Interactive Approaches to the Solution of a Class of Combinatorial Problems

A Thesis

submitted to the

UNIVERSITY OF NEWCASTLE UPON TYNE

for the degree

of

DOCTOR OF PHILOSOPHY

L. WALLER

September 1971

Acknowledgements

I am indebted to the Science Research Council for awarding me a Research Studentship to enable me to study at the University of Newcastle upon Tyne from October 1966 to September 1968 and also to the Computing Laboratory of the University in whose employment the latter stages of the research was conducted.

My thanks are due to my supervisor, Professor E.S. Page for helpful remarks and ideas and to the staff of the laboratory for assistance at all times.

Preface

This thesis considers the usefulness of interaction between a human and a powerful computer in attempting to solve a class of discrete optimization problems. Some typical problems are described in chapters 1 and 2 and the effectiveness of their exact solution by existing methods is assessed. Chapter 3 presents some heuristic techniques which produce good approximate solutions and the value of such methods is discussed.

An alternative approach, that of providing a mechanism for man-machine interaction is proposed in chapter 4. A system for providing easy access to a range of algorithmic and heuristic techniques is described. The system, named IMPACT, was implemented by the author and its many features include the interruption, interrogation, adjustment and resumption of a process or algorithm. Some novel interactive tree-manipulation techniques and their usage are introduced in chapter 5. This chapter also describes extensions to certain other heuristics in order to improve their power when used interactively.

Throughout the thesis a job-shop scheduling problem serves as a useful vehicle for illustrating ideas. This problem was investigated extensively and chapter 6 is devoted to the topic. The idea of a critical path of jobs through machines is introduced together with the slack time of a job upon a machine under a particular schedule.

Branch-and-bound approaches to the problem have been proposed in

the past. The performance of such an approach has been substantially improved, as is shown by new results. The improvement stems from two sources both of which were discovered interactively; i) a different branching procedure designed to exploit features of the job-shop scheduling problem, and ii) more realistic lower bounds than those originally proposed.

The final chapter discusses the generality of the approach and illustrates the extendability of IMPACT. Other discrete optimization problems are discussed briefly and a branch-and-bound formulation to one of them, an assignment problem, is presented. An interactive approach by other authors to the travelling salesman problem is reviewed and features similar to those experienced in the job-shop scheduling investigation are remarked upon. To conclude, the advantages to be gained from an interactive approach are discussed.

Contents.

Index to figures in the text.

Interactive Approaches to the Solution of a Class of Combinatorial Problems

Chapter 1: Introduction

Certain types of discrete optimization problems have the characteristics that a solution may be represented by a vector ν which minimizes some cost function C (ν). In effect, the cost function, C, is a 'black box' which takes as input a vector ν and outputs a cost associated with the vector. The difficulties presented by such problems lie in the fact that in general there are a great number of possible input vectors and the determination of one which produces the minimum cost is not trivial. The behaviour of a black box function might be quite complex and an orthodox approach to the problem is to utilise knowledge of the black box's behaviour to attempt to arrive at an analytical solution to the problem. Some particular black box functions are described and some results and limitations of the conventional methods of attack are given.

In later chapters a different approach is described. This approach is intended to utilise the powerful interactive facilities offered by modern multi-programming computers. Man-machine interaction, providing the ability to direct computational effort into areas which one's judgement and/or intuition suggest are worth exploring, would appear to be a powerful tool with which to attack the problem. Such an approach permits one to be ignorant of the mechanism of the

black box function. It is expected, however, that the behaviour of the black box will be uniform in a respect that 'similar' input vectors will exhibit similar cost values. (This concept of similarity between vectors is difficult to formalise but it is thought that the user of this approach may develop a 'feeling' for both the behaviour of the black box and the measure of likeness between input vectors). Later chapters describe the results of such an interactive approach and in particular how such an approach aided the development of a more powerful method of solution for one particular cost function. The particular cost functions described here may be classified as 'permutation problems' in the sense that the vector input to the cost function can be represented as a permutation of (1, 2, . . .,n). Much of the research involved and software produced is thus oriented towards permutation problems. However, other problems could be approached in a similar fashion and many of the techniques developed adopted.

1.1 The Job-Shop Scheduling Problem.

The job-shop scheduling problem exists in many forms, some of which possess exotic constraints and cost functions (see Sissons (1), and Waller (2)). The problems described here, whilst simple in its structure and cost function still presents great computational difficulties.

The problem is to find the best (in a sense to be described) order of processing of n jobs J_1, J_2, ..., J_n on m machines

$M_1, M_2, \ldots, M_m$ subject to

1) A job must pass through machine M_{j-1} before proceeding onto machine M_j.

2) A machine may process only one job at a time and the processing of a job upon a machine may not be interrupted once it has started.

In the problem there are n! different ways of putting the jobs onto the first machine, n! different ways of putting them onto the second machine, and so on. Thus the total number of possible sequences of placing jobs onto machines is $(n!)^m$. Each operation by a machine upon a job could be mapped bijectively onto the integers 1, 2, . . . , nm. Any permutation of (1, 2, . . . , nm) then represents a possible schedule of jobs through machines, although not all permutations will be feasible; (constraint 1, above, may be violated).

The processing time for job i upon machine j is represented by d(i, j)and thus the problem may be specified by the array $D = \{ d(i, j) \}$. A number of cost functions are available for the problems and two of the more common ones are i) minimise the total duration from the start of processing of the first job upon the first machine until the finishing of processing of the last job upon the last machine, ii) minimise the mean of the completion times of the jobs. The former cost function, often called minimising the makespan or schedule length will be considered here.

The two-machine job scheduling problem of minimising makespan has been solved analytically by Johnson (14) and an easily-applied procedure for obtaining an optimum given; job i precedes job j in an optimum solution if $\min \{d(i,1), d(j,2)\} < \min \{d(j,1), d(i,2)\}$. The approach carries over to three - machines only in special cases. A much more complicated rule was proposed by Dudek and Tenton (24) for the case where $m \geq 2$ but a counterexample by Karush (25) shows that an optimal solution is not guaranteed. Johnson's rule implies that the order of jobs through machines is the same for each machine. Roy (3) has shown that no advantage is gained by having different sequences of jobs for the first two machines, or different sequences for the last two machines. The total number of different sequences to be considered is thus $(n!)^{m-2}$. In order to convert the problem to a form in which all permutations considered are feasible, the following 'no passing' restriction is added.

3) The order of jobs through machines is the same for each machine.

Condition 3 means that for the case $m > 3$ an optimum under the three conditions above will not necessarily mean that no improvement could have been made with the no-passing restriction removed. Occasionally in later chapters the condition will be temporarily dispensed with but it will be made clear when this occurs and the reason for the condition's relaxation will be to allow the use of certain heuristic methods.

It can be seen that the jobs can be numbered (1, 2, . . .,n) and that any permutation of (1, 2, . . . n) represents a feasible sequence of jobs through machines. The problem has thus been transformed so that the input vector to the cost function is a permutation. Furthermore the cost of any permutation can be easily calculated.

1.11. The calculation of the makespan (or cost) of a permutation.

Consider the following example of 5 jobs to be processed upon 3 machines where the cost matrix D is as shown.

Jobs	Machines			
	I	II	III	
1	1	12	19	
2	13	12	18	= D
3	3	31	20	
4	27	24	12	
5	51	12	18	

Table 1: A problem of scheduling 5 jobs upon 3 machines.

Placing the jobs on the machines in the order 3, 2, 4, 5, 1 (inputting the permutation (3, 2, 4, 5, 1) to the cost function)would result in a cost (duration) of 143 being returned. Figure 1 depicts the sequence of events, the first few of them occurring as follows:

At time zero job number 3 would begin processing upon machine I and would leave this machine at time 3. Machine II is unoccupied at this time and so job number 3 can pass

