (l = j)) OR ((m = i) OR (m = j))
                     THEN
                        costofswitchmacinloc[l, m] :=
                           xchangecostforloc(l, m)
                     ELSE
                        BEGIN
                           macinl := oldmacinloc[l];
                           macinm := oldmacinloc[m];
                           updatecost := (d[j, l] - d[i, l] + d[i, m]
                              - d[j, m]) * (w[macini, macinm] + w[
                              macinj, macinl] - w[macinj, macinm] - w
                              [macini, macinl]);
                           costofswitchmacinloc[l, m] :=
                              costofswitchmacinloc[l, m] + updatecost
                           ;
                        END;
   END {updatemarclos} ;


PROCEDURE pairwiseinterchange;

   VAR
      bestl, bestm: location;
      exchange: boolean;
      largegain: integer;

   BEGIN
      initpairwiseexchangecost;
      REPEAT
         bestpair(bestl, bestm, largegain);
         IF largegain > 0
```

```
            THEN
               BEGIN
                  exchange := true;
                  keepoldmacinloc;
                  updatelocation(bestl, bestm);
                  updatemarclos(bestl, bestm);
                  costoflayout := costoflayout - largegain;
                  iteration := iteration + 1;
               END
            ELSE
               exchange := false;
         UNTIL NOT exchange;
      END {pairwiseinterchange} ;


   BEGIN {craft}
      iteration := 0;
      starttime := clock;
      costoflayout := overalllayoutcost;
      pairwiseinterchange;
      timeelapsed := clock - starttime;
      timeused := timeelapsed;
   END {craft} ;


PROCEDURE readsubproblem;

   VAR
      i, j, l: location;
      k, nolocfixed: integer;
      found: boolean;

   BEGIN
      read(data, sizeofsubproblem);
      FOR k := 1 TO sizeofsubproblem DO
         read(data, oldlocname[k], oldmacname[k]);
      read(data, nolocfixed);
      locationsfixed := [];
      IF nolocfixed > 0
      THEN
         BEGIN
            fixedlocgiven := true;
            FOR i := 1 TO nolocfixed DO
               BEGIN
                  read(data, j);
                  l := 1;
                  found := false;
                  WHILE NOT (found OR (l > nolocfixed)) DO
                     BEGIN
                        IF j = oldlocname[l]
                        THEN
                           BEGIN
                              locationsfixed := locationsfixed + [l];
                              found := true;
                           END
                        ELSE
                           l := l + 1;
                     END;
               END;
         END
      ELSE
         fixedlocgiven := false;
   END {readsubproblem} ;
```

```


PROCEDURE constructsubproblem;

   VAR
      i, j, oldloci, oldlocj: location;
      l, m, oldmacl, oldmacm: activity;
      k: integer;

   BEGIN
      FOR i := 1 TO sizeofsubproblem DO
         BEGIN
            oldloci := oldlocname[i];
            FOR j := 1 TO sizeofsubproblem DO
               BEGIN
                  oldlocj := oldlocname[j];
                  dsubprob[i, j] := d[oldloci, oldlocj];
               END;
         END;
      FOR l := 1 TO sizeofsubproblem DO
         BEGIN
            oldmacl := oldmacname[l];
            FOR m := 1 TO sizeofsubproblem DO
               BEGIN
                  oldmacm := oldmacname[m];
                  weightsubprob[l, m] := w[oldmacl, oldmacm];
               END;
         END;
      FOR k := 1 TO sizeofsubproblem DO
         tempmacinloc[k] := k;
      FOR k := 1 TO sizeofsubproblem DO
         templocofmac[tempmacinloc[k]] := k;
   END  {constructsubproblem} ;


PROCEDURE partialreconstructofsubsolution;

   VAR
      k, oldnameoftempactk: activity;
      tempnameoflocofk: location;

   BEGIN
      FOR k := 1 TO sizeofsubproblem DO
         BEGIN
            oldnameoftempactk := oldmacname[k];
            tempnameoflocofk := templocofmac[k];
            locofmac[oldnameoftempactk] := oldlocname[tempnameoflocofk];
         END;
   END  {partialreconstructofsubsolution} ;


PROCEDURE reconstructionofsubsolutions;

   VAR
      i: activity;

   BEGIN
      FOR i := 1 TO n DO
         macinloc[locofmac[i]] := i;
      locationsfixed := [];
   END  {reconstructionofsubsolutions} ;


```

```
PROCEDURE reportonsubproblem;

   VAR
      i, j: integer;

   BEGIN
      writeln('          DISTANCE MATRIX');
      write(' ': 8);
      FOR i := 1 TO sizeofsubproblem DO
         write(i: 4);
      writeln;
      write(' ': 8);
      FOR i := 1 TO sizeofsubproblem DO
         write(oldlocname[i]: 4);
      writeln;
      FOR i := 1 TO sizeofsubproblem DO
         BEGIN
            write(i: 4, oldlocname[i]: 4);
            FOR j := 1 TO sizeofsubproblem DO
               write(dsubprob[i, j]: 4);
            writeln;
         END;
      writeln;
      writeln('          WEIGHT MATRIX');
      write(' ': 8);
      FOR i := 1 TO sizeofsubproblem DO
         write(i: 4);
      writeln;
      write(' ': 8);
      FOR i := 1 TO sizeofsubproblem DO
         write(oldmacname[i]: 4);
      writeln;
      FOR i := 1 TO sizeofsubproblem DO
         BEGIN
            write(i: 4, oldmacname[i]: 4);
            FOR j := 1 TO sizeofsubproblem DO
               write(weightsubproblem[i, j]: 4);
            writeln;
         END;
      writeln;
      writeln(' SUB-PROBLEM ASSIGNMENT  LOC-MAC: ');
      FOR i := 1 TO sizeofsubproblem DO
         write(oldlocname[i]: 4, oldmacname[tempmacinloc[i]]: 4, '  ');
      writeln;
      writeln(' GLOBAL ASSIGNMENT  MAC-LOC:');
      FOR i := 1 TO n DO
         write(i: 4, locofmac[i]: 4, '  ');
      writeln;
      writeln;
   END {reportonsubproblem} ;


PROCEDURE solvedbypartitioning;

   VAR
      i, tempiterno, temptime, tempcost: integer;

   BEGIN
      IF noofpartitions > 1
      THEN
         BEGIN
            FOR i := 1 TO noofpartitions DO
               BEGIN
```

```
                  readsubproblem;
                  constructsubproblem;
                  craft(sizeofsubproblem, weightsubprob, dsubprob,
                     locationsfixed, tempmacinloc, templocofmac,
                     tempiterno, temptime, tempcost);
                  partialreconstructofsubsolution;
 {reportonsubproblem;}
               END;
            reconstructionofsubsolutions;
         END;
      craft(n, w, d, locationsfixed, macinloc, locofmac, iteration,
         timeused, costoflayout);
   END {solvedbypartitioning} ;


BEGIN {layout3}
   readcostanddistancematrices;
   starttime := clock;
   solvedbypartitioning;
   timeelapsed := clock - starttime;
   writeoutput;
   writeln(' PARTITIONING OVERHEADS ', timeelapsed - timeused);
   writeln(' TOTAL TIME ', timeelapsed: 4);
   writeln;
END {layout3} .
```

```
PROGRAM maxplanar(tetra, output, seed, input /);
   (*$I'RANDOM'  random number generator declarations.  *)

CONST
   maxn = 100;
   { number of vertices }
   maxm = 294;
   { number of arcs 3*n - 6 }
   maxf = 196;
   { number of aces 2*n -4 }
   maxvalence = 99;
   { n-1 }
   maxnocoef = 4950;
   { n*(n-1)div2 }
   big = 9999;

TYPE
   noderange = 1..maxn;
   arcrange = 1..maxm;
   facerange = 1..maxf;
   small = 0..127;
   nodeptr = ∧ nodelist;
   arcptr = ∧ arcinuse;
   faceptr = ∧ faces;
   nodelist = PACKED RECORD
                        arcloc: arcptr;
                        nextnode: nodeptr;
                     END;
   verticesinuse = PACKED RECORD
                             value1, value2: integer;
                             face1, face2: faceptr;
                          END;
   activevertex = ∧ verticesinuse;
   anodetable = PACKED RECORD
                          CASE active: boolean OF
                             true: (vactive: activevertex);
                             false: (valence: 0..maxvalence;
                                     nextvertex: nodeptr)
                       END;
   arcinuse = PACKED RECORD
                        n1, n2: noderange;
                        f1, f2: faceptr;
                        arcadj: arcptr;
                     END;
   faces = PACKED RECORD
                     v1, v2, v3: noderange;
                     faceadj: faceptr;
                  END;
   start =
      (maxweight, maxtetra, randomized);
   entry =
      (ordered, largest, delta);

VAR
   seed, tetra: text;
   nodetable: ARRAY
      [1..maxn] OF anodetable;
   newarc, firstarc, lastarc: arcptr;
   relchart: ARRAY
      [1..maxnocoef] OF small;
   newface, firstface, fnxtolast, lastface: faceptr;
   activenode, firstactivenode: activevertex;
   nextvertex, nodestore: nodeptr;
   shape: ARRAY
      [1..24] OF 1..6;
   sumw: ARRAY
      [0..maxn] OF PACKED RECORD
                             v: 0..maxn;
                             g: integer;
                          END;
   n, nv: 0..maxn;
   m, na: 0..maxm;
   f, nf: 0..maxf;
   nocoef: 1..maxnocoef;
   fremoved: faceptr;
   i, problem, timet, timec, timei: integer;
   anode: noderange;
   starting: start;
   enter: entry;
   firstround, arcswap, yswap: boolean;


PROCEDURE order2(VAR x, y: noderange);

   VAR
      z: noderange;

   BEGIN
      IF y < x THEN
         BEGIN
            z := x;
            x := y;
            y := z
         END
   END  {order2} ;


PROCEDURE order3(VAR x, y, z: noderange);

   BEGIN
      order2(x, y);
      order2(y, z);
      order2(x, y)
   END  {order3} ;


FUNCTION c(i, j: noderange): small;

   VAR
      k: 0..maxnocoef;
      i1, j1: noderange;

   BEGIN
      IF i = j
      THEN
         c := 0
      ELSE
         BEGIN
            i1 := i;
            j1 := j;
            order2(i1, j1);
            k := (i1 - 1) * n - (i1 - 1) * i1 DIV 2;
            c := relchart[k + j1 - i1]
         END
   END  {c} ;

```

```

FUNCTION assigncost: integer;

   VAR
      ptr: arcptr;
      cost: integer;
      i, j: noderange;

   BEGIN
      ptr := firstarc;
      cost := 0;
      WHILE ptr <> NIL DO
         BEGIN
            WITH ptr ^ DO
               BEGIN
                  i := n1;
                  j := n2;
               END;
            cost := cost + c(i, j);
            ptr := ptr ^.arcadj
         END;
      assigncost := cost
   END {assigncost} ;


FUNCTION starweight(v1, v2, v3, v4: noderange): integer;

   BEGIN
      starweight := c(v1, v2) + c(v1, v3) + c(v1, v4) + c(v2, v3) + c(v2
         , v4) + c(v3, v4)
   END {starweight} ;


FUNCTION yweight(v1, v2, v3, v4: noderange): integer;

   BEGIN
      yweight := c(v1, v2) + c(v1, v3) + c(v1, v4)
   END {yweight} ;


FUNCTION pickorder: noderange;

   BEGIN
      pickorder := sumw[nv + 1].v
   END {pickorder} ;


PROCEDURE readinput;

   VAR
      i: integer;

   BEGIN
      read(tetra, n, problem);
      FOR i := 1 TO n * (n - 1) DIV 2 DO
         read(tetra, relchart[i]);
      FOR i := 1 TO 24 DO
         read(tetra, shape[i]);
   END {readinput} ;


PROCEDURE initrandom;

```

```
   VAR
      s1, s2: integer;

   BEGIN
      reset(seed);
      read(seed, s1, s2);
      setrandom(s1, s2);
      writeln(' SEEDS USED: ', s1: 20, s2: 20);
   END {initrandom} ;


PROCEDURE replaceseeds;

   VAR
      s1, s2: integer;

   BEGIN
      rewrite(seed);
      getrandom(s1, s2);
      write(seed, s1, ' ', s2);
   END {replaceseeds} ;


PROCEDURE initialization;

   VAR
      i: integer;
      p: activevertex;

   BEGIN
      m := 3 * n - 6;
      f := 2 * n - 4;
      nocoef := n * (n - 1) DIV 2;
      FOR i := 1 TO n DO
         WITH nodetable[i] DO
            BEGIN
               active := true;
               new(p);
               vactive := p;
               WITH vactive ^ DO
                  BEGIN
                     value1 := 0;
                     value2 := 0;
                     face1 := NIL;
                     face2 := NIL;
                  END;
            END;
      IF enter = ordered THEN
         BEGIN
            FOR i := 1 TO n DO
               WITH sumw[i] DO
                  BEGIN
                     v := 0;
                     g := 0;
                  END;
            sumw[0].g := big;
         END;
      nextvertex := NIL;
      nodestore := NIL;
      firstface := NIL;
      lastface := NIL;
      fnxtolast := NIL;
      nv := 0;
      na := 0;
      nf := 0;
   END {initialization} ;


PROCEDURE garbagecollection;

   VAR
      p1, p2: faceptr;
      p3, p4: arcptr;
      p5, p6: nodeptr;
      i: integer;

   BEGIN
      p1 := firstface;
      WHILE p1 <> NIL DO
         BEGIN
            p2 := p1 ^.faceadj;
            dispose(p1);
            p1 := p2
         END;
      p3 := firstarc;
      WHILE p3 <> NIL DO
         BEGIN
            p4 := p3 ^.arcadj;
            dispose(p3);
            p3 := p4
         END;
      FOR i := 1 TO n DO
         BEGIN
            p5 := nodetable[i].nextvertex;
            WHILE p5 <> NIL DO
               BEGIN
                  p6 := p5 ^.nextnode;
                  dispose(p5);
                  p5 := p6;
               END;
         END;
   END {garbagecollection} ;


PROCEDURE deactivate(v: noderange);

   VAR
      p: activevertex;

   BEGIN
      WITH nodetable[v] DO
         BEGIN
            p := vactive;
            dispose(p);
            active := false;
            valence := 0;
            nextvertex := NIL;
         END;
   END {deactivate} ;


PROCEDURE intermediateresults;

   VAR
      i: integer;
      ptr: nodeptr;

   BEGIN
      FOR i := 1 TO n DO
         WITH nodetable[i] DO
            BEGIN
               IF active
               THEN
                  BEGIN
                     writeln(' NODE ', i: 3);
                     WITH vactive ∧ DO
                        BEGIN
                           IF value1 <> 0 THEN
                              WITH face1 ∧ DO
                                 writeln(' VALUE', value1: 5, v1: 3, v2:
                                    3, v3: 3);
                           IF value2 <> 0 THEN
                              WITH face2 ∧ DO
                                 writeln(' VALUE', value2: 5, v1: 3, v2:
                                    3, v3: 3)
                        END;
                  END
               ELSE
                  BEGIN
                     writeln(' NODE', i: 4, ' VALENCE ', valence: 4);
                     ptr := nextvertex;
                     WHILE ptr <> NIL DO
                        BEGIN
                           WITH ptr ∧, arcloc ∧ DO
                              writeln(' ARC ', n1: 3, n2: 3, ' FACE1 ',
                                 f1 ∧.v1: 3, f1 ∧.v2: 3, f1 ∧.v3: 3,
                                 ' FACE2 ', f2 ∧.v1: 3, f2 ∧.v2: 3, f2 ∧
                                 .v3: 3);
                           ptr := ptr ∧.nextnode;
                        END;
                  END;
               writeln;
            END;
   END {intermediateresults} ;


PROCEDURE insertinformation(k: noderange);

   VAR
      n1, n2, n3: noderange;

   BEGIN
      WITH nodetable[k].vactive ∧.face1 ∧ DO
         BEGIN
            n1 := v1;
            n2 := v2;
            n3 := v3
         END;
      writeln(' PUT NODE ', k: 3, ' INTO FACE ', n1: 3, n2: 3, n3: 3);
   END {insertinformation} ;


PROCEDURE statusreport;

   BEGIN
      writeln(' NUMBER OF VERTICES ', n: 5);
      writeln(' PROBLEM NUMBER ', problem: 5);
      CASE starting OF
         maxweight:
            writeln(' FOUR HEIGHEST WEIGHT VERTICES AS',
               ' STARTING TETRAHEDRON');
         maxtetra:
            writeln(' HEAVIEST TETRAHEDRON AS STARTING POINT');
         randomized:
            writeln(' RANDOM STARTING TETRAHEDRON')
      END;
      write(' NODE SELECTION ACCORDING TO ');
      CASE enter OF
         ordered:
            writeln(' WEIGHT ORDER');
         largest:
            writeln(' HIGHEST GAIN');
         delta:
            writeln(' HIGHEST COST')
      END;
   END {statusreport} ;


PROCEDURE bigtetra(VAR v1, v2, v3, v4: noderange);

   VAR
      i, j, k, l: noderange;
      base, weight: integer;

   BEGIN
      base := 0;
      FOR i := 1 TO n - 3 DO
         FOR j := i + 1 TO n - 2 DO
            FOR k := j + 1 TO n - 1 DO
               FOR l := k + 1 TO n DO
                  BEGIN
                     weight := starweight(i, j, k, l);
                     IF base <= weight THEN
                        BEGIN
                           base := weight;
                           v1 := i;
                           v2 := j;
                           v3 := k;
                           v4 := l;
                        END;
                  END
   END {bigtetra} ;


PROCEDURE random4nodes(VAR n1, n2, n3, n4: noderange);

   VAR
      anode: ARRAY
         [1..4] OF noderange;
      k: noderange;
      i, j: integer;
      same: boolean;

   BEGIN
      anode[1] := trunc(random * n) + 1;
      FOR i := 2 TO 4 DO
         BEGIN
            REPEAT
               same := false;
               k := trunc(random * n) + 1;
               FOR j := 1 TO i - 1 DO
                  IF anode[j] = k THEN
```

```
                     same := true;
            UNTIL NOT same;
            anode[i] := k;
         END;
      FOR i := 2 TO 4 DO
         FOR j := 4 DOWNTO i DO
            IF anode[j] < anode[j - 1] THEN
               BEGIN
                  k := anode[j - 1];
                  anode[j - 1] := anode[j];
                  anode[j] := k;
               END;
      n1 := anode[1];
      n2 := anode[2];
      n3 := anode[3];
      n4 := anode[4];
   END {random4nodes} ;


PROCEDURE longtable(i: noderange; val: integer);

   VAR
      j, k: integer;

   BEGIN
      j := i - 1;
      WHILE sumw[j].g < val DO
         BEGIN
            sumw[j + 1] := sumw[j];
            j := j - 1;
         END;
      WITH sumw[j + 1] DO
         BEGIN
            v := i;
            g := val
         END;
      IF i = n
      THEN
         FOR j := 4 DOWNTO 2 DO
            FOR k := j - 1 DOWNTO 1 DO
               IF sumw[j].v < sumw[k].v THEN
                  BEGIN
                     sumw[0] := sumw[j];
                     sumw[j] := sumw[k];
                     sumw[k] := sumw[0];
                  END;
   END {longtable} ;


PROCEDURE select4nodes(VAR v1, v2, v3, v4: noderange);

   VAR
      a: ARRAY
         [0..4] OF RECORD
                     v: 0..maxn;
                     g: integer
                  END;
      attractive, i, j: integer;


   PROCEDURE sorttable;

      VAR
```

```
         i, j: integer;

      BEGIN
         FOR i := 4 DOWNTO 2 DO
            FOR j := i - 1 DOWNTO 1 DO
               IF a[i].v < a[j].v THEN
                  BEGIN
                     a[0] := a[i];
                     a[i] := a[j];
                     a[j] := a[0];
                  END;
      END  {sorttable} ;


   PROCEDURE upthetable(i: noderange; val: integer);

      VAR
         j: 0..4;

      BEGIN
         j := 4;
         WHILE a[j].g < val DO
            BEGIN
               a[j] := a[j - 1];
               j := j - 1;
            END;
         IF j <> 4 THEN
            WITH a[j + 1] DO
               BEGIN
                  v := i;
                  g := val;
               END;
         IF i = n THEN
            sorttable;
      END  {upthetable} ;


   BEGIN  {select4nodes}
      IF starting = maxweight
      THEN
         BEGIN
            FOR i := 0 TO 4 DO
               WITH a[i] DO
                  BEGIN
                     v := 0;
                     g := 0;
                  END;
            a[0].g := big;
            FOR i := 1 TO n DO
               BEGIN
                  attractive := 0;
                  FOR j := 1 TO n DO
                     IF i <> j THEN
                        attractive := attractive + c(i, j);
                  IF enter = ordered
                  THEN
                     longtable(i, attractive)
                  ELSE
                     upthetable(i, attractive)
               END;
            IF enter = ordered
            THEN
               BEGIN
```

```
                  v1 := sumw[1].v;
                  v2 := sumw[2].v;
                  v3 := sumw[3].v;
                  v4 := sumw[4].v
               END
            ELSE
               BEGIN
                  v1 := a[1].v;
                  v2 := a[2].v;
                  v3 := a[3].v;
                  v4 := a[4].v;
               END;
         END
      ELSE
         IF starting = maxtetra
         THEN
            bigtetra(v1, v2, v3, v4)
         ELSE
            random4nodes(v1, v2, v3, v4);
   END  {select4nodes} ;


PROCEDURE tetrahedron;

   VAR
      v: ARRAY
         [1..4] OF noderange;
      i: 1..4;
      j: integer;


   PROCEDURE maketetrahedron;

      VAR
         i, j, k: 0..maxn;
         l, p: integer;
         newnode, nptr: nodeptr;
         e: ARRAY
            [1..6] OF arcptr;
         s: ARRAY
            [1..4] OF faceptr;

      BEGIN
         p := 0;
         FOR l := 1 TO 6 DO
            new(e[l]);
         FOR l := 1 TO 4 DO
            new(s[l]);
         { construct the node list}
         FOR i := 1 TO 4 DO
            BEGIN
               nptr := NIL;
               deactivate(v[i]);
               FOR j := 3 DOWNTO 1 DO
                  BEGIN
                     new(newnode);
                     newnode ∧.nextnode := nptr;
                     newnode ∧.arcloc := e[shape[p + j]];
                     nptr := newnode;
                  END;
               nodetable[v[i]].valence := 3;
               nodetable[v[i]].nextvertex := nptr;
               p := p + 3
            END;
         {construct nodetable}
         l := 1;
         FOR i := 1 TO 3 DO
            FOR j := i + 1 TO 4 DO
               BEGIN
                  WITH e[l] ∧ DO
                     BEGIN
                        n1 := v[i];
                        n2 := v[j];
                        f1 := s[shape[p + 1]];
                        f2 := s[shape[p + 2]];
                     END;
                  l := l + 1;
                  p := p + 2;
               END;
         firstarc := e[1];
         e[6] ∧.arcadj := NIL;
         lastarc := e[6];
         FOR i := 1 TO 5 DO
            e[i] ∧.arcadj := e[i + 1];
         {construct face}
         l := 1;
         FOR i := 1 TO 2 DO
            FOR j := i + 1 TO 3 DO
               FOR k := j + 1 TO 4 DO
                  BEGIN
                     WITH s[l] ∧ DO
                        BEGIN
                           v1 := v[i];
                           v2 := v[j];
                           v3 := v[k];
                        END;
                     l := l + 1;
                  END;
         firstface := s[1];
         lastface := s[4];
         FOR i := 1 TO 3 DO
            s[i] ∧.faceadj := s[i + 1];
         s[4] ∧.faceadj := NIL;
         nv := 4;
         na := 6;
         nf := 4;
      END {maketetrahedron} ;


   BEGIN {tetrahedron}
      select4nodes(v[1], v[2], v[3], v[4]);
      writeln(' INITIAL TETRAHEDRON ', v[1]: 4, v[2]: 4, v[3]: 4, v[4]:
         4);
      maketetrahedron;
   END {tetrahedron} ;


FUNCTION facevalue(v: noderange; f: faces): integer;

   BEGIN
      WITH f DO
         facevalue := c(v, v1) + c(v, v2) + c(v, v3);
   END {facevalue} ;


PROCEDURE savebig2(i: noderange; f: faceptr; value0: integer);
```

```
   BEGIN
      WITH nodetable[i].vactive ^ DO
         IF value2 < value0
         THEN
            IF value1 < value0
            THEN
               BEGIN
                  value2 := value1;
                  face2 := face1;
                  value1 := value0;
                  face1 := f;
               END
            ELSE
               BEGIN
                  value2 := value0;
                  face2 := f;
               END;
   END  {savebig2} ;


PROCEDURE nodegain(v: noderange);

   VAR
      ptr: faceptr;
      i: facerange;

   BEGIN
      IF nodetable[v].active
      THEN
         WITH nodetable[v].vactive ^ DO
            BEGIN
               ptr := firstface;
               FOR i := 1 TO nf DO
                  BEGIN
                     savebig2(v, ptr, facevalue(v, ptr ^));
                     ptr := ptr ^.faceadj
                  END;
            END
   END  {nodegain} ;


PROCEDURE gainupdate(v: noderange);

   VAR
      ptr: faceptr;
      i: facerange;

   BEGIN
      IF nodetable[v].active
      THEN
         WITH nodetable[v].vactive ^ DO
            BEGIN
               IF ((face1 = fremoved) OR (face2 = fremoved))
               THEN
                  BEGIN
                     value1 := 0;
                     value2 := 0;
                     nodegain(v)
                  END
               ELSE
                  BEGIN
                     savebig2(v, fremoved, facevalue(v, fremoved ^));
                     savebig2(v, fnxtolast, facevalue(v, fnxtolast ∧));
                     savebig2(v, lastface, facevalue(v, lastface ∧));
                  END;
            END;
   END {gainupdate} ;


FUNCTION pick1: noderange;

   VAR
      a, i: noderange;
      base: integer;

   BEGIN
      base := 0;
      FOR i := 1 TO n DO
         WITH nodetable[i] DO
            IF active THEN
               IF vactive ∧.value1 >= base THEN
                  BEGIN
                     base := vactive ∧.value1;
                     a := i;
                  END;
      pick1 := a
   END {pick1} ;


FUNCTION pick2: noderange;

   VAR
      a, i: noderange;
      base: integer;

   BEGIN
      base := 0;
      FOR i := 1 TO n DO
         WITH nodetable[i] DO
            IF active THEN
               WITH vactive ∧ DO
                  IF value1 - value2 >= base THEN
                     BEGIN
                        base := value1 - value2;
                        a := i;
                     END;
      pick2 := a
   END {pick2} ;


PROCEDURE addaface(nd1, nd2, nd3: noderange; location: faceptr);

   VAR
      n1, n2, n3: noderange;

   BEGIN
      n1 := nd1;
      n2 := nd2;
      n3 := nd3;
      order3(n1, n2, n3);
      WITH location ∧ DO
         BEGIN
            v1 := n1;
            v2 := n2;
            v3 := n3;
         END;
   END  {addaface} ;


PROCEDURE addanarc(nd1, nd2: noderange; a: arcptr; l1, l2: faceptr);

   VAR
      v1, v2: noderange;

   BEGIN
      v1 := nd1;
      v2 := nd2;
      order2(v1, v2);
      WITH a ∧ DO
         BEGIN
            n1 := v1;
            n2 := v2;
            f1 := l1;
            f2 := l2;
         END;
   END  {addanarc} ;


PROCEDURE addavertex(nd1, nd2: noderange; a1: arcptr);

   VAR
      this, next, ptr: nodeptr;
      a2: arcptr;
      nd: noderange;
      found: boolean;

   BEGIN
      new(ptr);
      WITH nodetable[nd1] DO
         BEGIN
            IF active
            THEN
               BEGIN
                  deactivate(nd1);
                  valence := 1;
                  nextvertex := ptr;
                  ptr ∧.arcloc := a1;
                  ptr ∧.nextnode := NIL;
               END
            ELSE
               BEGIN
                  this := NIL;
                  next := nextvertex;
                  found := false;
                  WHILE ((NOT found) AND (next <> NIL)) DO
                     BEGIN
                        a2 := next ∧.arcloc;
                        IF nd1 = a2 ∧.n1
                        THEN
                           nd := a2 ∧.n2
                        ELSE
                           nd := a2 ∧.n1;
                        IF nd > nd2
                        THEN
                           BEGIN
                              found := true;
                              IF this = NIL
                              THEN
```

```
                                 BEGIN
                                    ptr ^.nextnode := nextvertex;
                                    nextvertex := ptr;
                                 END
                              ELSE
                                 this ^.nextnode := ptr
                           END
                        ELSE
                           BEGIN
                              this := next;
                              next := next ^.nextnode;
                           END;
                     END;
                  IF next = NIL
                  THEN
                     BEGIN
                        IF this = NIL
                        THEN
                           nextvertex := ptr
                        ELSE
                           this ^.nextnode := ptr;
                        ptr ^.nextnode := NIL
                     END
                  ELSE
                     ptr ^.nextnode := next;
                  ptr ^.arcloc := a1;
                  valence := valence + 1;
               END
         END
   END {addavertex} ;


PROCEDURE changefaces(nd1, nd2, nd3: noderange; nf1, nf2: faceptr);


   PROCEDURE findarc(nd1, nd2: noderange; f1: faceptr);

      VAR
         v1, v2: noderange;
         l: arcptr;

      BEGIN
         v1 := nd1;
         v2 := nd2;
         order2(v1, v2);
         l := firstarc;
         WHILE ((l ^.n1 <> v1) OR (l ^.n2 <> v2)) DO
            l := l ^.arcadj;
         IF l ^.f1 = fremoved
         THEN
            l ^.f1 := f1
         ELSE
            l ^.f2 := f1
      END {findarc} ;


   BEGIN {changefaces}
      findarc(nd1, nd3, nf1);
      findarc(nd2, nd3, nf2)
   END {changefaces} ;


PROCEDURE adjface(v1, v2: noderange; fptr: faceptr);
```

```

   VAR
      anode: nodeptr;

   BEGIN
      anode := nodetable[v2].nextvertext;
      WHILE ((anode ∧.arcloc ∧.n1 <> v1) OR (anode ∧.arcloc ∧.n2 <> v2))
         DO
         anode := anode ∧.nextnode;
      WITH anode ∧.arcloc ∧ DO
         IF f1 = fremoved
         THEN
            f1 := fptr
         ELSE
            f2 := fptr
   END {adjface} ;


PROCEDURE addanode(stick: noderange; reject: faceptr);

   VAR
      i: integer;
      newnode, ptr: nodeptr;
      nf1, nf2: faceptr;
      n0, n1, n2, n3: noderange;
      a1, a2, a3: arcptr;

   BEGIN
      fremoved := reject;
      n0 := stick;
      WITH fremoved ∧ DO
         BEGIN
            n1 := v1;
            n2 := v2;
            n3 := v3;
         END;
      { enter new faces }
      addaface(n0, n1, n2, fremoved);
      new(nf1);
      addaface(n0, n1, n3, nf1);
      new(nf2);
      addaface(n0, n2, n3, nf2);
      adjface(n1, n3, nf1);
      adjface(n2, n3, nf2);
      lastface ∧.faceadj := nf1;
      nf2 ∧.faceadj := NIL;
      nf1 ∧.faceadj := nf2;
      fnxtolast := nf1;
      lastface := nf2;
      { enter new arcs }
      new(a1);
      new(a2);
      new(a3);
      addanarc(n0, n1, a1, fremoved, nf1);
      addanarc(n0, n2, a2, fremoved, nf2);
      addanarc(n0, n3, a3, nf1, nf2);
      lastarc ∧.arcadj := a1;
      a1 ∧.arcadj := a2;
      a2 ∧.arcadj := a3;
      a3 ∧.arcadj := NIL;
      lastarc := a3;
      { enter new vertex }
      addavertex(n1, n0, a1);
      addavertex(n2, n0, a2);
      addavertex(n3, n0, a3);
      addavertex(n0, n1, a1);
      addavertex(n0, n2, a2);
      addavertex(n0, n3, a3);
      { update indicies }
      nf := nf + 2;
      na := na + 3;
      nv := nv + 1;
   END {addanode} ;


FUNCTION switchable(anarc: arcptr): boolean;

   BEGIN
      WITH anarc ^ DO
         IF ((nodetable[n1].valence = 3) OR (nodetable[n2].valence = 3))
         THEN
            switchable := false
         ELSE
            switchable := true
   END {switchable} ;


FUNCTION thirdnode(anarc: arcptr; aface: faceptr): noderange;

   BEGIN
      WITH anarc ^, aface ^ DO
         IF ((v1 <> n1) AND (v1 <> n2))
         THEN
            thirdnode := v1
         ELSE
            IF ((v2 <> n1) AND (v2 <> n2))
            THEN
               thirdnode := v2
            ELSE
               thirdnode := v3
   END {thirdnode} ;


FUNCTION connected(a1, a2: noderange): arcptr;

   VAR
      v1, v2: noderange;
      vptr: nodeptr;
      found: boolean;

   BEGIN
      v1 := a1;
      v2 := a2;
      order2(v1, v2);
      found := false;
      vptr := nodetable[v2].nextvertex;
      WHILE ((NOT found) AND (vptr <> NIL)) DO
         WITH vptr ^.arcloc ^ DO
            IF v1 <> n1
            THEN
               vptr := vptr ^.nextnode
            ELSE
               found := true;
      IF found
      THEN
         connected := vptr ^.arcloc
      ELSE
         connected := NIL;
      { IF found THEN writeln( v1, v2, ' connected')
                  ELSE writeln( v1, v2, ' not connected'); }
   END {connected} ;


PROCEDURE removearc(p, q: noderange; anarc: arcptr);


   PROCEDURE removenode(n1: noderange; anarc: arcptr);

      VAR
         last, this: nodeptr;

      BEGIN
         this := nodetable[n1].nextvertex;
         last := NIL;
         WHILE this ∧.arcloc <> anarc DO
            BEGIN
               last := this;
               this := this ∧.nextnode;
            END;
         IF last = NIL
         THEN
            nodetable[n1].nextvertex := this ∧.nextnode
         ELSE
            last ∧.nextnode := this ∧.nextnode;
         dispose(this);
         nodetable[n1].valence := nodetable[n1].valence - 1;
      END {removenode} ;


   BEGIN {removearc}
      removenode(p, anarc);
      removenode(q, anarc);
   END {removearc} ;


PROCEDURE diagonalswitch(a1, a2, p, q: noderange; anarc: arcptr; fptr1,
   fptr2: faceptr);

   VAR
      dumarc1, dumarc2: arcptr;

   BEGIN
      dumarc1 := connected(a1, q);
      dumarc2 := connected(a2, p);
      addaface(a1, a2, p, fptr1);
      addaface(a1, a2, q, fptr2);
      addanarc(a1, a2, anarc, fptr1, fptr2);
      addavertex(a1, a2, anarc);
      addavertex(a2, a1, anarc);
      WITH dumarc1 ∧ DO
         IF f1 = fptr1
         THEN
            f1 := fptr2
         ELSE
            f2 := fptr2;
      WITH dumarc2 ∧ DO
         IF f1 = fptr2
         THEN
            f1 := fptr1
         ELSE
            f2 := fptrl;
      removearc(p, q, anarc);
   END  {diagonalswitch} ;


PROCEDURE redirectface(dl, d2: noderange; oldface, newface: faceptr);

   VAR
      dumarc: arcptr;

   BEGIN
      dumarc := connected(dl, d2);
      WITH dumarc ∧ DO
         IF fl = oldface
         THEN
            fl := newface
         ELSE
            f2 := newface
   END  {redirectface} ;


FUNCTION locatearc(dl, d2: noderange): arcptr;

   VAR
      anode: nodeptr;
      ndl, nd2: noderange;

   BEGIN
      ndl := dl;
      nd2 := d2;
      order2(ndl, nd2);
      anode := nodetable[nd2].nextvertex;
      WHILE NOT (anode ∧.arcloc ∧.nl = ndl) DO
         anode := anode ∧.nextnode;
      locatearc := anode ∧.arcloc;
   END  {locatearc} ;


FUNCTION locateface(dl, d2, d3: noderange): faceptr;

   VAR
      anarc: arcptr;
      ndl, nd2, nd3: noderange;

   BEGIN
      ndl := dl;
      nd2 := d2;
      nd3 := d3;
      order3(ndl, nd2, nd3);
      anarc := locatearc(ndl, nd3);
      WITH anarc ∧ DO
         IF fl ∧.v2 = nd2
         THEN
            locateface := fl
         ELSE
            locateface := f2;
   END  {locateface} ;


FUNCTION nonchangeablepair(nc, nd, nb, nal, na2: noderange): noderange;

   VAR
```

```
      aface: faceptr;
      anarc: arcptr;
      anode: noderange;

   BEGIN
      aface := locateface(nc, nd, nb);
      anarc := locatearc(nc, nb);
      REPEAT
         WITH anarc ^ DO
            IF f1 <> aface
            THEN
               aface := f1
            ELSE
               aface := f2;
         anode := thirdnode(anarc, aface);
         anarc := locatearc(nc, anode);
      UNTIL (anode = na1) OR (anode = na2);
      IF anode = na1
      THEN
         nonchangeablepair := na1
      ELSE
         nonchangeablepair := na2;
   END  {nonchangeablepair} ;