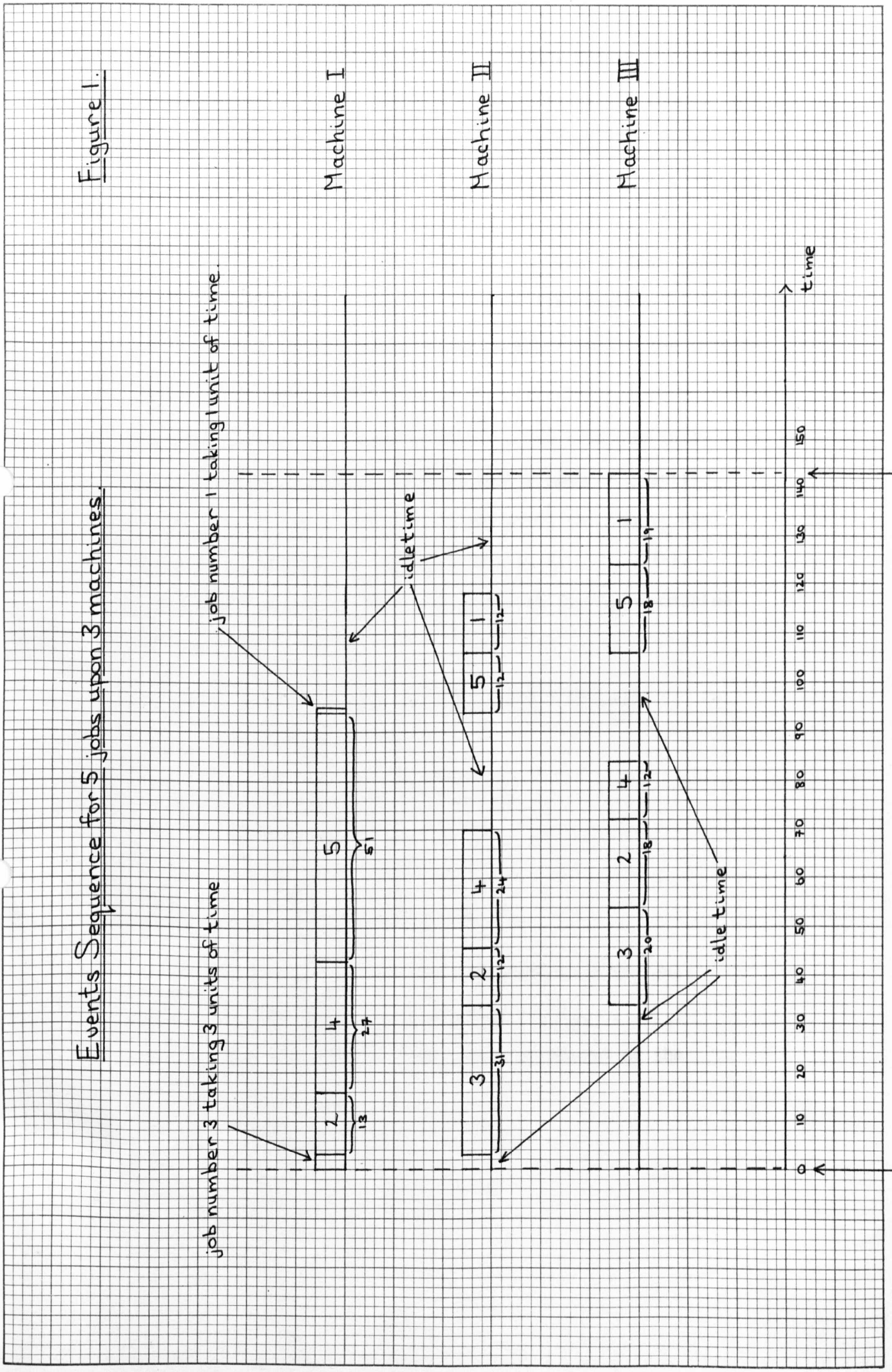


Figure 1.
Events Sequence for 5 jobs upon 3 machines.
job number 1 taking 1 unit of time
job number 3 taking 3 units of time
Machine I
Machine II
Machine III
idle time
idle time
time
START
FINISH
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
13
27
51
31
24
12
12
18
20
18
12
19

onto this machine and will leave it after a time 3 + 31 = 34. It then proceeds onto machine III (which is as yet unoccupied) and eventually leave this machine after a time 54. Meanwhile, once the first job has vacated the first machine, this machine is free to accept the next job, i.e. job number 2. This job occupies machine I until time 3 + 13 = 16 and is then ready to pass onto machine II. However, at this time machine II is still occupied by job number 3 and so job 2 awaits the completion of job 3 upon machine 2.

From the above it is apparent that i_k, the k th job in a sequence, cannot begin processing upon a machine j until machine j is ready to process it and until the job i_k has finished its processing upon machines 1, 2, . . ., j-1. Denoting 'the earliest finishing time' of job i_k on machine j as $f(i_k, j)$ we have

$$f(i_k,j) = \max \left\{ f(i_k, j-1),\ f(i_{k-1},j) \right\} + d(i_k,j)$$

$$\forall\ j>1 \text{ and } k>1$$

$$f(i_1,j) = \sum_{s=1}^{j} d(i_1,s) \qquad \forall\ j \geq 1$$

and

$$f(i_k, 1) = \sum_{s=1}^{k} d(i_s,1) \qquad \forall k \geq 1$$

This gives an algorithm for the calculation of the cost function for any complete permutation $(i_1,i_2,\ldots i_n)$ and the total cost (for n jobs upon m machines) of the permutation is $f(i_n,m)$.

In the previous discussion it was assumed that jobs were placed upon machines as early as possible; i.e. jobs are not deliberately held back as this could not in any way decrease the cost function, but could serve to increase it. This assumption holds throughout the discussion of the job-shop problem.

The calculation of the earliest finishing times can be performed simply with pencil and paper and the following worked example is instructive. For the problem stated earlier the earliest finishing times for the permutation (3, 2, 4, 5, 1) are as in figure 2.

	I	II	III			
3	3	34	54	f(3,1)	←f(3,2)	←f(3,3)
2	16	46	72	↑f(2,1)	↑f(2,2)	↑f(2,3)
4	43	70	84	↑f(4,1)	↑f(4,2)	↑f(4,3)
5	94	106	124	↑f(5,1)	←f(5,2)	←f(5,3)
1	95	118	143	↑f(1,1)	↑f(1,2)	↑f(1,3)

Figure 2: The Earliest Finishing Times of jobs upon Machines

The entries in the table correspond to the 'f-values' or 'earliest finishing times' on the right. The arrows associated with an f-value point to the earliest finishing time from which the f-value was derived.

A point to be observed is that the cost of any schedule is the sum of (n+m-1) job-machine times. In the example the cost of 143 is the sum of 7 job-machine times, i.e.

3 + 13 + 27 + 51 + 12 + 18 + 19.

1.2. Terminology

In order to fit in with the terminology widely used in connection with job-shop scheduling the terms 'part permutation', 'vertex' and 'node', used in later chapters are synonymous. A 'part permutation', or more strictly a 'part permutation of 1 to n' is an ordered sequence $(i, i_2, \ldots, i_r)$ of integers in the closed interval $[i, n]$, such that $1 \leq r \leq n$ and $i_s \neq i_t$ for $s \neq t$ and $1 \leq s \leq r$, $1 \leq t \leq r$. A part permutation can thus be a complete permutation of $1, \ldots, n$.

An attempt will be made to refer to $(i_1, i_2, \ldots, i_r)$ as a part permutation whilst the terms 'node' or 'vertex' will be applied to points in a graph or tree . (Such points will be defined by the part permutation associated with them).

The following terminology will be employed in connection with graphs. A graph consists of a set of nodes and a set of unilateral associations specified between parts of nodes. If node i is associated with node j, the association is called a branch from initial node i to terminal node j. A path is a sequence of branches such that the terminal node of each branch coincides with the initial node of the succeeding branch. Node j is reachable from node i if there is a path from node i to node j. The number of branches in a path is the length of the path. A circuit is a path in which the initial node coincides with the terminal node.

A tree is a graph which contains no circuits and has at most one branch entering each node. A root of a tree is a node which has no branches entering it, and a leaf is a node which has no branches leaving it. A root is said to lie at level zero of the tree, and a node which lies at the end of a path of length j from a root is on the jth level. The set of nodes which lie at the end of a path of length one from node x comprises the filial set of node x, and x is the parent node of that set. The set of nodes reachable from x is said to be governed by x and comprises the nodes of the subtree rooted at x.

1.3 The Travelling Salesman Problem

Another discrete optimisation problem of the category described earlier is the travelling salesman problem. In its simplest form the objective is to find an optimal 'tour' through n locations or cities which starts at one location, visits each of the remaining locations once and once only, and returns to the starting location. An optimal tour is a tour that minimises the total distance travelled.

The cost matrix $D = \{ d(i,j) \}$ specifies the distances involved; d(i,j) represents the cost of travelling from city i to city j. The n cities may be thus represented as a graph in which a branch from node i to node j has an associated distance d(i,j). If the distance d(i,j) associated with a branch is ∞ then there is no route from city i to city j without passing through at least one other city. It is thus possible for the problem to be

degenerate and have infinite cost if there exist two cities (or nodes) i and j such that node i is not reachable from node j. (Any problem considered later will be assumed to be non-degenerate).

Table 2 thus corresponds to the graph in figure 3.

City	1	2	3	4
1	∞	7	4	2
2	9	∞	5	8
3	4	3	∞	1
4	6	∞	8	∞

Table 2. The Distances between Cities

$D = \{ d(i,j) \}$

Without loss of generality the cities in the problem may be numbered 1, 2, . . ., n and a solution to the problem is then a permutation $P_n = (i_1, i_2, \ldots, i_n)$ of 1, 2, . . ., n that minimises the quantity $d(i_1,i_2) + d(i_2,i_3) + \ldots + d(i_{n-1},i_n)$

$$+ d(i_n,i_1) = \sum_{i=1}^{n-1} d(i_j,i_{j+1}) + d(i_n,i_1)$$

(The example subroutine given in Appendix 3 calculates this cost function).

The first item of an input permutation may be regarded as fixed and thus the number of possible solutions to the problem is $(n - 1)!$

Variations of the problem exist. In the case of a symmetrical cost matrix, $d(i,j) = d(j,i)$ and consequently the number of possible solutions if $\frac{1}{2}(n-1)!$ Sometimes the return to the starting location is not required and thus in the non-symmetric case there are $n!$ possible solutions whilst if symmetry exists $\frac{1}{2}n!$ permutations must be considered.