PROCEDURE mediumswitch(na2, nb1, na1, nb2, nc, nd: noderange);
     { replace na1-na2 by na2-nb1
       nc-nd are the other pair of vertices in the
       switching quadrilateral na1-nc-na2-nd
       nc is used as the anchor for searching }

   VAR
      r1, r2, r3: faceptr;
      anarc: arcptr;

   BEGIN
      r1 := locateface(na1, na2, nc);
      r2 := locateface(na1, na2, nd);
      r3 := locateface(nb1, nc, nd);
      addaface(na2, nb1, nc, r1);
      addaface(na2, nb1, nd, r2);
      addaface(na1, nc, nd, r3);
      redirectface(na1, nc, r1, r3);
      redirectface(na1, nd, r2, r3);
      redirectface(nb1, nc, r3, r1);
      redirectface(nb1, nd, r3, r2);
      anarc := locatearc(na1, na2);
      removearc(na1, na2, anarc);
      addanarc(na2, nb1, anarc, r1, r2);
      addavertex(nb1, na2, anarc);
      addavertex(na2, nb1, anarc);
      writeln(' MEDIUM SWITCH :', na1: 3, na2: 3, ' TO ', na2: 3, nb1: 3
         );
   END  {mediumswitch} ;


PROCEDURE switch(anarc: arcptr; VAR arcswap: boolean);

   TYPE
      replacetype =
         (noswitch, switcha2b1, switcha1b2, longleg);

   VAR
```

```
      a1, a2, b1, b2, c1, c2, anode: noderange;
      fptr1, fptr2, fptr3, fptr4: faceptr;
      joinedbase: arcptr;
      bestmove: replacetype;


   FUNCTION findswitch(w1, w2, w3, w4: integer): replacetype;

      VAR
         a: ARRAY
            [replacetype] OF integer;
         max: integer;
         i, kind: replacetype;

      BEGIN
         a[noswitch] := w1;
         a[switcha2b1] := w2;
         a[switcha1b2] := w3;
         a[longleg] := w4;
         max := w1;
         kind := noswitch;
         FOR i := switcha2b1 TO longleg DO
            IF a[i] > max THEN
               BEGIN
                  max := a[i];
                  kind := i;
               END;
         findswitch := kind;
      END  {findswitch} ;


   BEGIN  {switch}
      IF switchable(anarc)
      THEN
         BEGIN
            WITH anarc ^ DO
               BEGIN
                  fptr1 := f1;
                  fptr2 := f2;
                  a1 := n1;
                  a2 := n2;
                  c1 := thirdnode(anarc, fptr1);
                  c2 := thirdnode(anarc, fptr2);
               END;
            joinedbase := connected(c1, c2);
            IF joinedbase = NIL
            THEN
               BEGIN
                  IF c(c1, c2) > c(a1, a2)
                  THEN
                     BEGIN
                        writeln(' SWITCH ', a1: 3, a2: 3, ' TO ', c1: 3,
                           c2: 3);
                        diagonalswitch(c1, c2, a1, a2, anarc, fptr1,
                           fptr2);
                        arcswap := true;
                     END
               END
            ELSE
               BEGIN
                  fptr3 := joinedbase ^.f1;
                  fptr4 := joinedbase ^.f2;
                  b1 := thirdnode(joinedbased, fptr3);
                  b2 := thirdnode(joinedbase, fptr4);
                  anode := nonchangeablepair(c1, c2, b1, a1, a2);
                  IF anode <> a1 THEN
                     BEGIN
                        a2 := a1;
                        a1 := anode;
                     END;
                  bestmove := findswitch(c(a1, a2), c(a2, b1), c(a1, b2)
                     , c(b1, b2));
                  CASE bestmove OF
                     noswitch:
                        BEGIN
                        END;
                     switcha2b1:
                        mediumswitch(a2, b1, a1, b2, c1, c2);
                     switcha1b2:
                        mediumswitch(a1, b2, a2, b1, c1, c2);
                     longleg:
                        BEGIN
                           writeln(' LONGSWITCH ', a1: 3, a2: 3, ' TO ',
                              b1: 3, b2: 3);
                           diagonalswitch(b1, b2, c1, c2, joinedbase,
                              fptr3, fptr4);
                           diagonalswitch(c1, c2, a1, a2, anarc, fptr1,
                              fptr2);
                        END
                  END;
                  IF bestmove <> noswitch THEN
                     arcswap := true;
               END;
         END;
   END {switch} ;


PROCEDURE get3faces(anode: noderange; VAR face1, face2, face3: faceptr);

   VAR
      nptr: nodeptr;

   BEGIN
      nptr := nodetable[anode].nextvertex;
      WITH nptr ^.arcloc ^ DO
         BEGIN
            face1 := f1;
            face2 := f2;
         END;
      nptr := nptr ^.nextnode;
      WITH nptr ^.arcloc ^ DO
         IF ((f1 = face1) OR (f1 = face2))
         THEN
            face3 := f2
         ELSE
            face3 := f1;
   END {get3faces} ;


FUNCTION otherend(k: noderange; anarc: arcptr): noderange;

   BEGIN
      WITH anarc ^ DO
         IF (k = n1)
         THEN
            otherend := n2
         ELSE
            otherend := n1
   END {otherend} ;


PROCEDURE ychange(anode: noderange; r1, r2, r3, inface: faceptr);

   VAR
      b1, b2, b3, d1, d2, d3: noderange;
      a1, a2, a3: arcptr;
      nptr: nodeptr;

   BEGIN
      WITH inface ^ DO
         BEGIN
            d1 := v1;
            d2 := v2;
            d3 := v3;
         END;
      nptr := nodetable[anode].nextvertex;
      a1 := nptr ^.arcloc;
      nptr := nptr ^.nextnode;
      a2 := nptr ^.arcloc;
      nptr := nptr ^.nextnode;
      a3 := nptr ^.arcloc;
      b1 := otherend(anode, a1);
      b2 := otherend(anode, a2);
      b3 := otherend(anode, a3);
      WITH a1 ^ DO
         IF b2 = thirdnode(a1, f1)
         THEN
            BEGIN
               r1 := f1;
               r2 := f2;
            END
         ELSE
            BEGIN
               r1 := f2;
               r2 := f1;
            END;
      WITH a2 ^ DO
         IF b3 = thirdnode(a2, f1)
         THEN
            r3 := f1
         ELSE
            r3 := f2;
      redirectface(b1, b2, r1, inface);
      redirectface(b1, b3, r2, inface);
      redirectface(b2, b3, r3, inface);
      redirectface(d1, d2, inface, r1);
      redirectface(d1, d3, inface, r2);
      redirectface(d2, d3, inface, r3);
      removearc(anode, b1, a1);
      removearc(anode, b2, a2);
      removearc(anode, b3, a3);
      addaface(b1, b2, b3, inface);
      addaface(anode, d1, d2, r1);
      addaface(anode, d1, d3, r2);
      addaface(anode, d2, d3, r3);
      addanarc(anode, d1, a1, r1, r2);
      addanarc(anode, d2, a2, r1, r3);
      addanarc(anode, d3, a3, r2, r3);
      addavertex(anode, d1, a1);
      addavertex(anode, d2, a2);
      addavertex(anode, d3, a3);
      addavertex(d1, anode, a1);
      addavertex(d2, anode, a2);
      addavertex(d3, anode, a3);
   END {ychange} ;


PROCEDURE yswitch(anode: noderange; VAR yswap: boolean);

   VAR
      n1, n2, n3: noderange;
      r1, r2, r3, this: faceptr;
      highface: RECORD
                   f: faceptr;
                   v: integer;
                END;
      vptr: nodeptr;
      benefit: integer;

   BEGIN
      IF nodetable[anode].valence = 3
      THEN
         BEGIN
            get3faces(anode, r1, r2, r3);
            highface.f := NIL;
            highface.v := 0;
            this := firstface;
            WHILE this <> NIL DO
               BEGIN
                  IF ((this <> r1) AND ((this <> r2) AND (this <> r3)))
                  THEN
                     BEGIN
                        WITH this ^ DO
                           BEGIN
                              n1 := v1;
                              n2 := v2;
                              n3 := v3;
                           END;
                        benefit := yweight(anode, n1, n2, n3);
                        IF benefit > highface.v THEN
                           WITH highface DO
                              BEGIN
                                 f := this;
                                 v := benefit;
                              END;
                     END;
                  this := this ^.faceadj;
               END;
            vptr := nodetable[anode].nextvertex;
            n1 := otherend(anode, vptr ^.arcloc);
            vptr := vptr ^.nextnode;
            n2 := otherend(anode, vptr ^.arcloc);
            vptr := vptr ^.nextnode;
            n3 := otherend(anode, vptr ^.arcloc);
            IF highface.v > yweight(anode, n1, n2, n3)
            THEN
               BEGIN
                  writeln(' CHANGE ', anode: 3, ' IN FACE ', n1: 3, n2:
                     3, n3: 3);
                  WITH highface.f ^ DO
                     BEGIN
                        n1 := v1;
                        n2 := v2;
                        n3 := v3
                     END;
                  writeln('   INTO ', anode: 3, ' IN FACE ', n1: 3, n2:
                     3, n3: 3);
                  ychange(anode, r1, r2, r3, highface.f);
                  yswap := true
               END;
         END;
   END {yswitch} ;


BEGIN {maxplanar}
   initrandom;
   FOR starting := maxweight TO randomized DO
      FOR enter := ordered TO delta DO
         IF NOT ((starting = maxtetra) OR ((starting = randomized) AND (
            enter = ordered)))
         THEN
            BEGIN
               reset(tetra);
               readinput;
               statusreport;
               timec := clock;
               initialization;
               tetrahedron;
               FOR i := 1 TO n DO
                  nodegain(i);
               REPEAT
                  CASE enter OF
                     ordered:
                        anode := pickorder;
                     largest:
                        anode := pick1;
                     delta:
                        anode := pick2
                  END;
                  {insertinformation(anode);}
                  addanode(anode, nodetable[anode].vactive ^.face1);
                  FOR i := 1 TO n DO
                     gainupdate(i);
               UNTIL nv = n;
               timec := clock - timec;
               writeln(' RUNTIME FOR CONSTRUCTION ', timec: 6,
                  ' MIL-SEC');
               writeln(' TOTAL ASSIGNMENT COST ', assigncost: 6);
               timei := clock;
               firstround := true;
               yswap := false;
               REPEAT
                  newarc := firstarc;
                  arcswap := false;
                  WHILE newarc <> NIL DO
                     BEGIN
                        switch(newarc, arcswap);
                        newarc := newarc ^.arcadj;
                     END;
                  IF firstround OR ((arcswap = true) OR (yswap = true))
                  THEN
                     BEGIN
                        yswap := false;
                        FOR i := 1 TO n DO
                           yswitch(i, yswap);
                     END;
                  firstround := false;
               UNTIL ((arcswap = false) AND (yswap = false));
               timei := clock - timei;
               timet := timec + timei;
               writeln(' ITERATION TIME ', timei: 6, ' MIL-SEC');
               writeln(' FINAL ASSIGNMENT COST ', assigncost: 6, ' IN ',
                  timet: 6, ' MIL-SEC');
               writeln('1');
               garbagecollection;
            END;
   replaceseeds;
END {maxplanar} .
```

```
      PROGRAM ROC15 (INPUT,OUTPUT,ROCD,ROCDC,TAPE5=INPUT,
     1              TAPE6=OUTPUT,TAPE4=ROCD,TAPE3=ROCDC)


      IMPLICIT INTEGER (A-Z)

      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)
      COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                  DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                  DUP3(177)
      DIMENSION      NOWR(97),NOWC(97)

C
C     THIS PROGRAM IS SET UP TO REARRANGE ROWS AND COLUMN
C     OF A MATRIX ACCORDING TO RANKED ORDER CLUSTER ALGORITHM
C     ROC13 USE RADIX SORT (SHIFF SUBROUTINE) AS MAIN SORTING
C     ALGORITHM
C     INSERTING SORT IS USED AS SECONDARY SORTING PROCEDURE
C     DATA TO BE GENERATED BY PROGRAM....ROCDAT......
C     ROC1 FIRST PROGRAMMED IN DECEMBER 1979
C     THIS IS AN INTERACTIVE VERSION OF ROC1
C     ROC15 FIRST PROGRAMMED IN JANUARY 1980
C     THIS VERSION UPDATED JULY 1981
C     WRITTEN BY   V. NAKORNCHAI
C     COPYRIGHTED BY V. NAKORNCHAI  JULY 1981

C     MAINS VARIABLES

C     THE DATA ARE IN THE FORM OF  5 COLUMN REPRESENTATION
C     OROW           ORIGINAL ROW LOCATION
C     OCOL           ORIGINAL COLUMN LOCATION
C     NEXSR          ADDRESS TO THE NEXT DATA OF THE SAME ORIGINAL ROW
C     NEXSC          ADDRESS TO THE NEXT DATA OF THE SAME ORIGIAL COL
C     CAP            DATA VALUE

C     INROW          ACCESS TO THE ORIGINAL ROW
C     INCOL          ACCESS TO THE ORIGINAL COL
C     ROWE           NUMBER OF NON ZERO ELEMENTS IN A ROW
C     COLE           NUMBER OF NON ZERO ELEMENTS IN A COL
C     ORGROW         ORIGINAL NUMBER OF ROW IN THE MATRIX
C     ORGCOL         ORIGINAL NUMBER OF COL IN THE MATRIX
C     NROW           CURRENT  NUMBER OF ROW IN THE MATRIX
C     NCOL           CURRENT  NUMBER OF COL IN THE MATRIX
C     DUM            DUMMY MATRIX
C     LOCC(I)        CURRENT COLUMN OF COMPONENT I
C     LOCM(I)        CURRENT ROW OF MACHINE I
C     CCONT(I)       CURRENT  COMPONENT IN COLUMN I
C     RCONT(I)       CURRENT  MACHINE IN ROW I
C     NOP            TOTAL NUMBER OF NON ZERO ELEMENTS IN THE MATRIX


      WRITE(6,9530)
 9530 FORMAT(' TO READ DATA FROM THE ORIGINAL FILE ENTER ANY NO.',/,
     1       ' TO CONTINUE FROM PREVIOULY STORED STATE    (CR)')
      READ(5,*,END=130) ID
C
C     READ DATA FROM FILE ROCD
C
```

```
 9000 FORMAT(20I5)
   50 READ(4,9000) NCOL,NROW,NOP
      READ(4,9000) (INCOL(I),I=1,NCOL)
      READ(4,9000) (COLE(I), I=1,NCOL)
      READ(4,9000) (INROW(I),I=1,NROW)
      READ(4,9000) (ROWE(I), I=1,NROW)
      READ(4,9000) (OROW(I), I=1,NOP )
      READ(4,9000) (OCOL(I), I=1,NOP )
      READ(4,9000) (NEXSR(I),I=1,NOP )
      READ(4,9000) (NEXSC(I),I=1,NOP )
      READ(4,9000) (CAP(I),  I=1,NOP )

C
C     INITIALIZATION
C
      ITERA=0
      IDEL=1
      DO 100 I=1,NROW
      LOCM(I)=I
      NOWR(I)=I
      RCONT(I)=I
      BOTMAC(I)=0
  100 CONTINUE
      DO 120 I=1,NCOL
      LOCC(I)=I
      NOWC(I)=I
      CCONT(I)=I
  120 CONTINUE
      ORGROW=NROW
      ORGCOL=NCOL
      WRITE(6,9620)
 9620 FORMAT(' IN REPEATING THE SAME OPERATION CONSECUTIVELY ONLY',
     1       ' ONE INSTRUCTION GIVEN',/,' TO LIST INSTRUCTION  (CR)')

      CALL INIDUM
      GO TO 145
C
C     READ DATA FROM FILE ROCDC
C     I.E. CONTINUE FROM PREVIOUS STORED STATE
C
  130 READ(3,9000,END=140) ORGCOL,ORGROW,NCOL,NROW,NOP
      READ(3,9000) ITERA,IDEL,NMOD,NHEAD,DUMP
      READ(3,9000) (INCOL(I),  I=1,NCOL)
      READ(3,9000) (COLE(I),   I=1,NCOL)
      READ(3,9000) (INROW(I),  I=1,NROW)
      READ(3,9000) (ROWE(I),   I=1,NROW)
      READ(3,9000) (OROW(I),   I=1,NOP )
      READ(3,9000) (OCOL(I),   I=1,NOP )
      READ(3,9000) (NEXSR(I),  I=1,NOP )
      READ(3,9000) (NEXSC(I),  I=1,NOP )
      READ(3,9000) (CAP(I),    I=1,NOP )
      READ(3,9000) (NOWR(I),   I=1,NROW)
      READ(3,9000) (NOWC(I),   I=1,NCOL)
      READ(3,9000) (LOCM(I),   I=1,NROW)
      READ(3,9000) (LOCC(I),   I=1,NCOL)
      READ(3,9000) (RCONT(I),  I=1,NROW)
      READ(3,9000) (CCONT(I),  I=1,NCOL)
      READ(3,9000) (BOTMAC(I),I=1,NROW)
      READ(3,9000) (DUK1(I),   I=1,177 )
      READ(3,9000) (DUK2(I),   I=1,177 )
      READ(3,9000) (DUP1(I),   I=1,177 )
      READ(3,9000) (DUP2(I),   I=1,177 )
      READ(3,9000) (DUP3(I),   I=1,177 )
      READ(3,9000) (DUM1(I),  I=1,313 )
      READ(3,9000) (DUM2(I),  I=1,313 )
      READ(3,9000) (DUM3(I),  I=1,313 )

      GO TO 145
  140 WRITE(6,9540)
 9540 FORMAT(' NO PREVIOUS STATE DATA... READ FROM ORIGINAL SET')
      GO TO 50

C     REQUEST FOR INTERACTION IF REQUIRED
  145 WRITE(6,9630)
 9630 FORMAT ( ' IF INTERACTION IS REQUIRED ENTER  1 ELSE (CR)')
      READ(5,*,END=150) ID
      IF(ID.EQ.1)   CALL SETIN(ITERA)

C
C     SORT THE MACHINE ORDER
C
  150 DO 200 II=1,NCOL
      I=CCONT(NCOL-II+1)
C     IF NO OPERATION EXISTS   SKIP
      IF(COLE(I).EQ.0)    GO TO 200

      CALL CONSORT(I,-1)
      CALL SHIFF(COLE(I),-1)
  200 CONTINUE
C
C     CHECK FOR ANY REALLOCATION
C
      INERT=0
      DO 210 I=1,NROW
      IF(NOWR(I).NE.RCONT(I))THEN
          NOWR(I)=RCONT(I)
          INERT=1
      ENDIF
  210 CONTINUE
      IF(INERT.EQ.0)
     1    THEN
C             NO CHANGE    SORTING MAY BE COMPLETED
              IF(IDEL.EQ.1)
     1            THEN
                      IDEL=0
                      GO TO 205
                  ELSE
                       GO TO 2000
                  ENDIF
          ELSE
C             SORTING NOT COMPLETED
              ITERA=ITERA+1
C             REQUEST FOR MATRIX IF REQUIRED
              WRITE(6,9610) ITERA
              READ(5,*,END=205) ID
              IF(ID.EQ.1) CALL MATRIX (ITERA,1,0,0,0,0)
          ENDIF
C
C      SORT COMPONENT ORDER
C
  205 DO 220 II=1,NROW
      I=RCONT(NROW-II+1)
C     IF NO OPERATION EXISTS   SKIP
      IF(ROWE(I).EQ.0)    GO TO 220
      IF(BOTMAC(I).EQ.0)
     1    THEN
```

```
              CALL CONSORT(I,1)
              CALL SHIFF(ROWE(I),1)
          ENDIF
C     WRITE(6,9520) ITERA,II
  220 CONTINUE
C      CHECK FOR CHANGE IN REALLOCATION
      INERT=0
      DO 240 I=1,NCOL
      IF(NOWC(I).NE.CCONT(I)) THEN
          NOWC(I)=CCONT(I)
          INERT=1
      ENDIF
  240 CONTINUE
      IF(INERT.EQ.0)
     1    THEN
C             NO CHANGE    SORTING MAY BE COMPLETED
              IF(IDEL.EQ.1)
     1            THEN
                      IDEL=0
                      GO TO 150
                  ELSE
                       GO TO 2000
                  ENDIF
          ELSE
C             SORTING NOT COMPLETED
              ITERA=ITERA+1
              CALL MATRIX (ITERA,1,0,0,0,0)
              WRITE(6,9590)
              READ(5,*,END=150)IDEL
              IF(IDEL.EQ.-1)
     1            THEN
                      GO TO 2100
                  ELSEIF(IDEL.EQ.1)
     1                   THEN
                             CALL SETIN(ITERA)
                         ENDIF
                      GO TO 150
          ENDIF


 2000 CONTINUE
      WRITE(6,9600)
 9600 FORMAT(/,' STABLE ARRANGEMENT........',/,
     1          ' FURTHER INTERVENTION MAY BE REQUIRED')

 9590 FORMAT(' IF INTERVENTIONS ARE REQUIRED ENTER 1 ',/,
     1        ' TO TERMINATE THE PROBLEM ENTER -1',/,
     2        ' TO CONTINUE WITHOUT INTERVETION   (CR)')
 9610 FORMAT(' IF MATRIX OUTPUT AT ITERATION NO ',I3,2X,'REQUIRED',
     1        ' ENTER 1   ELSE  (CR)')
      WRITE(6,9590)
      READ(5,*,END=2100)IDEL
      IF(IDEL.EQ.1)
     1    THEN
              CALL SETIN(ITERA)
              GO TO 150
          ENDIF
C     OUTPUT THE RESULTS

 2100 CALL MATRIX(ITERA,0,0,0,0,0)
```

```
      WRITE(6,9500)
 9500 FORMAT(' ORDER OF THE MACHINES',//)
      WRITE(6,9000) (DUP2(RCONT(I)),I=1,NROW)
      WRITE(6,9510)
 9510 FORMAT(1X,//,' ORDER OF COMPONENTS',//)
      WRITE(6,9000)(CCONT(I),I=1,NCOL)
      REWIND 3
      WRITE(3,9000) ORGCOL,ORGROW,NCOL,NROW,NOP
      WRITE(3,9000) ITERA,IDEL,NMOD,NHEAD,DUMP
      WRITE(3,9000) (INCOL(I), I=1,NCOL)
      WRITE(3,9000) (COLE(I),  I=1,NCOL)
      WRITE(3,9000) (INROW(I), I=1,NROW)
      WRITE(3,9000) (ROWE(I),  I=1,NROW)
      WRITE(3,9000) (OROW(I),  I=1,NOP )
      WRITE(3,9000) (OCOL(I),  I=1,NOP )
      WRITE(3,9000) (NEXSR(I), I=1,NOP )
      WRITE(3,9000) (NEXSC(I), I=1,NOP )
      WRITE(3,9000) (CAP(I),   I=1,NOP )
      WRITE(3,9000) (NOWR(I),  I=1,NROW)
      WRITE(3,9000) (NOWC(I),  I=1,NCOL)
      WRITE(3,9000) (LOCM(I),  I=1,NROW)
      WRITE(3,9000) (LOCC(I),  I=1,NCOL)
      WRITE(3,9000) (RCONT(I), I=1,NROW)
      WRITE(3,9000) (CCONT(I), I=1,NCOL)
      WRITE(3,9000) (BOTMAC(I),I=1,NROW)
      WRITE(3,9000) (DUK1(I),  I=1,177 )
      WRITE(3,9000) (DUK2(I),  I=1,177 )
      WRITE(3,9000) (DUP1(I),  I=1,177 )
      WRITE(3,9000) (DUP2(I),  I=1,177 )
      WRITE(3,9000) (DUP3(I),  I=1,177 )
      WRITE(3,9000) (DUM1(I),  I=1,313 )
      WRITE(3,9000) (DUM2(I),  I=1,313 )
      WRITE(3,9000) (DUM3(I),  I=1,313 )

      END



      SUBROUTINE CONSORT (M,IDD)

      IMPLICIT INTEGER (A-Z)
      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)
      DIMENSION NOWR(97),NOWC(97)
      DATA  IR(1,1)/-999999/,IR(1,2)/-999999/
C
C     THE SUBROUTINE WILL CONSTRUCT A MATRIX TO BE CONTINUALLY
C     RADIX SORTED
C
C     MAIN VARIABLES
C     M       DIGIT TO BE RADIX SORTED
C     IDD     =-1     SORTED ALONG THE COLUMN I.E. REGROUP MACHINES
C             = 1     SORTED ALONG THE ROW I.E. REGROUP COMPONENTS
C     IR( ,1)         VALUE TO BE SORTED
C     IR( ,2)         M/C OR COMPONENT NUMBER


      KK=0
      IF(IDD.EQ.-1)
     1    THEN
              IN=INCOL(M)
C             REGROUPING MACHINE
              DO 10 I=2,COLE(M)+1
              I2=OROW(IN)
              IF(BOTMAC(I2).EQ.1)
     1            THEN
                      K=LOCM(I2)
                      KK=1
                  ELSE
                      K=LOCM(I2)
                  ENDIF
              CALL INSERT(I-1,K,I2)
              IN=NEXSC(IN)
   10         CONTINUE
              IF(KK.EQ.1)
     1            THEN
                      DO 15 I=2,COLE(M)+1
                      IR(I,1)=LOCM(IR(I,2))
   15                 CONTINUE
                  ENDIF
          ELSE
              IN=INROW(M)
C             REGROUPING COMPONENTS
              DO 20 I=2,ROWE(M)+1
              I2=OCOL(IN)
              CALL INSERT(I-1,LOCC(I2),I2)
              IN=NEXSR(IN)
   20         CONTINUE
          ENDIF
      RETURN
      END



      SUBROUTINE SHIFF(M,IDD)

      IMPLICIT INTEGER (A-Z)
      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)

C     THE SUBROUTINE IS RADIX SORTING IN ESSENCE
C     IN PRACTICE THE ALGORITHM IS PURELY SHIFTING
C     DIGITS AROUND

C     M       NUMBER OF ITEMS TO BE SHIFTED

      MM=M
      I=IR(M+1,1)
      J=I-1
      IF(IDD.EQ.-1)
     1    THEN
C              SORTING M/C ORDER
               WHILE(J.GE.1) DO
               IF(J.EQ.IR(MM,1))
     1             THEN
                       MM=MM-1
                       J=J-1
```

```
                   ELSE
                       RCONT(I)=RCONT(J)
                       I=I-1
                       J=J-1
                   ENDIF

               ENDWHILE
               DO 10 JJ=1,M
               RCONT(JJ)=IR(JJ+1,2)
   10          CONTINUE
               DO 20 JJ=1,NROW
               LOCM(RCONT(JJ))=JJ
   20          CONTINUE
          ELSE
C              SORTING COMPONENT ORDER

               WHILE(J.GE.1) DO
                   IF(J.EQ.IR(MM,1))
     1                 THEN
                           MM=MM-1
                           J=J-1
                       ELSE
                           CCONT(I)=CCONT(J)
                           I=I-1
                           J=J-1
                       ENDIF
               ENDWHILE

               DO 30 JJ=1,M
               CCONT(JJ)=IR(JJ+1,2)
   30          CONTINUE
               DO 40 JJ=1,NCOL
               LOCC(CCONT(JJ))=JJ
   40          CONTINUE
          ENDIF
      RETURN
      END
      SUBROUTINE INSERT (M,J1,J2)
      IMPLICIT INTEGER (A-Z)

      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)

C     THE SUBROUTINE IS CALLED BY  CONSORT
C     FOR REFERNCE SEE HOROWITZ AND SAHNI(1976)
C     'FUNDAMENTALS OF DATA STRUCTURES'
C     SORTED IN ***********NON-DECREASING ORDER************
C
C     MAIN VARIABLES
C
C     IR      RECORD TO BE INSERTED (SORTED)
C     M       SIZE OF THE ORIGINAL MATRIX NOT INCLUDING IR(1,1)
C     J1      INDEX TO BE SORTED
C     J2      THE DATA TO BE INSERTED ACCORDING TO J1
C
C     NOTE..... IR(1,1) ASSUME TO BE VERY LARGE NEGATIVE.......

      K=J1
      KK=J2
      N=M

      WHILE(K.LT.IR(N,1)) DO
      IR(N+1,1)=IR(N,1)
      IR(N+1,2)=IR(N,2)
      N=N-1

      ENDWHILE

      IR(N+1,1)=K
      IR(N+1,2)=KK

      RETURN
      END




      SUBROUTINE SETIN(ITERA)


      IMPLICIT INTEGER (A-Z)

      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)
      COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                  DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                  DUP3(177)
      DIMENSION  NOWR(97),NOWC(97)
C
C     THE ROUTINE VARIOUS DATA THAT MIGHT BE REQUIRED
C     DURING INTERACTIVE INTERVENTION
C
 9000 FORMAT(20I5)
 9530 FORMAT(' IF MATRIX PRINT OUT IS REQUIRED ENTER  1   ELSE   (CR)')
 9540 FORMAT(' IF THE PRESENT STATUS OF  MACHINES REQUIRED',
     1      ' ENTER 1   ELSE   (CR) ')
 9550 FORMAT(1X,///,' LIST OF THE BOTTLE-NECK MACHINE(S)')
 9560 FORMAT(1X,///,' LIST OF DUPLICATED MACHINE(S)')
 9570 FORMAT(' EMPTY')
 9580 FORMAT(' MACHINE ',I5,2X,'IS A DUPLICATION OF',I5)

      IP=0
  100 WRITE(6,9530)
      READ(5,*,END=110)ID
      IF(ID.EQ.1) CALL MATRIX(ITERA,0,0,0,0,0)
  110 WRITE(6,9540)
      READ(5,*,END=140) ID
      IF(ID.EQ.1)
     1    THEN
              WRITE(6,9550)
              IDD=0
              DO 120 I=1,NROW
                  IF(BOTMAC(I).EQ.1)
     1                THEN
                          WRITE(6,9000) I
                          IDD=1
                      ENDIF
  120         CONTINUE
```

```
              IF(IDD.EQ.0)    WRITE(6,9570)
              WRITE(6,9560)
              IF(NROW.GT.ORGROW)
     1            THEN
                      DO 125 I=ORGROW+1,NROW
                         WRITE(6,9580) I,DUP2(I)
  125                 CONTINUE
                  ELSE
                      WRITE(6,9570)
                  ENDIF
          ENDIF
  140 IF(IP.EQ.1)    GO TO 200

      CALL EXCEPT(ITERA)
      IP=1
      GO TO 100


  200 CONTINUE

      END



      SUBROUTINE EXCEPT(ITERA)




      IMPLICIT INTEGER (A-Z)

      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)

      COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                  DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                  DUP3(177)
C
C     THE SUBROUTINE WILL ALLOW INTERACTION WITH
C     THE MACHINE-COMPONENT MATRIX
C
 9500 FORMAT(' INPUT ERROR   PLEASE RE-ENTER ')

 9510 FORMAT(' ENTER  0  TO TERMINATE THE EXCEPTION ROUTINES',/,
     1       '        1  TO INSPECT LOCAL GROUPING OF OPERATIONS',/,
     2       '        2  TO DELETE AN OPERATION ',/,
     3       '        3  TO RE-ENTER AN OPERATION',/,
     4       '        4  TO DEFINE OR RELAX BOTTLE-NECK MACHINES',/,
     5       '        5  TO INCREASE NUMBER OF A TYPE OF M/C',/,
     6       '        6  TO MERGE TWO M/CS OF A CERTAIN TYPE',/,
     7       '        7  TO REORDER ROWS OR COLUMNS')

 9520 FORMAT (' 0-TERMINATE 1-ZOOM 2-DELETE 3-ENTER 4-BOTTLENECK',/,
     1        ' 5-DUPLICATE 6-MERGE 7-REORDER  FOR DETAILS (CR) ')


      IF (ITERA.GT.1) GO TO 110
  100 WRITE(6,9510)
      GO TO 120
  110 WRITE(6,9520)
  120 READ(5,*,END=100) ID
      IF       (ID.EQ.0)    THEN
                                  RETURN
        ELSEIF(ID.EQ.1)     THEN
                                  CALL ZOOM(ITERA)
        ELSEIF(ID.EQ.2)     THEN
                                  CALL DELETE
        ELSEIF(ID.EQ.3)     THEN
                                  CALL PUTBAK
        ELSEIF(ID.EQ.4)     THEN
                                  CALL BOTNECK
        ELSEIF(ID.EQ.5)     THEN
                                  CALL ENLARGE
        ELSEIF(ID.EQ.6)     THEN
                                  CALL MERGE
        ELSEIF(ID.EQ.7)     THEN
                                  CALL PATCH
                            ELSE
                                  WRITE(6,9500)
      ENDIF

      GO TO 110
      END


      SUBROUTINE DELETE



      IMPLICIT INTEGER (A-Z)

      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)

      COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                  DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                  DUP3(177)
C     THE SUBROUTINE WILL ALLOW INTERACTIVELY THE
C     REMOVAL OF AN OPERATION IN THE MACHINE-COMPONENT MATRIX
C
 9500 FORMAT(' INPUT ERROR   PLEASE RE-ENTER ')
 9510 FORMAT(' TO TERMINATE DELETE ROUTINE ENTER   0  0  ELSE',/,
     1       ' INPUT THE REQUIRED MACHINE AND COMPONENT')
 9520 FORMAT(' NO OPERATION LEFT ON M/C OR COMPONENT',//)



  100 WRITE(6,9510)
  110 READ(5,*,END=100) IM,IC
      BOUND=TESTB(IM,IC,NROW,NCOL)
      IF       (BOUND.EQ.0)    THEN
                                   GO TO 1000
        ELSEIF(BOUND.LE.1)     THEN
                                   WRITE(6,9500)
                                   GO TO 110
      ENDIF
      IF(COLE(IC).EQ.0.OR.ROWE(IM).EQ.0)
     1    THEN
C             NO OPERATION LEFT
              WRITE(6,9520)
              GO TO 110
          ENDIF

      CALL TESTC (IM,IC,BOUND,LOC0,LOC1)
      IF (BOUND.EQ.3)
     1    THEN
              CALL REMOVE(IM,IC,LOC0,LOC1,0)
          ELSEIF(BOUND.EQ.4)
     1            THEN
                      WRITE(6,9530)
 9530                 FORMAT(' ALREADY REMOVED OR NONEXISTANT')
                  ELSE
                      WRITE(6,9500)
                  ENDIF
      GO TO 110


 1000 CONTINUE
      RETURN
      END
      INTEGER FUNCTION TESTB(IMM,ICC,NROW,NCOL)

C     TO TEST THE BOUNDS OF THE INPUT
C

      IF(IMM.EQ.0.OR.ICC.EQ.0)
     1    THEN
C             TERMINATE THE PROCEDURE
              TESTB=0
          ELSEIF(IMM.EQ.-1.OR.ICC.EQ.-1)
     1            THEN
                      TESTB=-1
          ELSEIF(IMM.EQ.-99.OR.ICC.EQ.-99)
     1            THEN
                      TESTB=-99
          ELSEIF(IMM.LT.1.OR.IMM.GT.NROW.OR.
     1           ICC.LT.1.OR.ICC.GT.NCOL)
     2            THEN
C                     OUT OF BOUND
                      TESTB=1
                  ELSE
C                     WITHIN BOUNDS
                      TESTB=2
                  ENDIF
       RETURN
      END

      SUBROUTINE TESTC(IMM,ICC,BOUND,LOC0,LOC1)


      IMPLICIT INTEGER (A-Z)

      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
```

```
C      TO TEST WHETHER THE OPRATION CAN BE 'COVERED UP'

       ROWEI=ROWE(IMM)
       INR=INROW(IMM)
       LOCO=0

       WHILE(ROWEI.GT.0) DO
          IF(OCOL(INR).EQ.ICC)
      1       THEN
C                 CAN BE REMOVED
                   BOUND=3
                   LOC1=INR
                   RETURN
              ENDIF
              ROWEI=ROWEI-1
              LOCO=INR
              INR=NEXSR(INR)
       ENDWHILE
C      EITHER COVERED OR NONEXISTANT

       BOUND=4
       RETURN
       END


       SUBROUTINE TESTD (B1,B2,B3,BO)


       IMPLICIT INTEGER(A-Z)
C       TEST OF BOUNDS FOR MATRIX PRINTING

       IF(B1.EQ.0)
      1    THEN
               B1=1
               B2=BO
               B3=1
               RETURN
           ENDIF

       IF(B1.LT.0.OR.B1.GT.BO.OR.
      1   B2.LE.0.OR.B2.GT.BO)
      2    THEN
               B3=0
           ELSEIF(B1.GT.B2)
      1        THEN
                   B3=B2
                   B2=B1
                   B1=B3
                   B3=1
               ENDIF
       RETURN
       END

       SUBROUTINE REMOVE (MAC,COM,LOCO,LOC1,ENG)


       IMPLICIT INTEGER (A-Z)

       COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
      1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP

      COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                  DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                  DUP3(177)
C      TO REMOVE THE OPERATIONS FROM THE PRESENT CONSIDERATION
C      DUMP THE INFORMATION INTO MATRICES IN DUMSET
C      SUBROUTINE INIDUM MUST BE CALLED FIRST
C      ENG=0  NORMAL REMOVAL OF AN OPERATION
C      ENG=1  CREATING AN EXTRA MACHINE