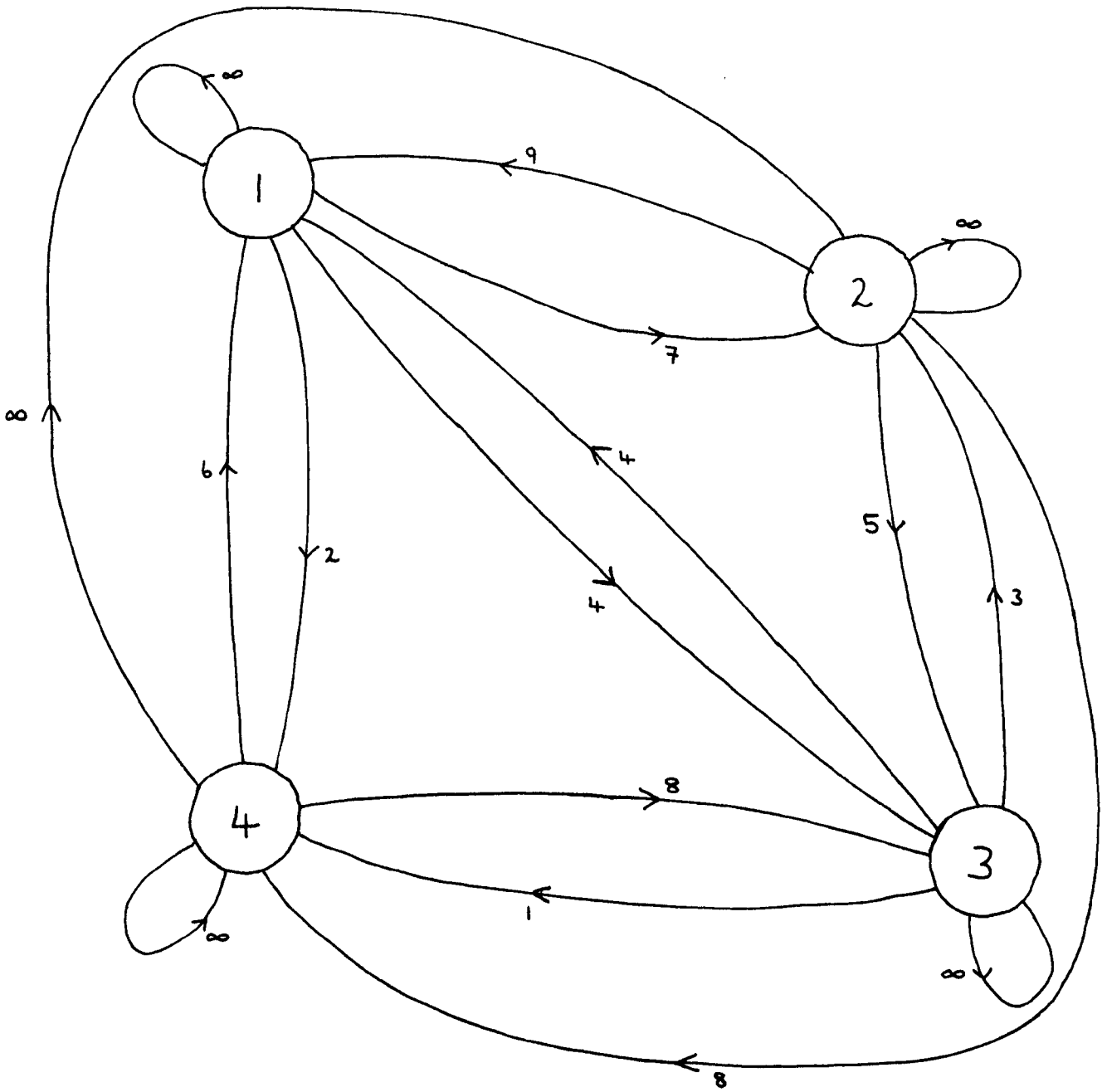


Figure 3 : The graph for a travelling salesman problem.

Certain scheduling problems may be expressed as travelling salesman problems. As an example from production scheduling, suppose that there is a production cycle of some time, during which an assembly line must produce each of n different models. If the cost of switching from model i to model j is d_{ij} and it is desired to minimise total setup cost, then the problem is a travelling salesman problem. Normally d_{ij} would not be equal to d_{ji} and thus the problem would not be symmetric.

1.4 A problem of assigning facilities to locations

Another discrete optimization problem that has to date eluded easy optimal solution can be stated as follows. A number, n, of facilities are to be assigned to n locations such that one and only one facility is assigned to each location. Associated with each pair of locations is a cost, or index of cost, of having to transfer a unit of material between the location pair. This is thus a 'distance' between the locations and whilst it might be a linear distance it need not be necessarily so. Similarly with each pair of facilities there is an index of the 'traffic intensity' between the facilities. The traffic intensity is a measure of some dependence between two facilities and could be the rate at which materials are transferred between the two facilities. The cost of assigning a pair of facilities to a pair of locations is the product of the location distance and the facility traffic intensity. The cost of a complete assignment will be the sum of all such products for each location-facility pairs in the assignment.

There are several practical examples of the problem. One such case is the layout of a plant where a number of machines, pieces of equipment, or departments must be assigned to a set of locations. The locations will be at fixed distances from each other and there will be a rate of flow of materials (possible people or parts) between each pair of facilities. In the field of ergonomics the layout of a control panel reduces to the above problem if the frequency of access to each control and from each control to every other is known or can be predicted.

The problem can be represented by a distance matrix $D = (d_{ij})$. where $d_{ii} = 0$ and $d_{ij} = d_{ji}$ for $1 \leq i, j \leq n$, and a flow matrix $F = \{f_{ij}\}$ where $f_{ii} = 0$ and $f_{ij} = f_{ji}$ for $1 \leq i, j \leq n$. The cost of assigning facilities k and ℓ to locations i and j is then $f_{k\ell}\, d_{ij}$.

A permutation $(\ell_1, \ell_2, \ldots, \ell_n)$ can be interpreted as assigning facility i to location ℓ_i and thus the cost of the above permutation is

$$\sum_{s=1}^{n} \sum_{t=s+1}^{n} f_{st}\, d_{\ell_s \ell_t}$$

As an example with the distance and flow matrices in the table below

Location	1	2	3	4	5
1	0	1	1	2	3
2	1	0	2	1	2
3	1	2	0	1	2
4	2	1	1	0	1
5	3	2	2	1	0

Distance Matrix D

Facilities	1	2	3	4	5
1	0	5	2	4	1
2	5	0	3	0	2
3	2	3	0	0	0
4	4	0	0	0	5
5	1	2	0	5	0

Flow Matrix F

Table 3: Data for an assignment problem

The cost of the permutation (3,5,2,1,4) is thus

$$f_{12}d_{35} + f_{13}d_{32} + f_{14}d_{31} + f_{15}d_{34}$$
$$+ f_{23}d_{52} + f_{24}d_{51} + f_{25}d_{54}$$
$$+ f_{34}d_{21} + f_{35}d_{24}$$
$$+ f_{45}d_{14}$$

$$= 5 \times 2 + 2 \times 2 + 4 \times 1 + 1 \times 1 + 3 \times 2 + 0 \times 3 + 2 \times 1$$
$$+ 0 \times 1 + 0 \times 1 + 5 \times 2 = 37.$$

Chapter 2 : Approaches to the Problem.

2.0 Complete Enumeration.

In the problem of determining which permutation of (1, 2, . . .,n) yields the minimum cost for some cost function the number of possible solution vectors, or permutations, is finite (= n!) and thus it is theoretically possible to generate each permutation, determine its associated cost and choose the permutation with the lowest cost. Such a method is, however, not practically feasible for anything but small n owing to the behaviour of the factorial. If n = 15 examination of the 15! possibilities at the rate of one per second would take approximately 40,000 years! An important feature of the problem is illustrated here; the effort involved in calculating the cost of a particular permutation is not prohibitive - the number of different permutations possible is the stumbling block.

A more efficient approach than explicit exhaustive enumeration is to attempt to exclude large groups of permutations by some criterion peculiar to the cost function, i.e. a 'sieve' is required. If by use of such a sieve a part permutation $(i_1, i_2, \ldots)$ may be seen to be inherently suboptimal because of its starting elements (i_1, i_2) then (n - 2)! permutations may be ruled out in one step, without having to examine each of them individually. Thus a powerful sieve could significantly reduce the number of permutations to be examined. This method of searching, implicit exhaustive enumeration, compares favourably with the explicit search providing that the sieves are easily computable

and stringent in the number of possibilities they eliminate. Several of the advocates of the implicit search technique make a comparison with explicit enumeration but the test to be applied to implicit search techniques is not whether they perform better than poor methods but whether they are computationally practical.

2.1 Branch-and-Bound Methods

Lomnicki (4) proposed attacking the job-shop scheduling problem by using a 'branch-and-bound' technique. Branch-and-bound is a form of implicit exhaustive enumeration and its essentials are a systematic method of searching (the 'branching' procedure) so that every possible solution vector is considered, and a criterion for obtaining a lower bound for any partial vector. At any stage in the search a sieve (a function of the bound calculated) is applied to the next part permutation and if the part permutation is deemed worthy of further investigation it is stored for later examination, otherwise it (and consequently all permutations beginning with this part permutation) is rejected. The process is repeated by exploring one of the part permutations stored earlier and continues until no more part permutations remain for examination. Variations of branch-and-bound methods can be arrived at by employing different branching methods and by altering the sieve (or function of the lower bounds).

2.11 Branching Methods.

The set of all permutations of n objects, numbered 1, 2, . . ., n can be conveniently represented as the leaves of a tree. (Such a tree for the case when n = 4 is shown in figure 4). The root of the tree corresponds to the null part permutation and

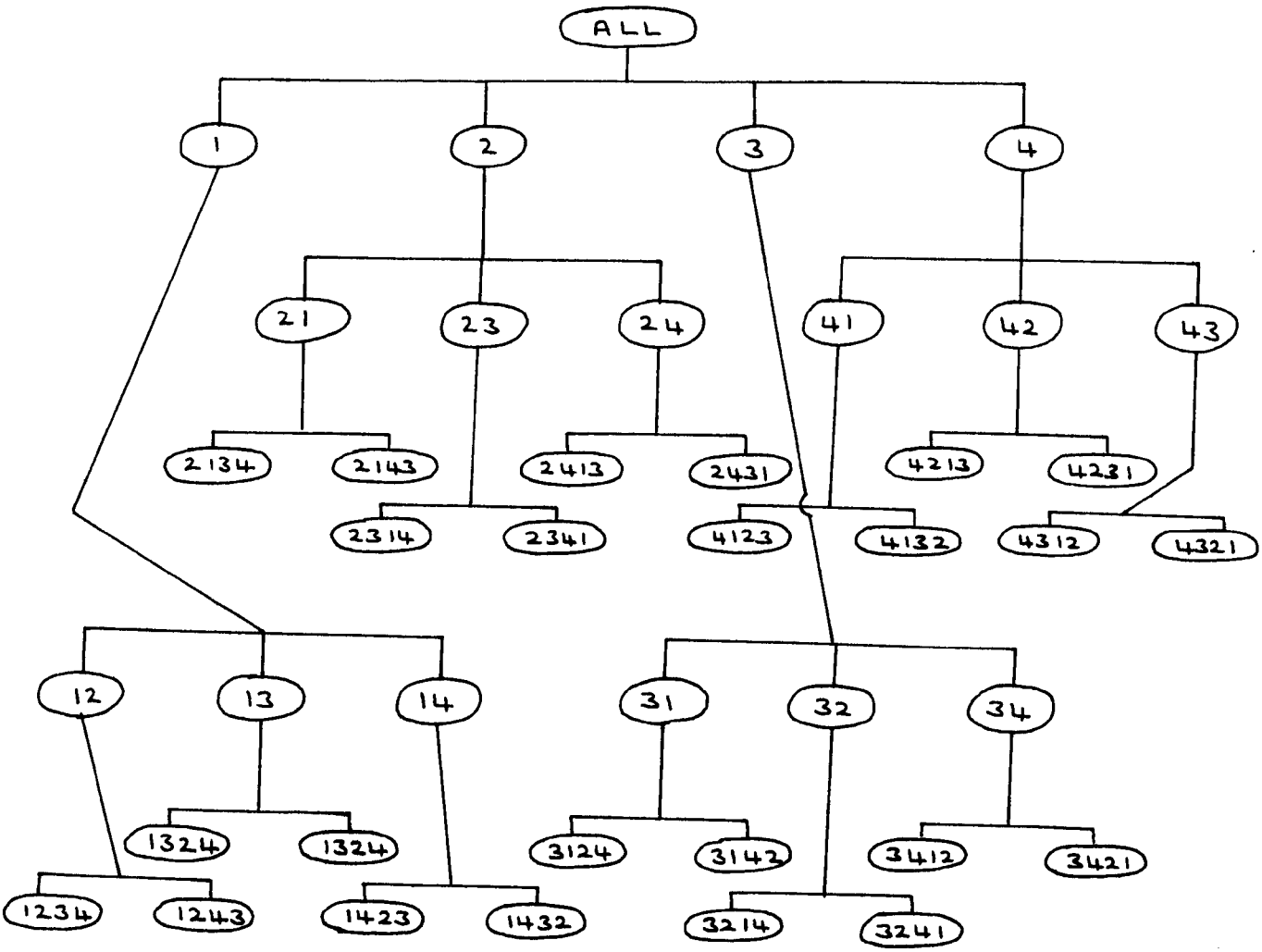


Figure 4: The permutations of (1,2,3,4) arranged as leaves of a tree.