C      IF CREATING A NEW MACHINE SKIP
      IF(ENG.EQ.1)   GO TO 10



C      COPY PART OF THE CONTENTS IN TO DUM MATRICES
C
      IC=DUK1(MAC)
      IF(IC.EQ.0)
     1    THEN
C            FIRST ENTRY
              DUK2(MAC)=DUMP
          ELSE
              ICC=DUK2(MAC)
              WHILE(IC.GT.1) DO
              ICC=DUM3(ICC)
              IC=IC-1
              ENDWHILE
              DUM3(ICC)=DUMP
          ENDIF
      DUM1(DUMP)=COM
      DUM2(DUMP)=LOC1
      DD=DUM3(DUMP)
      DUM3(DUMP)=0
      DUMP=DD
      DUK1(MAC)=DUK1(MAC)+1
C     REARRANGE INDICES TO BYPASS THE ELEMENT
C
C     ALONG THE ROW

C     CHECK FOR ONE OPERATION ONLY

   10 IF(ROWE(MAC).EQ.1)      GO TO 50

C     RESET ROW ENTRY INDEX IF NECCESSARY
      IF(LOCO.EQ.0)
     1    THEN
              INROW(MAC)=NEXSR(LOC1)
              IE=ROWE(MAC)
              ID=INROW(MAC)

              WHILE (IE.GT.2) DO
                  ID=NEXSR(ID)
                  IE=IE-1
              ENDWHILE

              NEXSR(ID)=INROW(MAC)
          ELSE
              NEXSR(LOCO)=NEXSR(LOC1)
          ENDIF
   50 ROWE(MAC)=ROWE(MAC)-1

C      ALONG THE COLUMN

C      IF CREATING A NEW MACHINE SKIP
       IF(ENG.EQ.1)  GO TO 150

C      CHECK FOR ONE OPERATION ONLY  IF FOUND SKIP

       IF(COLE(COM).EQ.1)     GO TO 100


C      RESET COLUMN ENTRY INDEX IF NECESSARY
       IF(INCOL(COM).EQ.LOC1) INCOL(COM)=NEXSC(LOC1)
C      BY PASS
       IE=COLE(COM)
       IDD=INCOL(COM)
       IF(IE.EQ.2)
     1     THEN
               NEXSC(INCOL(COM))=INCOL(COM)
               GO TO 100
           ENDIF

       WHILE(IE.GT.2) DO
           ID=IDD
           IDN=NEXSC(ID)
           IE=IE-1
           IF(OROW(IDN).EQ.MAC)
     1         THEN
C                  JUMP OUT OF LOOP
                   NEXSC(ID)=NEXSC(NEXSC(ID))
                   GO TO 100
               ELSE
                   IDD=IDN
               ENDIF
       ENDWHILE
       NEXSC(IDN)=NEXSC(NEXSC(IDN))
  100  COLE(COM)=COLE(COM)-1

  150  CONTINUE
       RETURN
       END




       SUBROUTINE PUTBAK

       IMPLICIT INTEGER (A-Z)
       COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1                OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2                DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
       COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1                BOTMAC(97)
       COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                   DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                   DUP3(177)


C      THE ROUTINE WILL ENABLE A PARTICULAR OPERATION TO BE RETURNED
C      INTO THE ORIGINAL MACHINE-COMPONENT MATRIX
```

```
9500 FORMAT(' INPUT ERROR  PLEASE RE-ENTRY')
9510 FORMAT(' TO TERMINATE PUTBAK ROUTINE ENTER  0  0',/,
    1       ' ELSE ENTER THE MACHINE AND COMPONENT NUMBERS')
9520 FORMAT(' THE OPERATION WAS NOT REMOVED ')
9530 FORMAT(' IF THE OPERATION IS TO BE PUT BACK IN THE SAME M/C',
    1       '  (CR)',/,' ELSE ENTER ALTENATIVE OF THE SAME TYPE')
9540 FORMAT(' THE TWO M/CS IS NOT OF THE SAME TYPE')



 100 WRITE(6,9510)
 110 READ(5,*,END=100) IM,IC
     BOUND=TESTB(IM,IC,NROW,NCOL)
     IF      (BOUND.EQ.0)    THEN
                                   RETURN
       ELSEIF(BOUND.LE.1)    THEN
                                   WRITE(6,9500)
                                   GO TO 110
     ENDIF

     IF(DUK1(IM).EQ.0)
    1    THEN
             WRITE(6,9520)
             GO TO 110
         ENDIF



     PK=0
     K =DUK2(IM)


     WHILE(K.GT.0) DO
         IF(DUM1(K).EQ.IC)
    1        THEN
                 IF(PK.EQ.0)
    1                THEN
                         DUK2(IM)=DUM3(K)
                     ELSE
                         DUM3(PK)=DUM3(K)
                     ENDIF
                 KK=DUM2(K)
                 DUK1(IM)=DUK1(IM)-1
                 DUM3(K)=DUMP
                 DUMP=K
                 GO TO 200
             ELSE
                 PK=K
                 K =DUM3(K)
             ENDIF
     ENDWHILE

C    OPERATION NOT FOUND
     WRITE(6,9520)
     GO TO 100

C    OPERATION FOUND

 200 WRITE(6,9530)
     READ(5,*,END=300) IM1
```

```
      BOUND=TESTB(IM1,1,NROW,1)
      IF      (BOUND.LE.1)                      THEN
                                                    WRITE(6,9500)
                                                    GO TO 200
        ELSEIF(DUP2(IM1).NE.DUP2(IM))           THEN
                                                    WRITE(6,9540)
                                                    GO TO 200
                                                ELSE
                                                    IM=IM1
                                                ENDIF

C     INSERT THE OPERATION INTO THE ORIGINAL DATA STRUCTURE

C     ALONG THE COLUMN

  300 IF(COLE(IC).EQ.0)
     1    THEN
              INCOL(IC)=KK
              NEXSC(KK)=KK
          ELSE
              I=NEXSC(INCOL(IC))
              NEXSC(INCOL(IC))=KK
              NEXSC(KK)=I
          ENDIF
      COLE(IC)=COLE(IC)+1

C      ALONG THE ROW

      IF(ROWE(IM).EQ.0)
     1    THEN
              INROW(IM)=KK
              NEXSR(KK)=KK
          ELSE
              I=NEXSR(INROW(IM))
              NEXSR(INROW(IM))=KK
              NEXSR(KK)=I
          ENDIF
      OROW(KK)=IM
      ROWE(IM)=ROWE(IM)+1
      GO TO 100

      END


      SUBROUTINE BOTNECK


      IMPLICIT INTEGER (A-Z)
      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)


 9500 FORMAT(' TO TERMINATE BOTTLE-NECK ROUTINE ENTER 0 0',/,
     1       ' TO SPECIFY A BOTTLE-NECK MACHINE ENTER 1 & M/C NUMBER',/,
     2       ' TO RELEASE A BOTTLE-NECK MACHINE ENTER 0 & M/C NUMBER')
 9510 FORMAT(' INPUT ERROR PLEASE RE-ENTER')

   50 WRITE(6,9500)
  100 READ(5,*,END=50) IDUM,IMAC

      IF((IDUM.NE.0.OR.IDUM.NE.1).AND.(IMAC.LT.0.OR.IMAC.GT.NROW))
     1    THEN
              WRITE(6,9510)
              GO TO 100
          ENDIF

      IF(IMAC.EQ.0)      RETURN

      IF(IDUM.EQ.1)
     1    THEN
              BOTMAC(IMAC)=1
          ELSE
              BOTMAC(IMAC)=0
          ENDIF

      GO TO 100

      END


      SUBROUTINE PATCH

      IMPLICIT INTEGER (A-Z)

      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)

 9500 FORMAT (' ENTER  0  TO RETURN',/,
     1         '        1  TO REORDER ROWS',/,
     2         '        2  TO REORDER COLUMNS')
 9510 FORMAT (' REORDERING THE ROW ')
 9520 FORMAT (' REORDERING THE COLUMN ')

  100 WRITE(6,9500)
      READ (5,*,END=100)I
      IF(I.EQ.1)
     1   THEN
            WRITE(6,9510)
            CALL JUGGLE (LOCM,RCONT,NROW)
      ELSEIF(I.EQ.2)
     1   THEN
            WRITE(6,9520)
            CALL JUGGLE (LOCC,CCONT,NCOL)
         ENDIF
      RETURN
      END


      SUBROUTINE JUGGLE (LOC, CONT, N)

      IMPLICIT INTEGER (A-Z)

      DIMENSION LOC(N), CONT(N), DUMMY(97)
      LOGICAL REPEAT

C     THIS ROUTINE IS CALLED BY PATCH WHICH INTURN
C     CALLED BY EXCEPT
 9000 FORMAT (10I5)
 9010 FORMAT (I5, ' IS OUT OF BOUND')
 9020 FORMAT (I5, ' IS ENTERED PREVIOUSLY')
 9500 FORMAT (' ENTER  0  TO EXIT',/,
     1        '        1     MOVE ELEMENTS TO THE FRONT',/,
     2        '        2     REENTRY THE WHOLE LIST',/,
     3        '        3     SWAP ANY TWO ELEMENTS')
 9510 FORMAT (' TO LIST THE PRESENT ORDER ENTER  1 ELSE  (CR)')
 9520 FORMAT (' ENTER THE ELEMENTS  ONE BY ONE',/,
     1        '       0 TO TERMINATE THE ENTRY')
 9530 FORMAT (' REENTRY  THE WHOLE LIST?',/,
     1        ' YES  ENTER 1   ELSE ANY NO.')
 9540 FORMAT (' ENTER THE NEW ORDER ONE BY ONE')
 9550 FORMAT (' ENTER THE PAIR REQUIRED TO BE SWAPPED',/,
     1        ' TO TERMINATE  ENTER  0  0')


   10 WRITE (6,9500)
      READ (5,*, END =10) I

          IF( I.EQ.0)
     1     THEN
             RETURN

C     MOVE ELEMENTS TO THE HEAD OF THE LIST

      ELSEIF(I.EQ.1)
     1     THEN
              ENTRY = 0
              WRITE (6, 9510)
              READ (5,*,END=20)D
              IF(D.EQ.1.) WRITE (6,9000) (CONT(J),J=1,N)
   20         WRITE (6,9520)
   30         READ(5,*) ELEMENT
              IF(ELEMENT.EQ.0.AND.ENTRY.EQ.0) GO TO 10
              IF(ELEMENT.EQ.0) GO TO 100
              IF(ELEMENT.LE.0.OR.ELEMENT.GT.N)
     1          THEN
                   WRITE(6,9010)ELEMENT
                   GO TO 30
          ELSEIF(ENTRY.EQ.0)
     1          THEN
                   ENTRY=1
                   DUMMY(I)=ELEMENT
                   GO TO 30
                ELSE
                   REPEAT = .FALSE.
                   E = ENTRY
   40              IF (.NOT.REPEAT )
     1               THEN
                       IF (DUMMY(E).EQ.ELEMENT) REPEAT=.TRUE.
                       E = E-1
                       IF(E.LE.0) GO TO 50
                       GO TO 40
                     ENDIF
   50              IF (REPEAT)
     1               THEN
                        WRITE (6,9020) ELEMENT
                     ELSE
                        ENTRY = ENTRY +1
                        DUMMY(ENTRY)= ELEMENT
                     ENDIF
                   GO TO 30
                 ENDIF
```

```
C     ENTRY SUCCESFUL
C             REMOVE THE PREVIOUS ENTRY
  100         DO 110 J=1, ENTRY
                CONT(LOC(DUMMY(J))) = 0
  110         CONTINUE
              E1=ENTRY + 1
              DO 120 J=1, N
                IF (CONT(J).NE.0)
     1             THEN
                      DUMMY(E1)= CONT(J)
                      E1=E1+ 1
                   ENDIF
  120         CONTINUE
              DO 130 J=1,N
                CONT(J) = DUMMY (J)
  130         CONTINUE
              DO 140 J=1,N
                LOC(CONT(J))=J
  140         CONTINUE


C     ENTER THE WHOLE LIST

      ELSEIF(I.EQ.2)
     1     THEN
              WRITE(6,9530)
              READ (5,*) J
C             IF NOT PROCESS GO BACK TO BEGINNING
              IF (J.NE.1) GO TO 10
C             TO GO AHEAD
              WRITE(6,9540)
              DO 300 J=1,N
  200            READ(5,*) ELEMENT
                 IF (ELEMENT.LE.0 .OR. ELEMENT.GT. N)
     1             THEN
                      WRITE(5,9010) ELEMENT
                      GO TO 200
                   ENDIF
                REPEAT = .FALSE.
                 J1 =J -1
                 IF (J1.EQ.0)
     1              THEN
                       DUMMY(J)=ELEMENT
                       GO TO 300
                    ENDIF

  210            IF (.NOT.REPEAT )
     1             THEN
                      IF (DUMMY(J1).EQ.ELEMENT) REPEAT=.TRUE.
                      J1=J1-1
                      IF (J1.EQ.0) GO TO 220
                      GO TO 210
                   ENDIF
  220            IF (REPEAT)
     1              THEN
                       WRITE (6,9020) ELEMENT
                       GO TO 200
                    ELSE
                       DUMMY(J) = ELEMENT
                    ENDIF
  300          CONTINUE
```

```
C              ENTRY SUCCESSFUL

               DO 310 J =1,N
                  CONT (J) =DUMMY (J)
  310          CONTINUE
               DO 320 J=1,N
                  LOC(CONT(J))= J
  320          CONTINUE


C      SWAPPING ARRANGEMENT

       ELSEIF (I.EQ. 3)
     1       THEN
  400          WRITE (6,9550)
  410          READ (5,*) E1,E2
               IF (E1.EQ.0 .OR. E2.EQ. 0) RETURN
               IF (E1.LT.0 .OR. E1.GT. N)
     1            THEN
                     WRITE(5,9010) E1
                     GO TO 400
                  ENDIF
               IF (E2.LT.0 .OR. E2. GT. N)
     1            THEN
                     WRITE(5,9010)E2
                     GO TO 400
                  ENDIF
               IF (E1.EQ.E2) GO TO 400

C              SWAPPING
               ROW1 = LOC(E1)
               ROW2 = LOC(E2)
               LOC(E1)  = LOC (E2)
               LOC(E2)  = ROW1
               DUMP = CONT(ROW1)
               CONT(ROW1) = CONT(ROW2)
               CONT(ROW2) = DUMP
               GO TO 410

       ENDIF
       RETURN
       END


       SUBROUTINE INIDUM


       IMPLICIT INTEGER (A-Z)

       COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1                OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2                DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP

       COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                   DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                   DUP3(177)


C
C      VARIABLES IN DUMSET
C
```

```
C      DUK1       NO OF ELEMENTS REMOVED FROM THE M/C
C      DUK2       POINTER TO CELLS WHERE THE REMOVED SET IS STORED
C      DUP1       NO OF DUPLICATED M/CS OF THIS TYPE
C      DUP2       TYPE OF M/C
C      DUP3       POINTER TO CELLS WHERE DUPLICATED SET IS STORED
C      DUM1       COLUMN NO. OR DUPLICATED M/C NO.
C      DUM2       POINTER IN SET1 OR M/C TYPE
C      DUM3       POINTER TO CELLS OF THE SAME SET

C      TO INITIALIZE DUMSET MATRICES


       DO 10 I=1,177
       DUK1(I)=0
       DUK2(I)=0
       DUP1(I)=0
       DUP2(I)=I
       DUP3(I)=0
    10 CONTINUE

       DO 20 I=1,233
       DUM1(I)=0
       DUM2(I)=0
       DUM3(I)=I+1
    20 CONTINUE

       DUM3(233)=1
       DUMP=1
C
C        CALCULATE VARIABLE FOR MATRIX HEADING
C
       IF(NROW.GE.10000)
      1    THEN
               NMOD=10000
               NHEAD=5
           ELSEIF(NROW.GE.1000)
      1            THEN
                       NMOD=1000
                       NHEAD=4
                   ELSEIF(NROW.GE.100)
      1                    THEN
                               NMOD=100
                               NHEAD=3
                           ELSEIF(NROW.GE.10)
      1                            THEN
                                       NMOD=10
                                       NHEAD=2
                                   ELSE
                                       NMOD=1
                                       NHEAD=1
                                   ENDIF

       RETURN
       END


       SUBROUTINE ZOOM(ITERA)

       IMPLICIT INTEGER(A-Z)

       COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
      1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
      2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
```

```
C     TO ALLOW INSPECTION OF LOCAL GROUPING

 9510 FORMAT(' DATA INPUT ERROR PLEASE RE-ENTER')
  100 WRITE(6,9500)
 9500 FORMAT(' ENTER THE RANGE OF LOCATIONS OF COMPONENTS')
      READ(5,*) IA,IB
      CALL TESTD (IA,IB,IC,NCOL)
      IF(IC.EQ.0)
     1    THEN
              WRITE(6,9510)
              GO TO 100
          ENDIF

  200 WRITE(6,9520)
 9520 FORMAT(' ENTER THE RANGE OF LOCATIONS OF MACHINES')
      READ(5,*) JA,JB
      CALL TESTD (JA,JB,JC,NROW)
      IF(JC.EQ.0)
     1    THEN
              WRITE(6,9510)
              GO TO 200
          ENDIF

      CALL MATRIX (ITERA,1,JA,JB,IA,IB)

      RETURN
      END


      SUBROUTINE ENLARGE


      IMPLICIT INTEGER (A-Z)
      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)
      COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                  DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                  DUP3(177)



 9500 FORMAT(' INPUT ERROR PLEASE RE-ENTRY')
 9510 FORMAT(' ENTER 0  TO TERMINATE ENLARGE M/CS PROCEDURE',/,
     1       ' ELSE  ENTER THE MACHINE TO BE INCREASED')
 9520 FORMAT(' NO OPERATION LEFT NO NEED TO DUPLICATE')
 9530 FORMAT(' ENTER 0 TO INDICATE THAT NO MORE COMPONENT',
     1       ' TO BE ENTERED FOR THIS DUPLICATION',/,
     2       ' ELSE ENTER THE COMPONENT NUMBER')
 9540 FORMAT(' THE OPERATION IS ALREADY COVERED OR NONEXISTANT')


  100 WRITE(6,9510)
  110 READ(5,*,END=100) OMAC
      BOUND=TESTB(OMAC,1,NROW,1)
      IF       (BOUND.EQ.0)     THEN
                                    RETURN
```

```
        ELSEIF(BOUND.NE.2)    THEN
                                  WRITE(6,9500)
                                  GO TO 110
      ENDIF

C      CHECK FOR NO OPERATION
      IF(ROWE(OMAC).EQ.0)
     1    THEN
              WRITE(6,9520)
              GO TO 110
          ENDIF

C     LOCATE AND INSERT THE NEW M/C INTO DUP LISTS
      IF(DUP1(OMAC).EQ.0)
     1    THEN
C             NO PREVIOUS DUPLICATION
              DUP3(OMAC)=DUMP
              DUM1(DUMP)=NROW+1
          ELSE
C             PREVIOUSLY DUPLICATED
              J=DUP3(OMAC)
              WHILE(DUM3(J).NE.0) DO
                  J=DUM3(J)
              ENDWHILE
              DUM3(J)=DUMP
              DUM1(DUMP)=NROW+1
          ENDIF

C      RESET THE INDICIES
      II=DUM3(DUMP)
      DUM2(DUMP)=DUP2(OMAC)
      DUM3(DUMP)=0
      DUMP=II
      NROW=NROW+1
      ROWE(NROW)=0
      LOCM(NROW)=NROW
      RCONT(NROW)=NROW
      DUP2(NROW)=DUP2(OMAC)
      BOTMAC(NROW)=0

C     ENTER THE LIST OF COMPONENTS
      JJ=0
  200 WRITE(6,9530)
  210 READ(5,*,END=200) IC
      BOUND=TESTB(1,IC,1,NCOL)
      IF      (BOUND.EQ.0)    THEN
                                  IF(JJ.EQ.0)
     1                                 THEN
C                                          NO ENTRY RESET INDICIES
                                           DUM3(DUP3(OMAC))=DUMP
                                           DUMP=DUP3(OMAC)
                                           NROW=NROW-1
                                       ENDIF
                                   GO TO 100
        ELSEIF(BOUND.NE.2)    THEN
                                   WRITE(6,9500)
                                   GO TO 210
      ENDIF

C     LOCATE THE OPERATION REQUIRED
      CALL TESTC(OMAC,IC,BOUND,LOC0,LOC1)
      IF(BOUND.EQ.4)
     1    THEN
```

```
C             NON-EXISTANCE
              WRITE(6,9540)
              GO TO 210
          ELSE
C             FOUND  RESET INDICIES
              JJ=1
              CALL REMOVE(OMAC,IC,LOC0,LOC1,1)
              ROWE(NROW)=ROWE(NROW)+1
              OROW(LOC1)=NROW
              IF(ROWE(NROW).EQ.1)
     1            THEN
                      INROW(NROW)=LOC1
                      NEXSR(LOC1)=LOG1
                  ELSE
                      NEXSR(LOC1)=NEXSR(INROW(NROW))
                      NEXSR(INROW(NROW))=LOC1
                      INROW(NROW)=LOC1
                  ENDIF
              GO TO 210
          ENDIF

      END


      SUBROUTINE MERGE


      IMPLICIT INTEGER (A-Z)
      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)
      COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                  DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                  DUP3(177)

 9500 FORMAT(' INPUT ERROR  PLEASE RE-ENTRY')
 9510 FORMAT(' ONLY MACHINES OF THE SAME TYPE CAN BE MERGED')
 9520 FORMAT(' TO TERMINATE THE MERGE PROCEDURE ENTER 0  0',/,
     1       ' ELSE ENTER THE TWO MACHINES TO BE MERGED',/,
     2       ' ENTER THE REMANING MACHINE FIRST')
 9530 FORMAT(' THE TWO MACHINES ARE NOT OF THE SAME TYPE')
 9540 FORMAT(' NO ELEMENT LEFT IN THE SECOND MACHINE')



C     TEST THE COMPATIBILITY OF DATA

      WRITE(6,9510)
  100 WRITE(6,9520)
  110 READ(5,*,END=100) IM1,IM2
      BOUND=TESTB(IM1,IM2,NROW,NROW)
      IF       (BOUND.EQ.0)             THEN
                                            RETURN
        ELSEIF(BOUND.NE.2.OR.
     1         IM1.EQ.IM2    )          THEN
C                                           NONCOMPATIBLE DATA
                                            WRITE(6,9500)
                                            GO TO 110
        ELSEIF(DUP2(IM1).NE.DUP2(IM2))  THEN
```

```
C                                           NOT THE SAME TYPE
                                            WRITE(6,9530)
                                            GO TO 110
        ELSEIF(ROWE(IM2).LE.0)          THEN
C                                           NO ELEMENT LEFTIN 2ND M/C
                                            WRITE(6,9540)
                                            GO TO 110
      ENDIF

C     MERGE THE MACHINES

C     CHANGE ROW NUMBER
      J=INROW(IM2)
      K=ROWE(IM2)-1
      WHILE(K.GT.0) DO
          OROW(J)=IM1
          J=NEXSR(J)
          K=K-1
      ENDWHILE
      OROW(J)=IM1

C
C     JOIN THE LISTS
      L=INROW(IM2)
      K=ROWE(IM2)
      NEXSR(J)=NEXSR(INROW(IM1))
      NEXSR(INROW(IM1))=L
      INROW(IM1)=L
      ROWE(IM1)=ROWE(IM1)+ROWE(IM2)
      ROWE(IM2)=0

      GO TO 110

      END


      SUBROUTINE MATRIX (ITERA,SUP,BBR,EER,BBC,EEC)

      IMPLICIT INTEGER (A-Z)
      COMMON /SET1/  INROW(97),INCOL(97),ROWE(97),COLE(97),
     1               OROW(313),OCOL(313),NEXSR(313),NEXSC(313),CAP(313),
     2               DUM(97),ORGROW,ORGCOL,NROW,NCOL,NOP
      COMMON /SORT1/ IR(97,2),LOCC(97),LOCM(97),CCONT(97),RCONT(97),
     1               BOTMAC(97)
      COMMON /DUMSET/   DUM1(233),DUM2(233),DUM3(233),DUMP,NMOD,NHEAD,
     1                  DUK1(177),DUK2(177),DUP1(177),DUP2(177),
     2                  DUP3(177)
      DIMENSION      ISPOT(130),ISIGN(4),IHEAD(130),NUM(9)

C     TO GENERATE GRAPHICALLY THE MACHINE-COMPONENT MATRIX
C
      DATA ISIGN(1)/1H1/,ISIGN(2)/1H /,ISIGN(3)/1H*/,ISIGN(4)/1H0/
      DATA ISPOT/130*(1H )/
      DATA NUM(1)/1H1/,NUM(2)/1H2/,NUM(3)/1H3/,NUM(4)/1H4/,NUM(5)/1H5/,
     1     NUM(6)/1H6/,NUM(7)/1H7/,NUM(8)/1H8/,NUM(9)/1H9/
 9500 FORMAT(X,///,7X,'      MATRIX AFTER ',I5,' ITERATION(S)',/)
 9510 FORMAT(10X,' COMPONENTS')
 9550 FORMAT(10X,' LOCATIONS')
 9010 FORMAT(1X,'(',I3,')',    I3,40(2X,A1))
 9020 FORMAT(9X,40(2X,A1))
 9030 FORMAT(1X,'(',I3,')',I3,1X,61(1X,A1))
 9040 FORMAT(10X,61(1X,A1))
 9050 FORMAT(1X,'(',I3,')',I3,2X,120A1)
 9060 FORMAT(11X,120A1)
 9070 FORMAT(1X,'(',I3,')',I3)

      BR=BBR
      ER=EER
      BC=BBC
      EC=EEC
      MHEAD=NHEAD
      ILOC=0
      IF(BR.EQ.0)
     1    THEN
              BR=1
              ER=NROW
              BC=1
              EC=NCOL
          ENDIF
      WIDTH=EC-BC

C     HEADING

      WRITE(6,9500) ITERA


 1000 MMOD=NMOD
      IF(ILOC.EQ.0)
     1    THEN
              ILOC=1
              DO 140 K=BC,EC
  140         DUM(K)=K
              WRITE(6,9550)
          ELSE
              ILOC=2
              DO 150 K=BC,EC
  150         DUM(K)=CCONT(K)
              WRITE(6,9510)
          ENDIF

      DO 210 K=1,MHEAD
          DO 200 KK=BC,EC
              FIG=DUM(KK)/MMOD
              IF(FIG.LE.0)
     1            THEN
                      IHEAD(KK)=ISIGN(4)
                  ELSE
                      IHEAD(KK)=NUM(FIG)
                  ENDIF
              DUM(KK)=MOD(DUM(KK),MMOD)
  200     CONTINUE
          IF(WIDTH.LE.40)
     1        THEN
                  WRITE(6,9020) (IHEAD(I),I=BC,EC)
              ELSEIF(WIDTH.LE.61)
     1                THEN
                          WRITE(6,9040) (IHEAD(I),I=BC,EC)
                      ELSE
                          WRITE(6,9060) (IHEAD(I),I=BC,EC)
                      ENDIF
          MMOD=MMOD/10
  210 CONTINUE

C     PRINT LOCATION IF NOT DONE SO
      IF(ILOC.EQ.1)   GO TO 1000
      DO 130 II=BR,ER
      MAC=RCONT(II)
      I=ROWE(MAC)
      KK=INROW(MAC)

      IF(KK.EQ.0)
     1    THEN
C             NO OPERATIONS TO BE PRINTED  SKIP
              WRITE(6,9070) II,MAC
              GO TO 130
          ENDIF
      DD=DUK1(MAC)
      IF(I.GT.0)
     1    THEN
              DO 10 J=1,I
                  K=LOCC(OCOL(KK))
                  ISPOT(K)=ISIGN(1)
                  KK=NEXSR(KK)
   10         CONTINUE
          ENDIF

      IF(SUP.EQ.0)
     1    THEN
              KK=DUK2(MAC)
              MAK=DUP2(MAC)
              IF(DD.GT.0)
     1            THEN
                      DO 15 J=1,DD
                      K=LOCC(DUM1(KK))
                      ISPOT(K)=ISIGN(3)
                      KK=DUM3(KK)
   15                 CONTINUE
                  ENDIF
          ELSE
              MAK=MAC
          ENDIF
      IF(WIDTH.LE.40)
     1    THEN
              WRITE(6,9010) II,MAK, (ISPOT(L),L=BC,EC)
          ELSEIF(WIDTH.LE.61)
     1            THEN
                      WRITE(6,9030) II,MAK, (ISPOT(L),L=BC,EC)
                  ELSE
                      WRITE(6,9050) II,MAK, (ISPOT(L),L=BC,EC)
                  ENDIF

C      CLEAR THE MATRIX READY TO BE USED AGAIN

      KK=INROW(MAC)

      DO 20 J=1,I
      K=LOCC(OCOL(KK))
      ISPOT(K)=ISIGN(2)
      KK=NEXSR(KK)
   20 CONTINUE

      DD=DUK1(MAC)
      IF(SUP.EQ.0.AND.DD.GT.0)
     1    THEN
              KK=DUK2(MAC)
              DO 25 J=1,DD
              K=LOCC(DUM1(KK))
              ISPOT(K)=ISIGN(2)
              KK=DUM3(KK)
   25         CONTINUE
          ENDIF

  130 CONTINUE
      WRITE(6,9530)
 9530 FORMAT(1X,///)

      RETURN

      END
```

```
PROGRAM salesv02(tourdata, output, maketm, totltm, makecs, totlcs, input
   /);

CONST
   maxcity = 60;
   infinity = 9999;

TYPE
   city = 0 .. maxcity;
   distance = 0 .. infinity;
   nodeptr = ^ anode;
   anode = PACKED RECORD
                     town: city;
                     nextnode: nodeptr;
                     linkfixed: boolean;
                  END;
   opmode =
      (alongrow, alongcol);
   printmode =
      (partial, infull);
   improvement =
      (threearc, fourarc);
   construction =
      (dolittle, shortlink, shadowlink, acircuit);
   xchangemode =
      (case0, case1, case2, case3, case4, case5);
   headptr = ^ headofchain;
   headofchain = PACKED RECORD
                           firstlink, sentinel: nodeptr;
                           nexthead: headptr;
                        END;

VAR
   tourdata, maketm, totltm, makecs, totlcs: text;
   n, ntownchange: city;
   tourlength, reducedfactor, problemno, starttime, timeelapsed,
      iteration, areduction, breduction: integer;
   c: ARRAY
      [1..maxcity, 1..maxcity] OF distance;
   rowgain: ARRAY
      [1..maxcity] OF PACKED RECORD
                                rowreduced: distance;
                                mincol, nextsmcol: city;
                                getoutok: boolean;
                             END;
   colgain: ARRAY
      [1..maxcity] OF PACKED RECORD
                                colreduced: distance;
                                minrow, nextsmrow: city;
                                getinok: boolean;
                             END;
   finaltime, finalcost: ARRAY
      [construction, improvement] OF integer;
   contime, concost: ARRAY
      [construction] OF integer;
   firsthead, sparehead: headptr;
   atown1, atown2, atown3, btown1, btown2, btown3, btown4, townchfirst,
      townchlast: nodeptr;
   change: boolean;
   optimising: improvement;
   starting: construction;
```

```
PROCEDURE readinput;

   VAR
      i, j: city;

   BEGIN
      reset(tourdata);
      read(tourdata, n, problemno);
      FOR i := 1 TO n DO
         FOR j := 1 TO n DO
            read(tourdata, c[i, j]);
      FOR i := 1 TO n DO
         c[i, i] := infinity;
   END {readinput} ;


PROCEDURE initialisation;

   VAR
      i: city;

   BEGIN
      FOR i := 1 TO n DO
         BEGIN
            rowgain[i].getoutok := true;
            colgain[i].getinok := true;
         END;
      firsthead := NIL;
      sparehead := NIL;
      townchfirst := NIL;
      townchlast := NIL;
      ntownchange := 0;
   END {initialisation} ;


PROCEDURE garbagecollection(VAR tourhead: headptr);

   VAR
      headnode: nodeptr;


   PROCEDURE collectgarbage(headnode: nodeptr);

      VAR
         thisone, nextone: nodeptr;

      BEGIN
         thisone := headnode;
         WHILE thisone <> NIL DO
            BEGIN
               nextone := thisone ∧.nextnode;
               dispose(thisone);
               thisone := nextone;
            END;
      END {collectgarbage} ;


   BEGIN {garbagecollection}
      IF tourhead <> NIL THEN
         BEGIN
            headnode := tourhead ∧.firstlink;
            collectgarbage(headnode);
            dispose(tourhead);
            tourhead := NIL;
         END;
      IF townchfirst <> NIL THEN
         BEGIN
            headnode := townchfirst;
            collectgarbage(headnode);
            townchfirst := NIL;
            townchlast := NIL;
            ntownchange := 0;
         END;
   END {garbagecollection} ;


PROCEDURE tourlists(printing: printmode);

   VAR
      thischain: headptr;
      thisnode: nodeptr;
      acity: city;
      i: integer;

   BEGIN
      thischain := firsthead;
      IF thischain = NIL
      THEN
         writeln(' NO TOUR ')
      ELSE
         writeln(' THE TOUR ');
      WHILE thischain <> NIL DO
         BEGIN
            i := 0;
            thisnode := thischain ^.firstlink;
            WHILE thisnode <> NIL DO
               BEGIN
                  acity := thisnode ^.town;
                  write(acity: 4);
                  thisnode := thisnode ^.nextnode;
                  i := i + 1;
                  IF i = 15 THEN
                     BEGIN
                        writeln;
                        i := 0;
                     END;
               END;
            IF (printing = infull) OR (starting = acircuit) THEN
               BEGIN
                  acity := thischain ^.firstlink ^.town;
                  write(acity: 4);
               END;
            writeln;
            thischain := thischain ^.nexthead;
         END;
   END {tourlists} ;


PROCEDURE writematrix;

   VAR
      i, j, k: city;
      cost: distance;

   BEGIN
      write(' ': 4);
      FOR i := 1 TO n DO
         IF colgain[i].getinok THEN
            write(i: 4);
      writeln;
      writeln;
      FOR i := 1 TO n DO
         IF rowgain[i].getoutok
         THEN
            BEGIN
               write(i: 4);
               FOR j := 1 TO n DO
                  IF colgain[j].getinok THEN
                     write(c[i, j]: 4);
               WITH rowgain[i] DO
                  BEGIN
                     k := mincol;
                     cost := rowreduced
                  END;
               writeln(cost: 4, k: 3);
            END;
      writeln;
      IF (starting = dolittle) OR (starting = shadowlink)
      THEN
         BEGIN
            write(' ': 4);
            FOR i := 1 TO n DO
               WITH colgain[i] DO
                  IF getinok THEN
                     BEGIN
                        cost := colreduced;
                        write(cost: 4);
                     END;
            writeln;
            write(' ': 4);
            FOR i := 1 TO n DO
               WITH colgain[i] DO
                  IF getinok THEN
                     BEGIN
                        k := minrow;
                        write(k: 4);
                     END;
            writeln;
         END;
   END {writematrix} ;


PROCEDURE findsmallest(fromcity: city);

   VAR
      tiny: integer;
      smallcity, tocity: integer;

   BEGIN
      tiny := infinity + 1;
      smallcity := 0;
      FOR tocity := 1 TO n DO
         IF colgain[tocity].getinok THEN
            IF c[fromcity, tocity] < tiny THEN
               BEGIN
                  tiny := c[fromcity, tocity];
                  smallcity := tocity;
               END;
      WITH rowgain[fromcity] DO
```

```
         BEGIN
            mincol := smallcity;
            rowreduced := c[fromcity, mincol];
         END;
   END {findsmallest} ;


PROCEDURE findtwosmallest(acity: city; roworcol: opmode);

   VAR
      tiny1, tiny2: integer;
      city1, city2, fromcity, tocity: integer;

   BEGIN
      tiny1 := infinity + 1;
      tiny2 := infinity + 2;
      city1 := 0;
      city2 := 0;
      IF roworcol = alongrow
      THEN
         BEGIN
            fromcity := acity;
            FOR tocity := 1 TO n DO
               IF colgain[tocity].getinok
               THEN
                  IF c[fromcity, tocity] < tiny2
                  THEN
                     IF c[fromcity, tocity] < tiny1
                     THEN
                        BEGIN
                           tiny2 := tiny1;
                           city2 := city1;
                           tiny1 := c[fromcity, tocity];
                           city1 := tocity;
                        END
                     ELSE
                        BEGIN
                           tiny2 := c[fromcity, tocity];
                           city2 := tocity;
                        END;
            WITH rowgain[fromcity] DO
               BEGIN
                  mincol := city1;
                  nextsmcol := city2;
                  rowreduced := c[fromcity, city2] - c[fromcity, city1];
               END;
         END
      ELSE
         BEGIN
            tocity := acity;
            FOR fromcity := 1 TO n DO
               IF rowgain[fromcity].getoutok
               THEN
                  IF c[fromcity, tocity] < tiny2
                  THEN
                     IF c[fromcity, tocity] < tiny1
                     THEN
                        BEGIN
                           tiny2 := tiny1;
                           city2 := city1;
                           tiny1 := c[fromcity, tocity];
                           city1 := fromcity;
                        END
                     ELSE
                        BEGIN
                           tiny2 := c[fromcity, tocity];
                           city2 := fromcity;
                        END;
            WITH colgain[tocity] DO
               BEGIN
                  minrow := city1;
                  nextsmrow := city2;
                  colreduced := c[city2, tocity] - c[city1, tocity];
               END;
         END;
   END {findtwosmallest} ;