branches emanating from the root lead to n nodes, each of which corresponds to the placing of a different one of the elements 1, 2, . . . , n in the first position of the permutation. The filial set of any of these nodes consists of n-1 nodes which correspond to allocations for the first two positions in the permutation. The node corresponding to the part permutation (i_1) where $i_1 \varepsilon (1, 2, \ldots, n) = I_n$ thus has a filial set consisting of n-1 different part permutations of the form (i_1,k) where $k \varepsilon \{I_n \sim (i_1)\}$. At a level below any of these n(n-1) nodes there are another n-2 nodes and in general at a level j of the tree there are to be found those nodes corresponding to part permutations with the first j positions allocated. If $P_j = (i_1,i_2, \ldots, i_j)$ is a part permutation the node corresponding to P_j will have n - j direct descendents of the form $P_{j+1} = (i_1, i_2, \ldots, i_j, k)$ where $k \varepsilon \{I_n \sim P_j\}$. It follows that T_{j+1}, the total number of nodes at level j + 1, = (n - j) x T_j for $1 < j \leq n$. Since $T_1 = n$, $T_2 = n(n-1)$ it is clear that $T_n = n!$ and that each of these nodes corresponds to a different permutation of $(1, 2, \ldots, n)$.

2.111 Branching from the Lowest Bound

An attractive method for the systematic investigation of the tree is that of examining at any stage the node that possesses the lowest lower bound. This can be termed 'branching from the lowest bound' and it is apparent that, in terms of the number of nodes examined, no other branching method can better this, since any node examined under this method must be examined at some time to ensure that an optimum solution has been found.

Whenever a node has been examined and appropriate lower bounds to the associated part permutation calculated, the bounds must be sorted and placed with any lower bounds already stored from the investigation of other part permutations. The smallest from the whole set is then investigated further. This method is advocated by Ignall and Schrage (5) for the job-shop scheduling problem and has the drawback of potentially requiring a huge amount of storage space for the part permutations upon which investigation has been suspended. The worst possible case would require the storage of n! part permutations upon which investigation has been suspended.

2.112 Backtracking.

A branching method requiring a much smaller amount of storage space is that of backtracking, or in the terminology of Lawler and Wood (6), 'Branching from the Newest Active Bounding problem'. At any stage of the search if the node P_j, $j < n$, is being investigated the next node to be examined will be one of the filial set of P_j, i.e. some P_{j+1}. The most logical choice of P_{j+1} will be the one that has the smallest lower bound. If the lower bound is sufficiently low for P_{j+1} to warrant investigation (i.e. if the bound is less than the lowest cost of all permutations as yet examined) it will be examined further. Whenever a node P_j is deemed unworthy of further examination the next node to be examined is found by climbing up the tree one level at a time until a suitable node P_l ($l < j$) is encountered. (This mechanism produces a lexicographical ordering amongst part permutations in that $P_j = (i_1, i_2, \ldots, i_j)$ is said to be

nearer to $P'_j = (i'_1, i'_2, \ldots, i'_j)$ than to $P''_j = (i''_1, i''_2, \ldots, i''_j)$ if there exists some k with $1 < k \leq j$ and $i'_k = i_k$, $i''_k \neq i_k$ such that $i_s = i'_s = i''_s$ for all $s < k$).

Whenever a complete permutation is encountered the actual cost of the permutation is calculated and if the cost is less than that associated with the previous best permutation a record is made of it. The search terminates when no node exists that is worthy of further investigation.

It can be seen that the tree will be investigated in a backtrack fashion, and such a search lends itself to an easy implementation by the use of a 'push-down' stack. At the first level of the tree n nodes are generated and ordered according to their lower bounds. The one with the lowest lower bound is retained and the remainder placed upon the stack so that in the event of any of them being required later the smallest will be retrieved first. Similarly at the next level

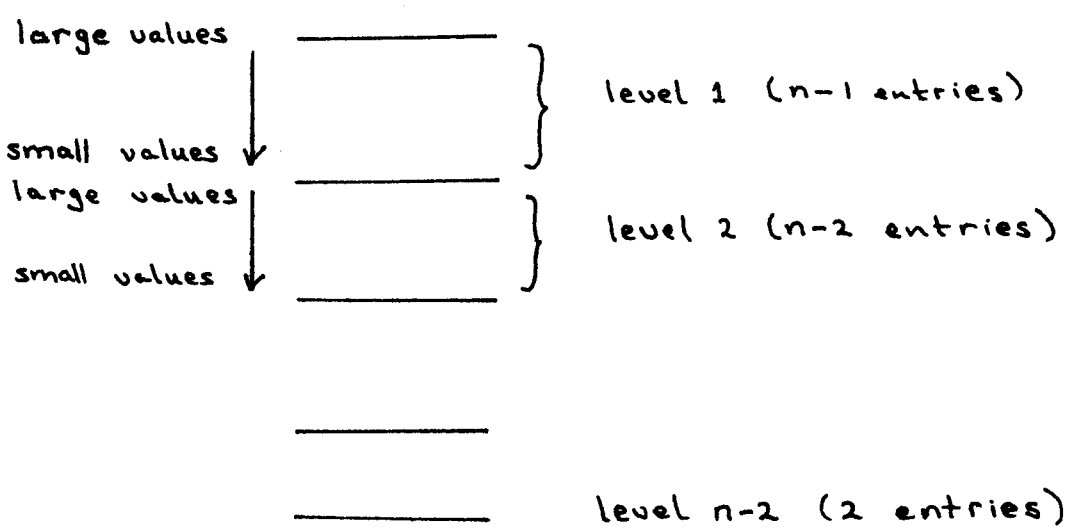


Figure 5: Arrangement of a 'push-down' stack for backtracking.

of the tree n-2 lower bounds will be placed upon the stack, again ordered amongst themselves. Figure 5 depicts the arrangement of the lower bounds on the stack. When a complete vector is reached bounds are removed from the bottom of the stack and either rejected (the lower bound is too high) or explored further. When a node is explored further any new nodes generated can be added to the stack as before. It can thus be seen that only as many nodes will be added as have been removed and hence no more storage is required. (In practice only still potentially fruitful nodes are added to the stack and thus at least as many nodes are removed as are replaced). The maximum number of storage locations required for the stack is thus $(n - 1) + (n - 2) + \ldots + 2 = \frac{n(n-1)}{2} - 1$.

A flowchart illustrating the backtrack method is given in figure 6. The part of the flowchart marked by the dotted lines is not essential to the search but illustrates certain controls which are described in later chapters.

To summarize, the essential difference between the two branching methods described is that the choice of which part permutation to next examine is made locally within the tree when backtracking, whilst a global decision (from the whole tree) is made under the 'branch from the lowest bound' strategy.

2.2 Lower Bounds for the job-shop scheduling problem.

The branch-and-bound approach is general in that in order to tackle a different permutation type problem only a new bounding function and cost function needs to be provided. The calculation of the lower bounds utilises knowledge about the behaviour of the cost function; usually the lower bound is derived by considering the

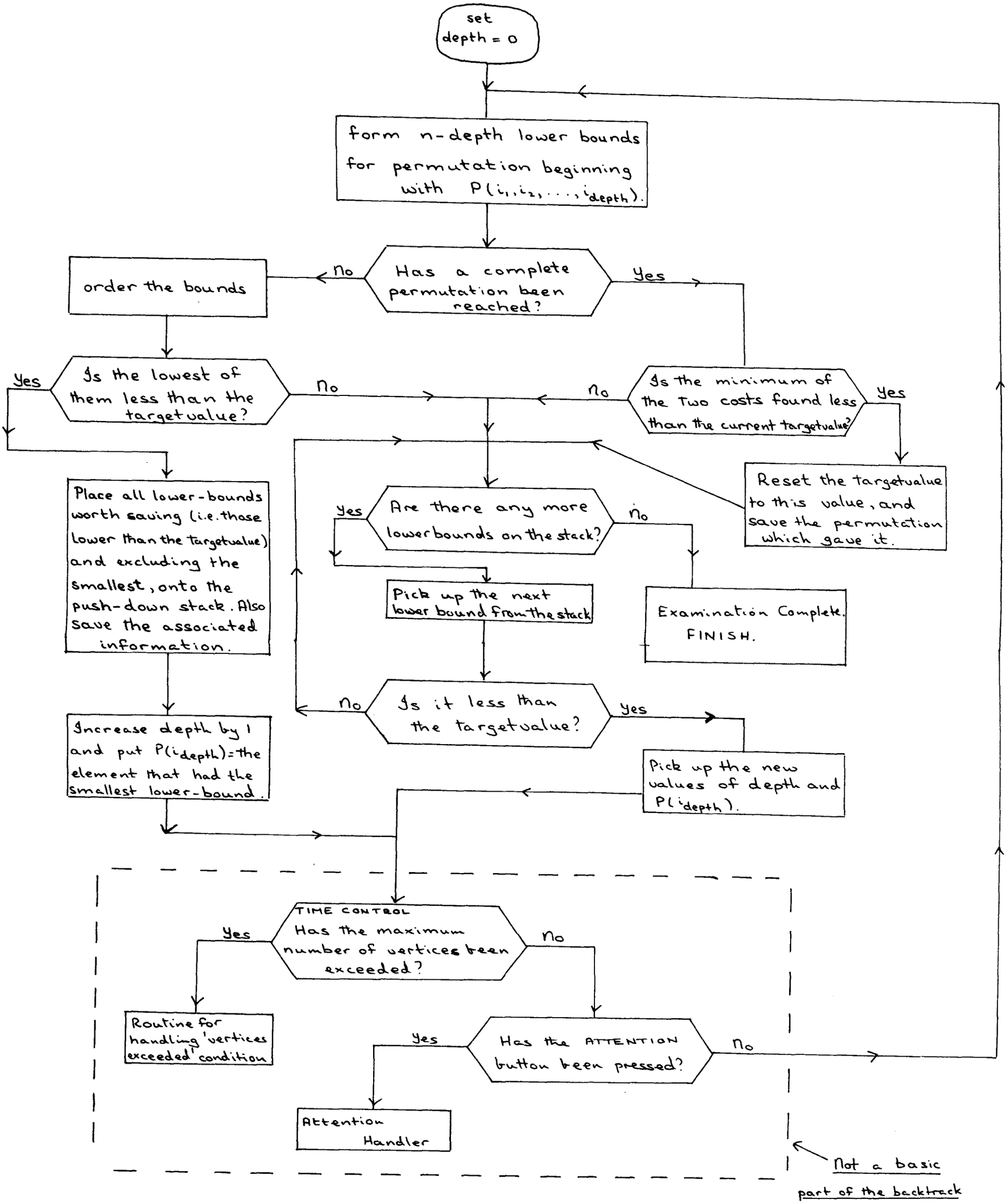


Figure 6: A Flowchart For Backtracking.