PROCEDURE updatematrix(addfrom, addto: city);

   BEGIN
      IF (starting = dolittle) OR (starting = shadowlink)
      THEN
         BEGIN
            WITH rowgain[addfrom] DO
               IF (mincol = addto) OR (nextsmcol = addto) THEN
                  findtwosmallest(addfrom, alongrow);
            WITH colgain[addto] DO
               IF (minrow = addfrom) OR (nextsmrow = addfrom) THEN
                  findtwosmallest(addto, alongcol);
         END
      ELSE
         IF starting = shortlink THEN
            WITH rowgain[addfrom] DO
               IF mincol = addto THEN
                  findsmallest(addfrom);
   END {updatematrix} ;


PROCEDURE updatecolumn(totown: city);

   VAR
      thisrow: nodeptr;
      i, chrow, aminrow, anextsmrow, city1, city2: city;
      tiny1, tiny2: integer;


   PROCEDURE twoup(chrow: city);

      BEGIN
         IF c[chrow, totown] < tiny2
         THEN
            IF c[chrow, totown] < tiny1
            THEN
               BEGIN
                  tiny2 := tiny1;
                  city2 := city1;
                  tiny1 := c[chrow, totown];
                  city1 := chrow;
               END
            ELSE
               BEGIN
                  tiny2 := c[chrow, totown];
                  city2 := chrow;
               END;
      END {twoup} ;
   BEGIN  {updatecolumn}
      WITH colgain[totown] DO
         BEGIN
            thisrow := townchfirst;
            city1 := minrow;
            city2 := nextsmrow;
            tiny1 := infinity;
            tiny2 := infinity;
            aminrow := minrow;
            anextsmrow := nextsmrow;
            twoup(aminrow);
            twoup(anextsmrow);
            FOR i := 1 TO ntownchange DO
               BEGIN
                  chrow := thisrow ∧.town;
                  twoup(chrow);
                  thisrow := thisrow ∧.nextnode;
               END;
            minrow := city1;
            nextsmrow := city2;
            colreduced := c[city2, totown] - c[city1, totown];
         END;
   END  {updatecolumn} ;


PROCEDURE updaterows;

   VAR
      thiscol: nodeptr;
      fromtown, i, chcol, amincol, anextsmcol, city1, city2: city;
      tiny1, tiny2: integer;


   PROCEDURE twouprow(chcol: city);

      BEGIN
         IF c[fromtown, chcol] < tiny2
         THEN
            IF c[fromtown, chcol] < tiny1
            THEN
               BEGIN
                  tiny2 := tiny1;
                  city2 := city1;
                  tiny1 := c[fromtown, chcol];
                  city1 := chcol;
               END
            ELSE
               BEGIN
                  tiny2 := c[fromtown, chcol];
                  city2 := chcol;
               END;
      END  {twouprow} ;


   BEGIN  {updaterows}
      FOR fromtown := 1 TO n DO
         WITH rowgain[fromtown] DO
            IF getoutok
            THEN
               BEGIN
                  thiscol := townchfirst;
                  city1 := mincol;
                  city2 := nextsmcol;
                  tiny1 := infinity;
                  tiny2 := infinity;
                  amincol := mincol;
                  anextsmcol := nextsmcol;
                  twouprow(amincol);
                  twouprow(anextsmcol);
                  FOR i := 1 TO ntownchange DO
                     BEGIN
                        chcol := thiscol ∧.town;
                        twouprow(chcol);
                        thiscol := thiscol ∧.nextnode;
                     END;
                  mincol := city1;
                  nextsmcol := city2;
                  rowreduced := c[fromtown, city2] - c[fromtown, city1];
               END;
   END {updaterows} ;


PROCEDURE addtotownlist(atown: city);

   VAR
      anewnode: nodeptr;

   BEGIN
      IF townchfirst = NIL
      THEN
         BEGIN
            new(anewnode);
            townchfirst := anewnode;
            townchlast := townchfirst;
            WITH anewnode ∧ DO
               BEGIN
                  nextnode := NIL;
                  town := atown;
               END;
         END
      ELSE
         IF townchlast ∧.nextnode = NIL
         THEN
            BEGIN
               new(anewnode);
               townchlast ∧.nextnode := anewnode;
               townchlast := anewnode;
               WITH anewnode ∧ DO
                  BEGIN
                     nextnode := NIL;
                     town := atown;
                  END;
            END
         ELSE
            BEGIN
               townchlast := townchlast ∧.nextnode;
               townchlast ∧.town := atown;
            END;
      ntownchange := ntownchange + 1;
   END {addtotownlist} ;


PROCEDURE reduceable(linksassigned: integer; VAR fromcity, tocity: city;
   roworcol: opmode);

   VAR
      i: city;

   BEGIN
      IF linksassigned = 0
      THEN
         FOR i := 1 TO n DO
            BEGIN
               findtwosmallest(i, roworcol);
            END
      ELSE
         IF roworcol = alongrow
         THEN
            BEGIN
               FOR i := 1 TO n DO
                  WITH rowgain[i] DO
                     IF getoutok AND ((mincol = tocity) OR (nextsmcol =
                        tocity))
                     THEN
                        findtwosmallest(i, alongrow);
            END
         ELSE
            FOR i := 1 TO n DO
               WITH colgain[i] DO
                  IF getinok THEN
                     IF (minrow = fromcity) OR (nextsmrow = fromcity)
                     THEN
                        findtwosmallest(i, alongcol)
                     ELSE
                        updatecolumn(i);
   END {reduceable} ;


FUNCTION sumoffactors: integer;

   VAR
      i: city;
      sum: integer;

   BEGIN
      sum := 0;
      FOR i := 1 TO n DO
         WITH rowgain[i] DO
            IF getoutok THEN
               sum := sum + c[i, mincol];
      FOR i := 1 TO n DO
         WITH colgain[i] DO
            IF getinok THEN
               sum := sum + c[minrow, i];
      sumoffactors := sum;
   END {sumoffactors} ;


PROCEDURE reducecost(VAR row, col: city; along: opmode);

   VAR
      reduce: distance;
      i: city;

   BEGIN
      reduce := c[row, col];
      IF reduce <> 0
      THEN
         BEGIN
            IF along = alongrow
            THEN
               BEGIN
                  FOR i := 1 TO n DO
                     IF colgain[i].getinok THEN
                        c[row, i] := c[row, i] - reduce;
                  addtotownlist(row);
               END
            ELSE
               BEGIN
                  FOR i := 1 TO n DO
                     IF rowgain[i].getoutok THEN
                        c[i, col] := c[i, col] - reduce;
                  addtotownlist(col);
               END;
         END;
   END  {reducecost} ;


PROCEDURE reducematrix(along: opmode);

   VAR
      i, j: city;

   BEGIN
      IF along = alongrow
      THEN
         BEGIN
            FOR i := 1 TO n DO
               WITH rowgain[i] DO
                  IF getoutok THEN
                     BEGIN
                        j := mincol;
                        reducecost(i, j, alongrow);
                        reducedfactor := reducedfactor + c[i, j];
                     END;
         END
      ELSE
         BEGIN
            FOR i := 1 TO n DO
               WITH colgain[i] DO
                  IF getinok THEN
                     BEGIN
                        j := minrow;
                        reducecost(j, i, alongcol);
                        reducedfactor := reducedfactor + c[j, i];
                     END;
         END;
   END  {reducematrix} ;


PROCEDURE nextlittlelink(VAR fromcity, tocity: city);

   VAR
      i, j: city;
      shadowcost, smallofrow: integer;

   BEGIN
      shadowcost := - 1;
      FOR i := 1 TO n DO
         WITH rowgain[i] DO
```

```
            IF getoutok
            THEN
               IF rowreduced <> 0
               THEN
                  BEGIN
                     IF (rowreduced + colgain[mincol].colreduced) >
                        shadowcost
                     THEN
                        BEGIN
                           fromcity := i;
                           tocity := mincol;
                           shadowcost := rowreduced + colgain[mincol].
                              colreduced;
                        END;
                  END
               ELSE
                  BEGIN
                     smallofrow := c[i, mincol];
                     FOR j := 1 TO n DO
                        WITH colgain[j] DO
                           IF getinok
                           THEN
                              IF c[i, j] = smallofrow THEN
                                 IF (rowreduced + colreduced) >
                                    shadowcost
                                 THEN
                                    BEGIN
                                       fromcity := i;
                                       tocity := j;
                                       shadowcost := rowreduced +
                                          colreduced;
                                    END;
                  END;
   END {nextlittlelink} ;


FUNCTION lastinalink(fromcity: city; VAR thechain: headptr): boolean;

   VAR
      thischain: headptr;
      found: boolean;

   BEGIN
      found := false;
      thischain := firsthead;
      WHILE ((thischain <> NIL) AND (NOT found)) DO
         IF thischain ∧.sentinel ∧.town = fromcity
         THEN
            found := true
         ELSE
            thischain := thischain ∧.nexthead;
      thechain := thischain;
      lastinalink := found;
   END {lastinalink} ;


FUNCTION firstinalink(tocity: city; VAR lasthead: headptr): boolean;

   VAR
      found: boolean;
      thishead, afterthis: headptr;
      link: nodeptr;

   BEGIN
      found := false;
      thishead := NIL;
      afterthis := firsthead;
      WHILE ((afterthis <> NIL) AND (NOT found)) DO
         IF afterthis ∧.firstlink ∧.town = tocity
         THEN
            found := true
         ELSE
            BEGIN
               thishead := afterthis;
               afterthis := afterthis ∧.nexthead;
            END;
      lasthead := thishead;
      firstinalink := found;
   END {firstinalink} ;


PROCEDURE joinhead(fromcity: city; lasthead: headptr);

   VAR
      thishead: headptr;
      newnode: nodeptr;

   BEGIN
      IF lasthead = NIL
      THEN
         thishead := firsthead
      ELSE
         thishead := lasthead ∧.nexthead;
      new(newnode);
      WITH thishead ∧, newnode ∧ DO
         BEGIN
            nextnode := firstlink;
            linkfixed := false;
            town := fromcity;
            firstlink := newnode;
         END;
   END {joinhead} ;


PROCEDURE jointail(tocity: city; thischain: headptr);

   VAR
      newnode: nodeptr;

   BEGIN
      new(newnode);
      thischain ∧.sentinel ∧.nextnode := newnode;
      thischain ∧.sentinel := newnode;
      WITH newnode ∧ DO
         BEGIN
            town := tocity;
            nextnode := NIL;
            linkfixed := false;
         END;
   END {jointail} ;


PROCEDURE makenewchain(fromcity, tocity: city; lasthead: headptr);

   VAR
      newhead: headptr;
      nodefrom, nodeto: nodeptr;

   BEGIN
      new(newhead);
      new(nodefrom);
      new(nodeto);
      IF lasthead = NIL
      THEN
         firsthead := newhead
      ELSE
         lasthead ^.nexthead := newhead;
      WITH newhead ^ DO
         BEGIN
            firstlink := nodefrom;
            sentinel := nodeto;
            nexthead := NIL;
         END;
      WITH nodefrom ^ DO
         BEGIN
            town := fromcity;
            nextnode := nodeto;
            linkfixed := false;
         END;
      WITH nodeto ^ DO
         BEGIN
            town := tocity;
            nextnode := NIL;
            linkfixed := false;
         END;
   END {makenewchain} ;


PROCEDURE jointwochains(lasthead, secondchain: headptr);

   VAR
      thishead: headptr;
      lastnode: nodeptr;

   BEGIN
      lastnode := secondchain ^.sentinel;
      IF lasthead = NIL
      THEN
         thishead := firsthead
      ELSE
         thishead := lasthead ^.nexthead;
      lastnode ^.nextnode := thishead ^.firstlink;
      IF lasthead = NIL
      THEN
         firsthead := thishead ^.nexthead
      ELSE
         lasthead ^.nexthead := thishead ^.nexthead;
      secondchain ^.sentinel := thishead ^.sentinel;
      dispose(thishead);
   END {jointwochains} ;


PROCEDURE addanotherlink(links: integer; fromcity, tocity: city);

   VAR
      first, last: boolean;
      headbeforefirst, secondchain: headptr;
      firstcity, lastcity: city;

   BEGIN
      first := firstinalink(tocity, headbeforefirst);
      last := lastinalink(fromcity, secondchain);
      IF first THEN
         IF headbeforefirst = NIL
         THEN
            lastcity := firsthead ∧.sentinel ∧.town
         ELSE
            lastcity := headbeforefirst ∧.nexthead ∧.sentinel ∧.town;
      IF last THEN
         firstcity := secondchain ∧.firstlink ∧.town;
      IF first
      THEN
         IF last
         THEN
            BEGIN
               jointwochains(headbeforefirst, secondchain);
               c[lastcity, firstcity] := infinity;
               IF links <> (n - 1) THEN
                  updatematrix(lastcity, firstcity);
            END
         ELSE
            BEGIN
               joinhead(fromcity, headbeforefirst);
               c[lastcity, fromcity] := infinity;
               IF links <> (n - 1) THEN
                  updatematrix(lastcity, fromcity);
            END
      ELSE
         IF last
         THEN
            BEGIN
               jointail(tocity, secondchain);
               c[tocity, firstcity] := infinity;
               IF links <> (n - 1) THEN
                  updatematrix(tocity, firstcity);
            END
         ELSE
            BEGIN
               makenewchain(fromcity, tocity, headbeforefirst);
               c[tocity, fromcity] := infinity;
               updatematrix(tocity, fromcity);
            END;
   END  {addanotherlink} ;


PROCEDURE contractmatrix(fromcity, tocity: city);

   VAR
      i: city;

   BEGIN
      rowgain[fromcity].getoutok := false;
      colgain[tocity].getinok := false;
   END  {contractmatrix} ;


PROCEDURE littletsp;

   VAR
      linksassigned: integer;
      fromcity, tocity: city;

   BEGIN
      linksassigned := 0;
      REPEAT
         ntownchange := 0;
         reduceable(linksassigned, fromcity, tocity, alongrow);
         reducematrix(alongrow);
         reduceable(linksassigned, fromcity, tocity, alongcol);
         ntownchange := 0;
         townchlast := townchfirst;
         reducematrix(alongcol);
         updaterows;
         nextlittlelink(fromcity, tocity);
         IF problemno > 400 THEN
            BEGIN
               writeln(' EXIT NEXTLITTLELINK ', fromcity: 4, tocity: 4);
               writeln;
               writematrix;
            END;
         contractmatrix(fromcity, tocity);
         linksassigned := linksassigned + 1;
         addanotherlink(linksassigned, fromcity, tocity);
         IF problemno > 300 THEN
            tourlists(partial);
      UNTIL linksassigned = (n - 1);
   END  {littletsp} ;


PROCEDURE neighbourmatrix(linksassigned: integer; VAR tocity: city);

   VAR
      i: city;

   BEGIN
      IF linksassigned = 0
      THEN
         FOR i := 1 TO n DO
            findsmallest(i)
      ELSE
         BEGIN
            FOR i := 1 TO n DO
               WITH rowgain[i] DO
                  IF getoutok AND (mincol = tocity) THEN
                     findsmallest(i);
         END;
   END  {neighbourmatrix} ;


PROCEDURE nextneighbour(VAR fromcity, tocity: city);

   VAR
      i: city;
      tiny: integer;

   BEGIN
      tiny := infinity + 1;
      FOR i := 1 TO n DO
         WITH rowgain[i] DO
            IF getoutok THEN
               IF rowreduced < tiny THEN
                  BEGIN
                     tiny := rowreduced;
                     fromcity := i;
                     tocity := mincol;
                  END;
   END  {nextneighbour} ;


PROCEDURE nearestneighbour;

   VAR
      linksassigned: integer;
      fromcity, tocity: city;

   BEGIN
      linksassigned := 0;
      REPEAT
         neighbourmatrix(linksassigned, tocity);
         IF problemno > 400 THEN
            writematrix;
         nextneighbour(fromcity, tocity);
         contractmatrix(fromcity, tocity);
         linksassigned := linksassigned + 1;
         addanotherlink(linksassigned, fromcity, tocity);
         IF problemno > 400 THEN
            BEGIN
               writeln(' EXIT NEXTNEIGHBOUR ', fromcity: 4, tocity: 4);
               tourlists(partial);
            END;
      UNTIL linksassigned = (n - 1);
   END  {nearestneighbour} ;


PROCEDURE shadowmatrix(linksassigned: integer; VAR fromcity, tocity:
   city);

   VAR
      i: city;

   BEGIN
      IF linksassigned = 0
      THEN
         FOR i := 1 TO n DO
            BEGIN
               findtwosmallest(i, alongrow);
               findtwosmallest(i, alongcol);
            END
      ELSE
         BEGIN
            FOR i := 1 TO n DO
               WITH rowgain[i] DO
                  IF getoutok AND ((mincol = tocity) OR (nextsmcol =
                     tocity))
                  THEN
                     findtwosmallest(i, alongrow);
            FOR i := 1 TO n DO
               WITH colgain[i] DO
                  IF getinok AND ((minrow = fromcity) OR (nextsmrow =
                     fromcity))
                  THEN
                     findtwosmallest(i, alongcol);
         END;
   END  {shadowmatrix} ;


PROCEDURE nextshadow(VAR fromcity, tocity: city);

```

```
   VAR
      i, afromcity, atocity: city;
      large: integer;

   BEGIN
      large := - infinity;
      FOR i := 1 TO n DO
         WITH rowgain[i] DO
            IF getoutok THEN
               IF rowreduced > large THEN
                  BEGIN
                     large := rowreduced;
                     afromcity := i;
                     atocity := mincol;
                  END;
      FOR i := 1 TO n DO
         WITH colgain[i] DO
            IF getinok THEN
               IF colreduced > large THEN
                  BEGIN
                     large := colreduced;
                     afromcity := minrow;
                     atocity := i;
                  END;
      fromcity := afromcity;
      tocity := atocity;
   END {nextshadow} ;


PROCEDURE shadowneighbour;

   VAR
      linksassigned: integer;
      fromcity, tocity: city;
      roworcol: opmode;

   BEGIN
      linksassigned := 0;
      REPEAT
         shadowmatrix(linksassigned, fromcity, tocity);
         IF problemno > 300 THEN
            writematrix;
         nextshadow(fromcity, tocity);
         IF problemno > 300 THEN
            BEGIN
               writeln(' EXIT NEXTSHADOW ', fromcity: 4, tocity: 4);
               tourlists(partial);
            END;
         contractmatrix(fromcity, tocity);
         linksassigned := linksassigned + 1;
         addanotherlink(linksassigned, fromcity, tocity);
         IF problemno > 400 THEN
            tourlists(partial);
      UNTIL linksassigned = (n - 1);
   END {shadowneighbour} ;


PROCEDURE tourstarter(VAR fromcity, tocity: city);

   VAR
      i, j: city;
      fromtown, totown, small: integer;
      ahead: headptr;
      townptr1, townptr2: nodeptr;

   BEGIN
      small := infinity;
      fromtown := 0;
      totown := 0;
      FOR i := 1 TO n - 1 DO
         FOR j := i TO n DO
            IF (c[i, j] + c[j, i]) < small THEN
               BEGIN
                  fromtown := i;
                  totown := j;
                  small := c[i, j] + c[j, i];
               END;
      new(ahead);
      new(townptr1);
      new(townptr2);
      firsthead := ahead;
      WITH firsthead ∧ DO
         BEGIN
            firstlink := townptr1;
            sentinel := townptr2;
            nexthead := NIL;
         END;
      WITH townptr1 ∧ DO
         BEGIN
            town := fromtown;
            nextnode := townptr2;
         END;
      WITH townptr2 ∧ DO
         BEGIN
            town := totown;
            nextnode := NIL;
         END;
      fromcity := fromtown;
      tocity := totown;
   END {tourstarter} ;


PROCEDURE inserttown(fromtown, newtown, totown: city);

   VAR
      townptr, newcity: nodeptr;

   BEGIN
      new(newcity);
      townptr := firsthead ∧.firstlink;
      WHILE fromtown <> townptr ∧.town DO
         townptr := townptr ∧.nextnode;
      WITH newcity ∧ DO
         BEGIN
            nextnode := townptr ∧.nextnode;
            town := newtown;
         END;
      townptr ∧.nextnode := newcity;
      IF fromtown = firsthead ∧.sentinel ∧.town THEN
         firsthead ∧.sentinel := newcity;
   END {inserttown} ;


PROCEDURE tourinsertion(VAR tourlength: integer);

   VAR
   assigned: PACKED ARRAY
      [1..maxcity] OF boolean;
   i, fromcity, tocity, newcity: city;
   currentcost, citiesassigned: integer;


PROCEDURE towntoinsert(VAR fromtown, newtown, totown: city);

   VAR
      i, lasttown, nexttown, before, this, after: city;
      townptr: nodeptr;
      small: integer;

   BEGIN
      small := infinity;
      FOR i := 1 TO n DO
         IF NOT assigned[i]
         THEN
            BEGIN
               townptr := firsthead ∧.firstlink;
               WHILE townptr <> NIL DO
                  BEGIN
                     lasttown := townptr ∧.town;
                     IF townptr = firsthead ∧.sentinel
                     THEN
                        nexttown := firsthead ∧.firstlink ∧.town
                     ELSE
                        nexttown := townptr ∧.nextnode ∧.town;
                     IF (c[lasttown, i] + c[i, nexttown] - c[lasttown
                        , nexttown]) < small
                     THEN
                        BEGIN
                           small := c[lasttown, i] + c[i, nexttown] -
                              c[lasttown, nexttown];
                           before := lasttown;
                           this := i;
                           after := nexttown;
                        END;
                     townptr := townptr ∧.nextnode;
                  END;
            END;
      fromtown := before;
      newtown := this;
      totown := after;
   END  {towntoinsert} ;


BEGIN  {tourinsertion}
   FOR i := 1 TO n DO
      assigned[i] := false;
   tourstarter(fromcity, tocity);
   IF problemno > 400 THEN
      tourlists(infull);
   assigned[fromcity] := true;
   assigned[tocity] := true;
   currentcost := c[fromcity, tocity] + c[tocity, fromcity];
   citiesassigned := 2;
   REPEAT
      towntoinsert(fromcity, newcity, tocity);
      inserttown(fromcity, newcity, tocity);
      assigned[newcity] := true;
      IF problemno > 400 THEN
         BEGIN
```

```
               tourlists(infull);
               writeln(' EXIT TOWNTOINSERT: INSERT ', newcity: 4,
                  ' BETWEEN ', fromcity: 4, tocity: 4);
            END;
         citiesassigned := citiesassigned + 1;
         currentcost := currentcost + c[fromcity, newcity] + c[newcity,
            tocity] - c[fromcity, tocity];
      UNTIL citiesassigned = n;
      tourlength := currentcost;
   END {tourinsertion} ;


PROCEDURE copytour;

   VAR
      anewhead: headptr;
      lastnode, thisnode, oldone: nodeptr;
      firstround: boolean;

   BEGIN
      firstround := true;
      IF sparehead <> NIL THEN
         garbagecollection(sparehead);
      IF firsthead <> NIL
      THEN
         BEGIN
            new(anewhead);
            sparehead := anewhead;
            oldone := firsthead ∧.firstlink;
            WHILE oldone <> NIL DO
               WITH oldone ∧ DO
                  BEGIN
                     new(thisnode);
                     IF firstround
                     THEN
                        BEGIN
                           sparehead ∧.firstlink := thisnode;
                           firstround := false;
                        END
                     ELSE
                        lastnode ∧.nextnode := thisnode;
                     thisnode ∧.town := town;
                     thisnode ∧.linkfixed := linkfixed;
                     lastnode := thisnode;
                     oldone := nextnode;
                  END;
            sparehead ∧.sentinel := lastnode;
            sparehead ∧.nexthead := NIL;
         END;
      lastnode ∧.nextnode := NIL;
   END {copytour} ;


PROCEDURE tourcost(VAR finalcost: integer);

   VAR
      cost: integer;
      this, last: nodeptr;

   BEGIN
      cost := 0;
      IF firsthead <> NIL
      THEN
```

```
         BEGIN
            last := firsthead ^.sentinel;
            this := firsthead ^.firstlink;
            WHILE this <> NIL DO
               BEGIN
                  cost := cost + c[last ^.town, this ^.town];
                  last := this;
                  this := this ^.nextnode;
               END;
         END;
      finalcost := cost;
   END {tourcost} ;


PROCEDURE last2but1(VAR lastbut2, lastbut1: nodeptr);

   VAR
      k: city;
      townptr: nodeptr;

   BEGIN
      townptr := firsthead ^.firstlink;
      FOR k := 1 TO n - 3 DO
         townptr := townptr ^.nextnode;
      lastbut2 := townptr;
      lastbut1 := lastbut2 ^.nextnode;
      IF lastbut1 ^.nextnode <> firsthead ^.sentinel THEN
         writeln(' TOUR ERROR FOUND BY LAST2BUT1');
   END {last2but1} ;


FUNCTION good3opt(townptr1, townptr2, townptr3: nodeptr; VAR benefit:
   integer): boolean;

   VAR
      f1, f2, f3, f4, t1, t2, t3: city;

   BEGIN
      f1 := townptr1 ^.town;
      t1 := townptr1 ^.nextnode ^.town;
      f2 := townptr2 ^.town;
      t2 := townptr2 ^.nextnode ^.town;
      f3 := townptr3 ^.town;
      IF townptr3 = firsthead ^.sentinel
      THEN
         t3 := firsthead ^.firstlink ^.town
      ELSE
         t3 := townptr3 ^.nextnode ^.town;
      benefit := c[f1, t1] + c[f2, t2] + c[f3, t3] - (c[f1, t2] + c[f3,
         t1] + c[f2, t3]);
      IF benefit > 0
      THEN
         good3opt := true
      ELSE
         good3opt := false;
   END {good3opt} ;


PROCEDURE change3opt(townptr1, townptr2, townptr3: nodeptr);

   VAR
      nextto1, nextto2, nextto3: nodeptr;

   BEGIN
      nextto1 := townptr1 ∧.nextnode;
      nextto2 := townptr2 ∧.nextnode;
      nextto3 := townptr3 ∧.nextnode;
      townptr1 ∧.nextnode := nextto2;
      townptr2 ∧.nextnode := nextto3;
      townptr3 ∧.nextnode := nextto1;
      IF nextto3 = NIL THEN
         firsthead ∧.sentinel := townptr2;
   END {change3opt} ;


PROCEDURE threeopta(VAR town1, town2, town3: nodeptr; VAR reduce:
   integer);

   VAR
      lastbut2, lastbut1, lastone, bestptr1, bestptr2, bestptr3,
         townptr1, townptr2, townptr3: nodeptr;
      reduction, bestreduction: integer;
      beneficial: boolean;

   BEGIN
      bestreduction := - infinity;
      WITH firsthead ∧ DO
         BEGIN
            lastone := sentinel;
            townptr1 := firstlink;
         END;
      last2but1(lastbut2, lastbut1);
      WHILE townptr1 <> lastbut1 DO
         BEGIN
            townptr2 := townptr1 ∧.nextnode;
            WHILE townptr2 <> lastone DO
               BEGIN
                  townptr3 := townptr2 ∧.nextnode;
                  WHILE townptr3 <> NIL DO
                     BEGIN
                        beneficial := good3opt(townptr1, townptr2,
                           townptr3, reduction);
                        IF beneficial AND (reduction > bestreduction)
                        THEN
                           BEGIN
                              bestptr1 := townptr1;
                              bestptr2 := townptr2;
                              bestptr3 := townptr3;
                              bestreduction := reduction;
                           END;
                        townptr3 := townptr3 ∧.nextnode;
                     END;
                  townptr2 := townptr2 ∧.nextnode;
               END;
            townptr1 := townptr1 ∧.nextnode;
         END;
      town1 := bestptr1;
      town2 := bestptr2;
      town3 := bestptr3;
      reduce := bestreduction;
   END {threeopta} ;


FUNCTION paralbefore2(ptrone, ptrtwo: nodeptr): boolean;

   VAR
```

```
      this: nodeptr;

   BEGIN
      this := ptrone;
      WHILE (this <> ptrtwo) AND (this <> NIL) DO
         this := this ∧.nextnode;
      IF this = ptrtwo
      THEN
         paralbefore2 := true
      ELSE
         paralbefore2 := false;
   END  {paralbefore2} ;


FUNCTION nextinthetour(i: nodeptr): nodeptr;

   VAR
      j: nodeptr;

   BEGIN
      j := i ∧.nextnode;
      IF j = NIL THEN
         j := firsthead ∧.firstlink;
      nextinthetour := j;
   END  {nextinthetour} ;


FUNCTION partial4opt(townptr1, townptr2: nodeptr): integer;

   VAR
      after1, after2: nodeptr;
      f1, t1, f2, t2: city;

   BEGIN
      f1 := townptr1 ∧.town;
      after1 := nextinthetour(townptr1);
      t1 := after1 ∧.town;
      f2 := townptr2 ∧.town;
      after2 := nextinthetour(townptr2);
      t2 := after2 ∧.town;
      partial4opt := c[f1, t1] + c[f2, t2] - c[f1, t2] - c[f2, t1];
   END  {partial4opt} ;


PROCEDURE best4opta(townptr1, townptr2: nodeptr; VAR townptr3, townptr4:
   nodeptr; VAR gain2: integer);

   VAR
      bestptr3, bestptr4, i, j, k: nodeptr;
      bestgain, again, costf3t3: integer;
      f3, f4, t3, t4: city;

   BEGIN
      bestgain := - infinity;
      i := townptr1 ∧.nextnode;
      WHILE i <> townptr2 DO
         BEGIN
            WITH i ∧ DO
               BEGIN
                  f3 := town;
                  t3 := nextnode ∧.town;
               END;
            costf3t3 := c[f3, t3];
            j := nextinthetour(townptr2);
            WHILE j <> townptr1 DO
               BEGIN
                  f4 := j ∧.town;
                  k := nextinthetour(j);
                  t4 := k ∧.town;
                  again := costf3t3 + c[f4, t4] - c[f3, t4] - c[f4, t3];
                  IF again > bestgain THEN
                     BEGIN
                        bestgain := again;
                        bestptr3 := i;
                        bestptr4 := j;
                     END;
                  j := k;
               END;
            i := i ∧.nextnode;
         END;
      townptr3 := bestptr3;
      townptr4 := bestptr4;
      gain2 := bestgain;
   END  {best4opta} ;


PROCEDURE change4a(townptr1, townptr2, townptr3, townptr4: nodeptr);

   VAR
      nextto1, nextto2, nextto3, nextto4: nodeptr;

   BEGIN
      nextto1 := townptr1 ∧.nextnode;
      nextto2 := townptr2 ∧.nextnode;
      nextto3 := townptr3 ∧.nextnode;
      nextto4 := townptr4 ∧.nextnode;
      townptr1 ∧.nextnode := nextto2;
      townptr2 ∧.nextnode := nextto1;
      townptr3 ∧.nextnode := nextto4;
      townptr4 ∧.nextnode := nextto3;
      IF nextto2 = NIL THEN
         BEGIN
            firsthead ∧.sentinel := townptr1;
            townptr1 ∧.nextnode := NIL;
         END;
      IF nextto4 = NIL THEN
         BEGIN
            firsthead ∧.sentinel := townptr3;
            townptr3 ∧.nextnode := NIL;
         END;
   END  {change4a} ;


PROCEDURE fouroptb(VAR town1, town2, town3, town4: nodeptr; VAR reduce:
   integer);

   VAR
      lastbut2, lastbut1, lastone, lastptr1, limitptr1, last1mtptr1,
         bestptr1, bestptr2, bestptr3, bestptr4, townptr1, townptr2,
         townptr3, townptr4: nodeptr;
      partgain, gain2, bestgain: integer;
      beneficial: boolean;

   BEGIN
      bestgain := - infinity;
      last2but1(lastbut2, lastbut1);
      townptr1 := firsthead ^.firstlink;
      limitptr1 := lastbut1;
      WHILE townptr1 <> limitptr1 DO
         BEGIN
            townptr2 := townptr1 ^.nextnode ^.nextnode;
            WHILE townptr2 <> NIL DO
               BEGIN
                  partgain := partial4opt(townptr1, townptr2);
                  IF partgain > 0
                  THEN
                     BEGIN
                        best4opta(townptr1, townptr2, townptr3, townptr4
                           , gain2);
                        partgain := partgain + gain2;
                        IF partgain > bestgain THEN
                           BEGIN
                              bestptr1 := townptr1;
                              bestptr2 := townptr2;
                              bestptr3 := townptr3;
                              bestptr4 := townptr4;
                              bestgain := partgain;
                           END;
                     END;
                  townptr2 := townptr2 ^.nextnode;
               END;
            townptr1 := townptr1 ^.nextnode;
         END;
      town1 := bestptr1;
      town2 := bestptr2;
      town3 := bestptr3;
      town4 := bestptr4;
      reduce := bestgain;
   END {fouroptb} ;


PROCEDURE writetofiles;

   VAR
      i: construction;
      j: improvement;

   BEGIN
      write(maketm, problemno: 4, ' ');
      write(makecs, problemno: 4, ' ');
      write(totltm, problemno: 4, ' ');
      write(totlcs, problemno: 4, ' ');
      FOR i := dolittle TO acircuit DO
         BEGIN
            write(maketm, contime[i]: 7, ' ');
            write(makecs, concost[i]: 7, ' ');
            FOR j := threearc TO fourarc DO
               BEGIN
                  write(totltm, finaltime[i, j]: 7, ' ');
                  write(totlcs, finalcost[i, j]: 7, ' ');
               END;
         END;
      writeln(maketm);
      writeln(makecs);
      writeln(totltm);
      writeln(totlcs);
   END {writetofiles} ;


```

```
BEGIN {salesv02}
   readinput;
   FOR starting := dolittle TO acircuit DO
      BEGIN
         initialisation;
         starttime := clock;
         CASE starting OF
            dolittle:
               littletsp;
            shortlink:
               nearestneighbour;
            shadowlink:
               shadowneighbour;
            acircuit:
               tourinsertion(tourlength);
         END;
         timeelapsed := clock - starttime;
         readinput;
         IF starting <> acircuit THEN
            tourcost(tourlength);
         copytour;
         contime[starting] := timeelapsed;
         concost[starting] := tourlength;
         writeln(' PROBLEM  NO ', problemno: 6, ' ': 2, starting: 2 oct,
            ' CONSTRUCTION LENGTH ', tourlength: 7,
            ' CONSTRUCTION TIME ', timeelapsed: 7);
         tourlists(infull);
         writeln;
         FOR optimising := threearc TO fourarc DO
            BEGIN
               IF optimising = threearc
               THEN
                  BEGIN
                     iteration := 0;
                     change := false;
                     starttime := clock;
                     REPEAT
                        threeopta(atown1, atown2, atown3, areduction);
                        IF areduction > 0
                        THEN
                           BEGIN
                              change3opt(atown1, atown2, atown3);
                              tourlength := tourlength - areduction;
                              iteration := iteration + 1;
                              change := true;
                           END
                        ELSE
                           change := false;
                     UNTIL NOT change;
                     timeelapsed := clock - starttime;
                     finaltime[starting, optimising] := contime[starting
                        ] + timeelapsed;
                  END
               ELSE
                  BEGIN
                     iteration := 0;
                     change := false;
                     garbagecollection(firsthead);
                     firsthead := sparehead;
                     sparehead := NIL;
                     tourcost(tourlength);
                     starttime := clock;
                     REPEAT
```

```
                        threeopta(atown1, atown2, atown3, areduction);
                        fouroptb(btown1, btown2, btown3, btown4,
                           breduction);
                        IF (areduction > 0) OR (breduction > 0)
                        THEN
                           BEGIN
                              IF areduction > breduction
                              THEN
                                 BEGIN
                                    change3opt(atown1, atown2, atown3);
                                    tourlength := tourlength -
                                       areduction;
                                 END
                              ELSE
                                 BEGIN
                                    change4a(btown1, btown2, btown3,
                                       btown4);
                                    tourlength := tourlength -
                                       breduction;
                                 END;
                              iteration := iteration + 1;
                              change := true;
                           END
                        ELSE
                           change := false;
                     UNTIL NOT change;
                     timeelapsed := clock - starttime;
                     finaltime[starting, optimising] := finaltime[
                        starting, threearc] + timeelapsed;
                  END;
               finalcost[starting, optimising] := tourlength;
               writeln(' PROBLEM NUMBER ', problemno: 4, '  ', starting:
                  2 oct, '  ', optimising: 2 oct, ' NO OF ITERATION(S) '
                  , iteration: 3, ' FINAL TOURLENGTH ', tourlength: 7,
                  ' FINAL TIME ', finaltime[starting, optimising]: 7);
               tourlists(infull);
               writeln;
            END;
         garbagecollection(firsthead);
         writeln;
         writeln;
      END;
   writetofiles;
END  {salesv02} .
```