actual contribution to the cost of the permutation by the elements of the part permutation, and the minimum contribution that could occur by the placing of the remaining (unallocated) elements of the permutation. A lower bound for a part permutation for the job-shop scheduling problem is derived as follows.

Suppose that the n jobs which are to be processed upon the m machines have job-machine times given by $D = \{ d(i,k) \}$, where $d(i,k)$ is the time taken for job i to be processed upon machine k. For a schedule beginning with $P_j = (i_1, i_2, \ldots, i_j)$ m lower bounds can be determined in the following manner:

m - 1 of the lower bounds are obtained by the time for the first j jobs to finish upon the first k machines plus the time for the remaining n - j jobs to finish upon the kth machine (assuming no idle time), plus the time for the last job to be processed upon the remaining m - k machines. As no decision has yet been made as to which job will be processed last, that job, out of those still unassigned, that occupies the last m - k machines for the least time is chosen.

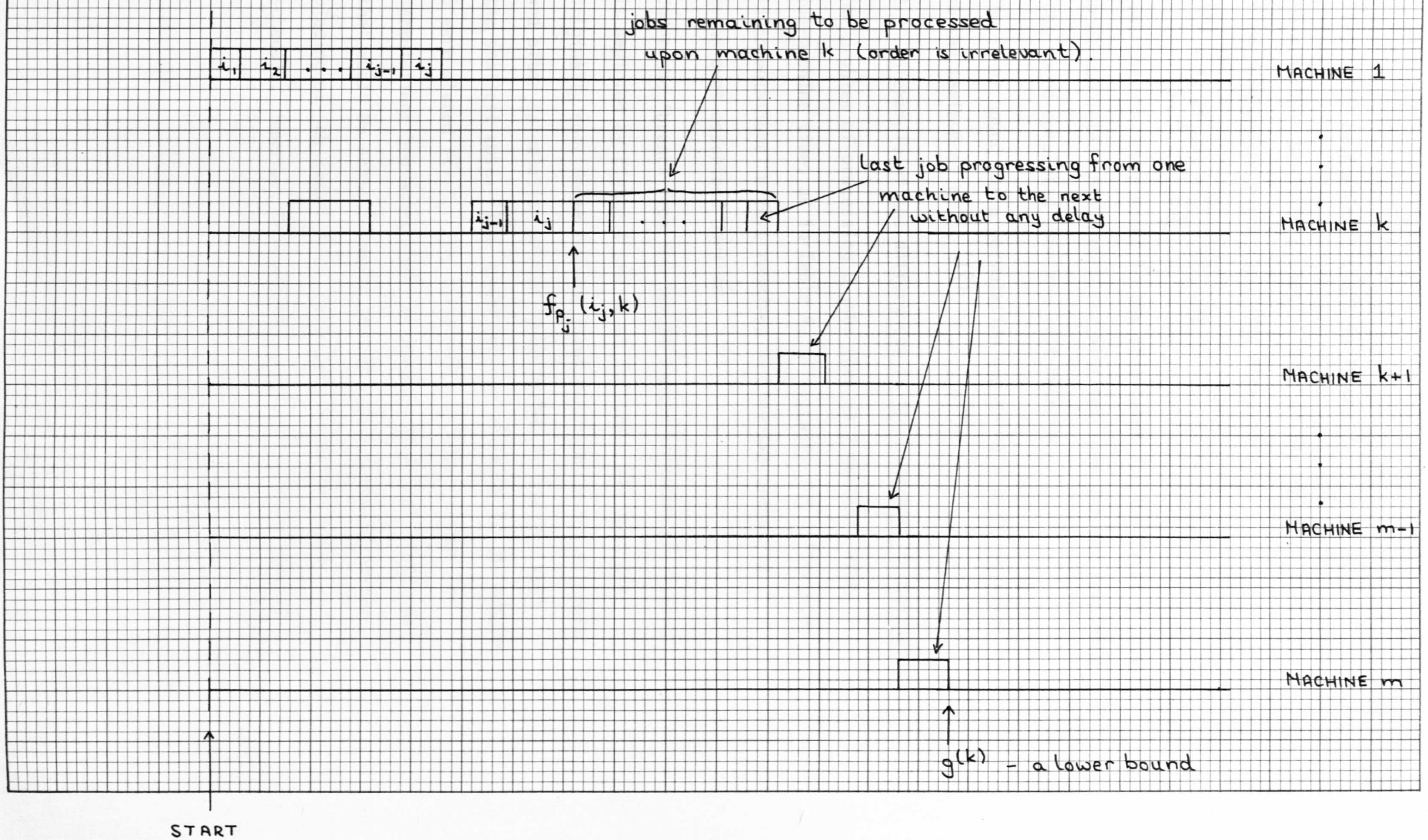
Thus $g^{(k)}$, the k th lower bound, is given by

$$g^{(k)} = f_{P_j}(i_j, k) + \sum_{s \in \overline{P}_j} d(i_s, k) + \min_{s \in \overline{P}_j} \left\{ \sum_{t=k+1}^{m} d(s,t) \right\}$$

where $k = 1, 2, \ldots, m-1$. $\overline{P}_j = \{ I_n \sim P_j \}$

and $f_{P_j}(i_j,k)$ is the earliest finishing time of job i_j upon machine k under the schedule defined by the part permutation P_j. (This terminology agrees with 1.11 except that the schedule P_j has been appended to f). The general bound $g^{(k)}$ is

Figure 7: An Illustration of lower bounds for the job-shop scheduling problem.



illustrated in the Gantt chart in figure 7.

The mth lower bound is a special case of the above in that the third term on the right hand side disappears.

$$g^{(m)} = f_{P_j}(i_j,m) + \sum_{s\varepsilon\overline{P_j}} d(i_s,m)$$

i.e. the mth lower bound is the time taken for the first j jobs to be completed upon m machines plus the time for the remaining jobs to be completed upon the last machine.

The lower bound for a permutation beginning with P_j is taken to be the maximum of $g^{(1)}, g^{(2)}, \ldots, g^{(m)}$.

2.3 An example of the differences in the branching techniques

The problem of processing 6 jobs through 3 machines with job-machine times as in Table 4, illustrates the differences in computational effort required for the two branching techniques described earlier.

Under the 'branch from the lowest bound' approach the search is depicted by figure 8 and is as follows. All jobs are originally considered for the first position in the schedule and after calculation of the lower bounds it appears that the smallest cost, 1216, could only be achieved if job number 6 were placed first. The calculation of lower bounds for permutations beginning with (6) reveals that the lowest cost that could be achieved would be 1381, corresponding to the placing of job number 1 in the second position of the schedule. However, from reexamination of the remainder of

	Machines		
Jobs	I	II	III
1	153	141	6
2	242	69	197
3	168	228	313
4	308	211	173
5	42	221	311
6	87	129	0
Totals	1000	999	1000

Table 4: An array of job-machine times

the tree it is apparent that by starting the permutation with (5) a cost of 1263 might be achieved. The calculation of lower bounds for permutations beginning with (5,1), (5,2) (5,3), (5,4) and (5,6) shows that the smallest of these is 1263, which is at least as small as any bound still active in the tree. There is equality at this level and so an arbitrary choice of node, say (5,2), is made. Calculation of the appropriate lower bounds gives those associated with nodes (5,2,1), (5,2,3), (5,2,4), (5,2,6) as marked on the diagram. The next node to explore can be any of (5,1), (5,3), (5,4), (5,6), (5,2,1), (5,2,3) and (5,2,6) and since it appears to be good sense to make the choice from those part permutations that have most elements; i.e. from (5,2,1), (5,2,3) and (5,2,6),suppose node (5,2,3) is selected. Continuing along these lines (5,2,3,4) is reached and its two descendents (5,2,3,4,1,6) and (5,2,3,4,6,1) are complete permutations and hence their costs can be determined by the algorithm in 1.11. The smallest of these two costs, 1263, is (less than or) equal to the lowest lower bound associated with any unexplored node in the tree. The search is thus completed after 25 nodes have been examined and the maximum

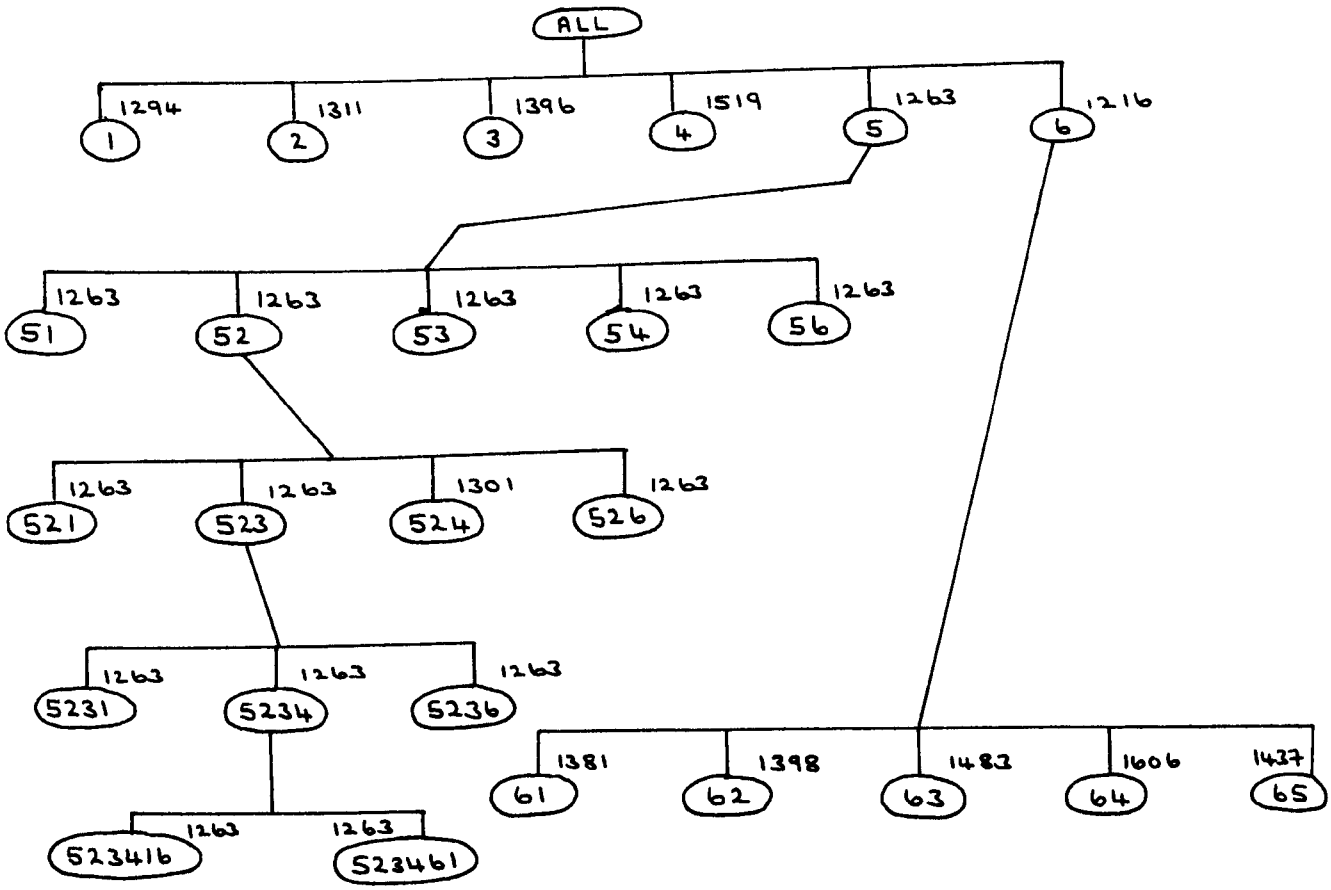


Figure 8: The tree examined under the method of 'Branching from the lowest bound'.

number of nodes which had to be stored during the search was 18.

In the backtracking approach the tree in figure 9 is examined and the essential difference is that the exploration of the node (6,1) is not suspended. The nodes (6,1,2), (6,1,2,5), (6,1,2,5,3,4) and (6,1,2,5,4,3) are examined and a cost of 1569 determined. Backtracking up the tree now takes place and any node with a lower bound greater than the target value of 1569 is rejected. The nodes (6,1,2,3), (6,1,2,4), (6,1,5), (6,1,3) and (6,1,4) are thus rejected (in that order) and the sub-tree beginning (6,2) is explored. Nodes (6,2,5), (6,2,5,3), (6,2,5,3,1,4) and (6,2,5,3,4,1) are investigated and the new target value of 1422 discovered. (The choice of the node (6,2,5,3) rather than either of (6,2,5,1) and (6,2,5,4) occurs in this example only because of the ordering imposed by a sorting routine used in the program from which the example was taken). Further backtracking results in the rejection of (6,2,5,1), (6,2,5,4), (6,2,1), (6,2,3), (6,2,4), (6,5), (6,3), (6,4) and finally the subtree beginning with (5) is explored. The value of 1263 corresponding to the optimum permutation is then reached and the search terminates after 43 nodes have been examined. Only 9 stack locations were required for storage during the search.

2.4 The behaviour of the branch-and-bound algorithm upon the job-shop scheduling problem

2.41 Review of previous work

The application of the branch-and-bound method to the job-shop scheduling problem appears to have been first suggested

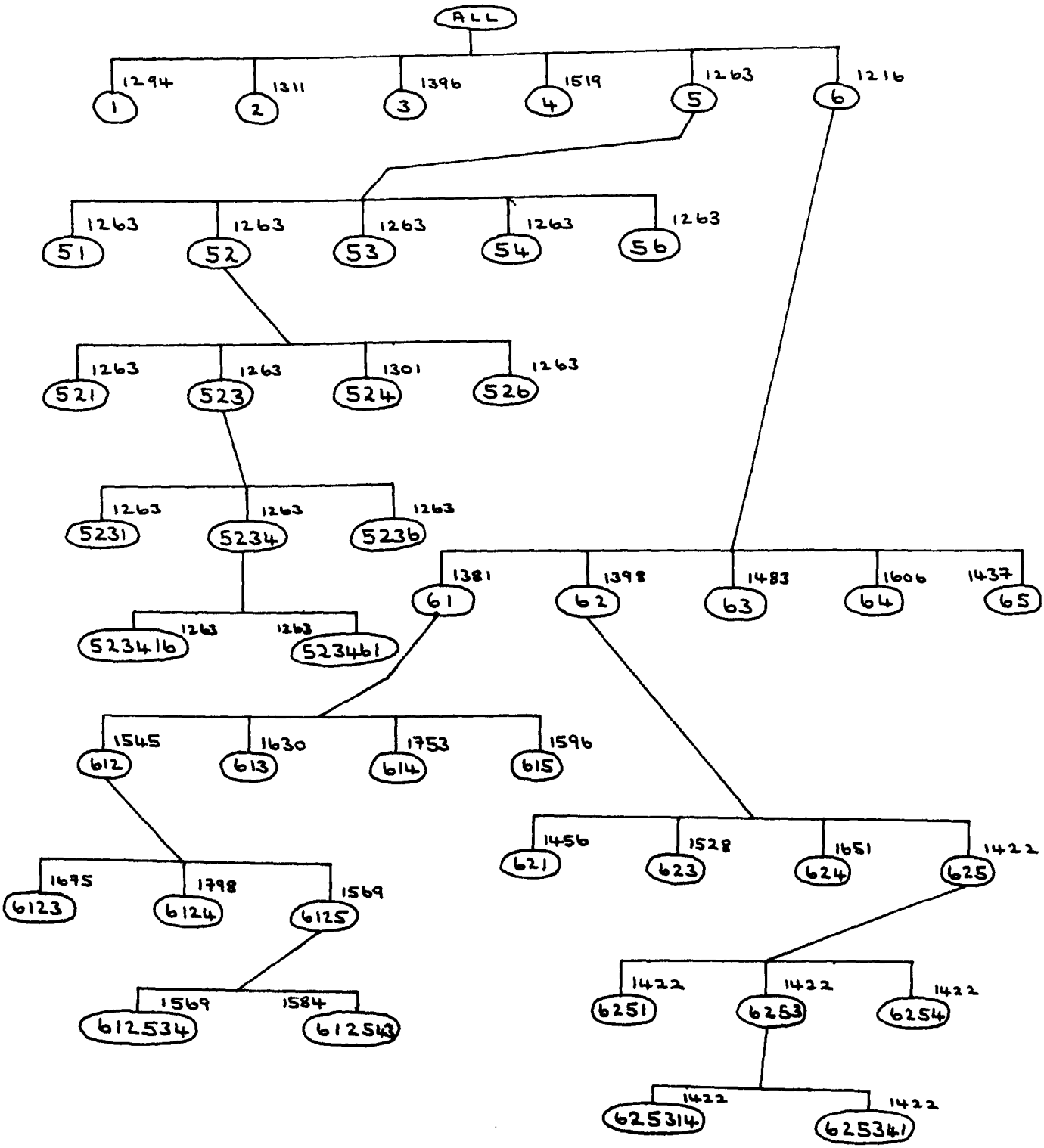


Figure 9: The tree examined under backtracking.

independently by Lomnicki, and Ignall and Schrage. Lomnicki advocated a backtracking approach to the three machine problem and claimed that 'an efficient algorithm can be given for finding the exact solution to the job-shop scheduling problem on three machines with comparatively little effort! . Furthermore he asserted that 'for a higher number of jobs there should be no difficulty in finding the exact solution very quickly with the aid of an electronic computer!. Lomnicki recognised that 'the efficiency of the process rests very strongly on the devices used to split the subsets and to find the bounds' and he felt that 'the three lower bounds appear to be quite efficient but it is possible that some better method might be found'. Lomnicki also introduces the idea of the 'reversed job-shop scheduling' problem. It is known that if $P_n = i_1, i_2, \ldots, i_n)$ is a permutation of n jobs upon m machines with cost $C(P_n)$, that the 'reverse problem' of scheduling n jobs upon m machines $M'_1, M'_2, \ldots, M'_m$ where the time taken for the jth job upon the kth machine is $d'(j,k) = d(j, m+1 - k)$ will have a cost $C(P_n)$ for the permutation $Q_n = (\ell_1, \ell_2, \ldots, \ell_n)$ if $\ell_t = i_{n+1-t}$ for $t = 1, 2, \ldots n$. The reverse problem for that given in table 4 is given in table 5 and the cost of the permutation (6,1,4,3,2,5) is 1263.

	Machines		
Jobs	I'	II'	III'
1	6	141	153
2	197	69	242
3	313	228	168
4	173	211	308
5	311	221	42
6	0	129	87
Totals	1000	999	1000

Table 5: The 'reversed problem' for Table 4

An intuitive representation of the reverse problem is to consider a film of the processing of jobs under a sequence $(i_1, i_2, \ldots, i_n)$. The running of the film backwards at the same speed would correspond to the sequence $(i_n, i_{n-1}, \ldots, i_2, i)$ being run through machines $M_m, M_{m-1}, \ldots, M_1$ and would of course take the same time to run.

Lomnicki states that the relative loading of the machines can appear to affect the performance of the backtracking algorithm and that it appears that the following rule can reduce the amount of computation required to solve a particular problem. If the first machine is most heavily loaded then apply the algorithm to the reverse problem, whilst if the third machine is most heavily loaded the algorithm should be applied to the normal problem. If the middle (the second) machine is most heavily loaded then one would not expect much difference in computation no matter which algorithm is applied.

The paper of Ignall and Schrage considered two objective functions for the job-shop scheduling problem: i) the minimization of makespan and ii) the minimization of the mean of completion times. A branch from the lowest bound approach is advocated and the authors also introduce a concept of 'dominated nodes' which can reduce the amount of computation required and also decrease the maximum list size for nodes upon which investigation has been temporarily suspended. If $P_r = i_1, i_2, \ldots, i_r)$ and $P'_r = (i'_1, i'_2, \ldots, i'_r)$ are part permutations such that $(i_1, i_2, \ldots, i_r)$ is a permutation of $(i'_1, i'_2, \ldots, i'_r)$

then $f_{P_r}(i_r,1) = \sum_{s=1}^{r} d(i_s,1) = \sum_{s=1}^{r} d(i'_s,1)$

$= f_{P'_r}(i'_r, 1)$. If it transpires that $f_{P_r}(i_r,k) \leq f_{P'_r,k})$ for $k = 2,3, \ldots, m$

then we say that P_r dominates P'_r and it is clear that any schedule beginning with P'_r can only be improved by replacing the first r elements with P_r. Thus even if the lower bound associated with P'_r might indicate that P'_r is worth examination it can be discarded if such a node P_r has been previously examined.

A paper by Brown and Lomnicki (7) comments upon the application of the concept of dominated nodes and also about the usefulness of the reversion rule of Lomnicki. Whilst both ideas are useful it is recognised that 'with the generalisation to more machines both refinements .... become less and less useful with the increasing number of machines'. The application of the dominated nodes concept has been observed to reduce the number of nodes to be examined by about 13 per cent on average. The reversion technique it is claimed could lower the number of nodes by about 33 per cent. The experiments from which these results are obtained used problems in which the number of jobs to be scheduled ranged from 4 to 10 whilst the number of machines ranged from 3 to 7. The algorithms were coded in various languages (e.g. FORTRAN, CLP (Cornell List Processor))

and the resulting programs run upon various computers (ICT 1301, CDC 1604). The comparison of execution times for the solution of the problems was therefore of little value and more meaning can be obtained from the results if the measure of computation is taken as the number of nodes examined in finding a solution. Another difficulty encountered is that of data generation; the method of generating the job-machine times differed from author to author and again made comparison difficult.

A paper by Waller reported an investigation into the problem which used the backtracking approach suggested by Lomnicki. The algorithm was programmed in Algol for the English Electric KDF9 Computer and job-machine times were generated randomly in the range 0 to 30 by using the multiplicative congruential random number generator.

$$X_{n+1} = 5 \times X_n \pmod{2^{35}}$$

The expected total load (or processing time) for each machine was thus the same. However, the actual loads for any of the machines were very rarely the same because the method of generation did not attempt to arrange this. The examples were 'small' in that from 5 to 9 jobs were to be processed upon 3 to 7 machines. It was seen that even for problems of the same size (same number of jobs and same number of machines) the amount of computation required varied greatly. Some problems would be solved after a small number (<100) of vertices had been examined whilst others would require in excess of 10,000 vertices for their complete solution.

This agrees with the experiences of Brown and Lomnicki and it appeared that the relative loadings of the machines might be connected with the unpredictable behaviour of the algorithm; again a conclusion arrived at by Brown and Lomnicki. It also appeared that, on average, the number of vertices examined in solving a problem was a linear function of e^{kn} where k is a constant approximately 2.3. The number of machines did not appear to significantly affect the computational effort required.

A 'good' starting solution obtained by heuristic methods did little to reduce the computation necessary for the backtracking algorithm, and use of the branch and bound approach to produce all optimal solutions seemed to be impractical since the effort to produce just one solution could be great even for small numbers of jobs.

2.42 Features of the algorithm requiring investigation

The following points were considered and investigated.

(i) The relative loadings of the various machines appears to affect the performance of the algorithm, can such loadings be exploited?

(ii) How useful is the reversion technique proposed by Lomnicki?

(iii) How useful and practical in terms of computer storage is the dominated nodes attack advocated by Ignall and Schrage?

(iv) Would the method of branching from the lowest bound give better performance than that of backtracking?

(v) Can the lower bounds proposed be strengthened?

(vi) What are the practical limitations of the algorithm?

2.421. Data Generation.

Initially the effect of the relative loading of the machines upon the behaviour of the backtrack algorithm was investigated. It was decided that one should be able to specify that a particular machine was to have an excess load of W% and data was generated in the following manner.

Job-machine times were generated, using the random number generator of 2.41, in the range 0 - 30. The smallest of such times was then ascertained and subtracted from all the times, ensuring of course that at least one time had the value zero. For each machine k the total loading, T_k, was calculated and each time upon the machine multiplied by $100/T_k$ so that the total load for each machine was 100 units. (In the case of the machine specified to have an excess load the multiplication factor was $(100 + W)/T_k$). Table 4 shows data generated in this fashion; the times have been sealed by a factor of 10 and rounded to integers for convenience.

An attempt at standardisation of data was made by making one of the generated times zero. If a constant c is added to each job-machine time the relative loadings of the machines will alter, the cost of any schedule will increase by $(n + m - 1)c$, but the optimum permutation and the behaviour of the branch-and-bound algorithm will remain unchanged. Hence, if the data is not standardised,any results obtained could be meaningless. Some approximate solutions to the problem have been assessed for merit on the percentage that the approximate cost is within the

optimum cost. Such claims should be treated with suspicion since by the addition of a suitable constant to each job-machine time the appropriate percentage could be made as low as required. With the standardisation process above any percentage quoted would be the largest possible.

2.422 The effect of the relative loadings and the reversion technique.

Using data generated for n = 7, 8, 9, 10 and m = 3 it was seen that the relative loadings of the machines did appear to affect the computational effort needed for a backtracking approach. In general it appeared that the more heavily loaded the first machine the more computational effort required - a result that agreed with Brown and Lomnicki. The reversion technique could then pay dividends but cases were observed where this technique was of no use. In one example of 10 jobs upon 3 machines the loadings on all machines were the same and yet the normal problem yielded to backtracking after a few hundred vertices had been examined whilst the reverse problem required in excess of 20,000 vertices. The author felt that the relative loading was more a symptom of the problem than the cause and since the reversion technique would appear to be less useful for larger m, the approach was carried no further at this stage. It is however, resurrected in Chapter 5.

2.423 The usefulness of the concept of dominated nodes.

A slight modification to the original Algol program allowed its use to ascertain the benefits to be obtained by the application of dominance tests within a search. Additional storage space was provided for the earliest finishing times of nodes which had

been investigated. Since m - 1 times must be stored for each combination and $\sum_{j=2}^{n-2} \binom{n}{j}$ is the total number of combinations for which history may be needed the 'limited' core storage of the KDF9 ($\sim$ 10,000 48 - bit words) was claimed. Using the storage in an unsophisticated fashion a feasibility study of the dominated nodes approach was conducted. A program parameter specified the maximum number of pieces of information to be kept for any combination out of the integers 1, 2, . . ., n during the search. If this maximum was achieved for any combination then additional information for that combination was discarded and the fact noted. A pigeon-hole method of table lookup was adopted for any combination of j elements out of n, the key computed for a part permutation $(i_1, i_2, \ldots, i_j)$ being $\sum_{k=1}^{j} 2^{(i_k - 1)}$. The method was thus rough and ready and if anything would not show the results of dominance tests to their full advantage.

Some large reductions in the number of vertices examined were observed after the application of dominance tests. In one problem of scheduling 9 jobs upon 3 machines the use of dominance tests enabled the number of vertices examined to be cut from 10,000 (with no dominance tests) to 3,000 (with the tests). More computing time is of course required per vertex investigated. In an 8 jobs 3 machines problem it was found that the computing time taken with dominance tests was the same as the time taken to solve the problem without dominance checks although a saving of 40% in the number of vertices examined was achieved. However, as was pointed out earlier the main difficulty with this type of problem is not the amount of computation per permutation

examined but the number of permutations to be examined. As will be seen later the penalty for not discarding a part permutation could be the examination of a substantial part of the subtree emanating from the node associated with the part permutation. Linear increases in computational effort per vertex examined can thus be tolerated.

Dominance tests are thus particularly useful in curtailing computational effort for a problem. They do however present some storage and organisational problems and their effect is likely to be less pronounced for larger values of m.

2.424 A comparison between branching methods.

As stated earlier backtracking enjoys an advantage over branching from the lowest bound in that storage required for a backtracking problem is predetermined and small whilst the latter method's needs are unpredictable. A further advantage as far as the job-shop scheduling problem is concerned is that knowledge of the earliest finishing times for a part permutation $(i_1, i_2, \ldots, i_r)$ can be exploited for the calculation of the times for a part permutation $(i_1, i_2, \ldots, i_{r+1})$. Backtracking does however in general explore more vertices than are absolutely essential and a criticism of the method is that the search could spend a vast amount of time in a part of the tree from which the payoff could be small; the time might be better utilised in another part of the tree where a payoff could rule out the necessity for examination of the former sub-tree. Careful examination of results obtained from backtracking has enabled the author to answer this criticism to a certain extent and also obviated the need for coding of the branch

from the lowest bound method. Figure 10 shows that for the back-tracking algorithm the problems that gave most trouble (in terms of the number of vertices searched) examined most of the vertices in order to verify that an optimum had been found; i.e. the main effort of the computation was involved in searching the remaining parts of the tree for an improvement in what transpires to be the optimum. Thus these nodes examined must have had lower bounds less than the cost of the optimum and would require examination no matter which branching method was adopted.

This result is useful in two ways. Firstly it saved substantial programming effort and investigation, and secondly it pointed out the most serious defect of the branch-and-bound algorithm when applied to job-shop scheduling - the fact that once the optimum has been discovered the bounds calculated were not stringent enough to detect it. This also explains why in the earlier investigation by Waller the existence of a good starting (approximate) solution was of little help.

2.425 The Limitations of the Algorithm

The previous investigations have shown that the algorithm with perhaps the use of a good starting value, the reversion technique and the use of dominated nodes would be very unlikely to be capable of tackling 'large' ($n \gtrsim 20$) problems, within a reasonable amount of computing time. Its use to provide all optimum solutions would be even more restricted. It appears that the lower bounds calculated are not stringent enough and unless they are made stronger the branch-and-bound algorithm for the job-shop scheduling problem could only be used as an approximate method for large problems.

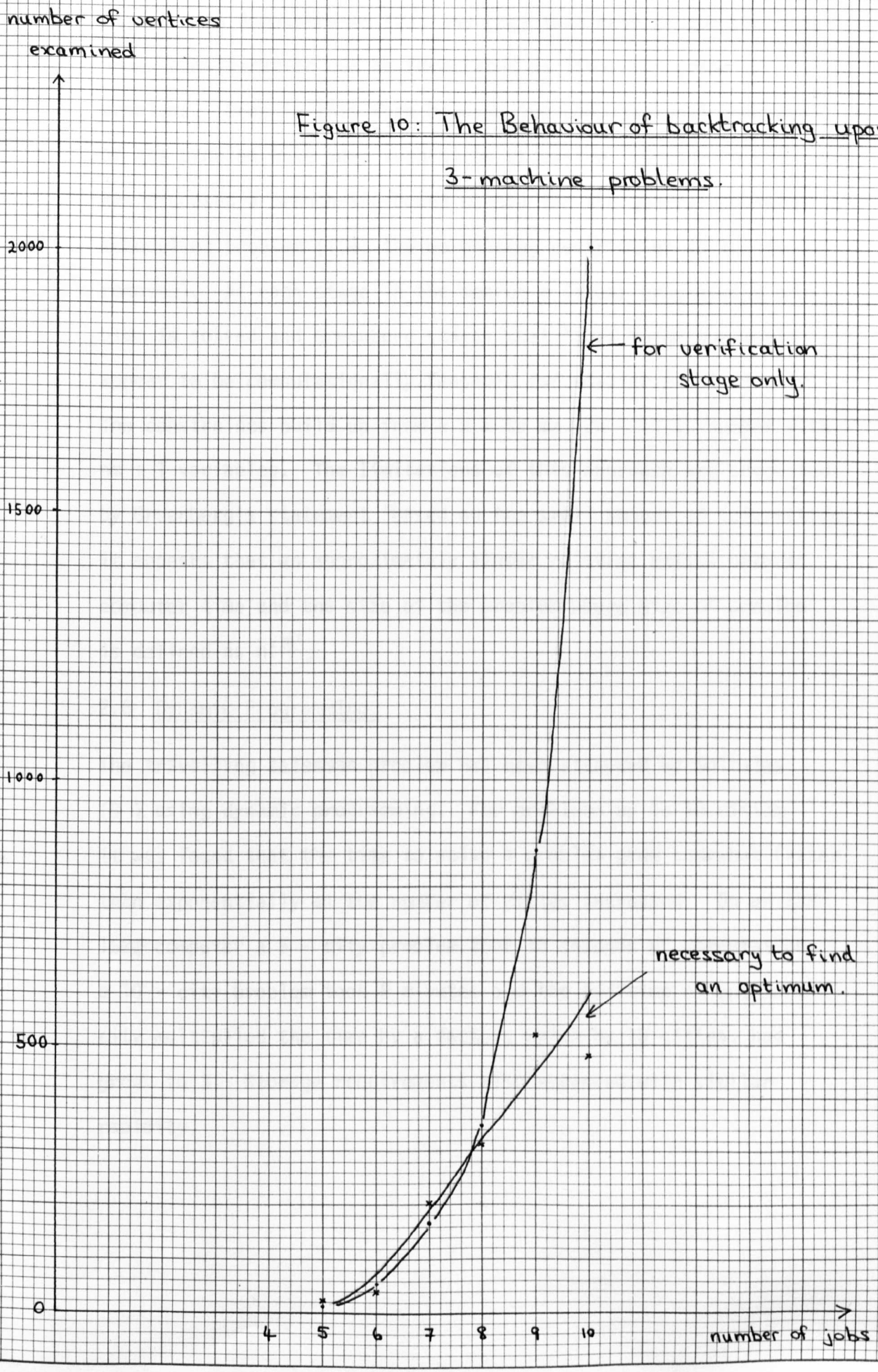


Figure 10: The Behaviour of backtracking upon 3-machine problems.

2.5 Integer Linear Programming

It has been demonstrated (23) and (26) that many of the problems under consideration can be formulated as linear programming problems. Once in this form however the computational demands of the problem have been seen to be excessive. Greenberg (8) has proposed a branch-and-bound method for the general job-shop scheduling problem (i.e. the no-passing restriction in 1.1 is removed). At each stage in the search a linear programming problem is solved and Greenberg states that 'this technique would appear to be computationally feasible for smaller problems only' The constraints build up from n x m to mx (2n - 1) in number with nm + 1 variables. The limitations of this approach are suggested by Greenberg's statement 'based on the small amount of information, it appears that an approximately linear relation exists between computer time and the size of the problem where size of problem is defined as $n!^{m'}$.

2.6 Dynamic Programming.

The travelling salesman problem has been attacked by Golzales (9) using dynamic programming. For problems up to 10 cities it was found that the time to solve a problem grew somewhat faster than exponentially as the number of cities increased. On an IBM 1620 computer a 5 - city problem took 10 seconds of computation, a 10 - city problem took 8 minutes and the addition of one more city multiplied the time by a factor, which by 10 cities, had grown to 3. Storage requirements also expanded with similar rapidity.

Equally gloomy results have been reported by Held and Karp (10) using an IBM 7090 computer. Problems of up to 13 cities have been solved and Little et al (11) state that 'a 13 city problem took 17 seconds and calculation reveals that if their (Held and Karp) computational demand grows at the same rate as that of Gonzales, a 20 city problem would require about 10 hours and storage space requirements may have become prohibitive by then!.

Chapter 3. Heuristic Techniques.

3.1 Local Rules

Many heuristic techniques that have been applied to discrete optimization problems have been special purpose techniques that have been developed for particular cost functions. As an example, the first heuristic methods applied to job-shop scheduling were loading rules which were local rules in that the machine operator would use them to determine which of the waiting jobs to process next. As this individual was unlikely to possess computational aid the rules had to be simple so as to be easily applied. Possible loading rules are:

1) Process the job that arrived first. (FIFO - first in, first out).
2) Process the job that can be finished most quickly. This is the SIO ('shortest imminent operation') rule.
3) Process the job that has the longest remaining time of processing. (LRT rule).
4) Choose a job at random and process it.

The rules are described as though they lead to a unique set of decisions and as this is not always the case some tie-breaking mechanism must also be employed. Many of the rules have a certain amount of logic behind their conception. FIFO is intuitively fair in the sense that if customers are awaiting the finished product, the completion order of the jobs is the same as the starting order. The SIO rule adopts the philosophy

of satisfying as many customers as possible within a fixed period of time but does not allow for the possibility of larger jobs waiting an unreasonable amount of time for processing. The drawbacks in the application of loading rules occur because only the characteristics of the load for a single machine are considered and the local cost function applied need not reflect the overall cost function. Thus the resulting schedule may be poor. A further disadvantage is that, like many heuristic approaches, local rules give no indication as to how good is the solution they produce. There is thus no criterion for deciding whether or not to attempt to improve the solution obtained with, perhaps, a different heuristic.

3.2 Branch-and-Bound as a Heuristic

The branch-and-bound method, as described in Chapter 2, would appear to be computationally impracticable for the exact solution of the job-shop scheduling problem from the results of the investigation performed. However, by slight modifications to the method, heuristic approaches can be obtained. An advantage of such approaches is that some measure of goodness of the solution derived is also available.

3.21 Branch-and-Bound without Backtracking.

Ashour (12) suggests a method which he terms 'branch-and-bound without backtracking'. In this branching from the newest active bounding problem is performed and the search terminates when the bottom of the permutation tree is reached. The cost function is thus applied for two permutations only, and the one with the lowest associated cost is adopted as an approximate solution. A measure as to how good this solution is can be determined by

scanning the unexamined part of the tree and ascertaining the lowest lower bound. Thus it is known what might have been achieved.

The quality of solutions given by this approach for the job-shop scheduling problem is defined by Ashour as the quotient of the optimal schedule time and the approximate solution. Figures given for this 'efficiency' are provided in the paper but it should be remembered that the experimental data for the problems concerned was not standardized as described in 2.421. Furthermore, the figures were obtained for small problems ($6 \leq n \leq 12$) and there is no guarantee that the efficiency will be as high if larger problems are considered. The approach does however have merit in that an attempt is made at limiting the amount of computation expended in obtaining a solution.

3.22 Seeking a reasonable improvement upon a particular value.

This method also attempts to limit the computation expended upon a problem. Whenever a new target value is discovered a decision is made to look for another solution which will have a cost a certain amount lower than the current targetvalue. The improvement sought would normally be a function of the targetvalue and the lowest unexamined lower bound in the tree. Parts of the tree that might possibly give a small improvement in the lowest cost found so far can thus be rejected with the result that the computational effort might be curtailed. (There are of course cases where the search time might be increased). By retaining the lowest lower bound discarded the value of the optimum can be fixed within certain limits and thus a measure of goodness of the approximate solution can be obtained. This technique is discussed further in 5.44.

3.23 A Solution in a Bounded Period of Time.

Lawler and Wood suggest that the branch-and-bound method might be used to find as good a solution as possible in a bounded period of time, T, in the following manner. Initially a search is made for an optimum solution in a time T/2. If no such solution vector is found then a solution will be sought that differs from the optimum by no more than, say, 5 per cent. A time of T/4 will be allocated for this search and if a vector is found in this time, any remaining time can be spent trying to improve upon the solution found. If no such solution vector is found in the time T/4 the process of bisecting the remaining time and looking for a more approximate solution can be repeated. At any stage the estimate of the optimum will be the lowest lower bound still active in the tree. Implementation of a scheme of this nature requires a 'branch from the lowest bound' approach initially, whilst features of backtracking would be employed later.

3.3. Heuristics based upon Sorting methods

The job-shop scheduling problem is that of optimizing a given function of several arguments. When the no-passing restriction is removed there are many restrictions.

The job-shop scheduling problem with the no-passing restriction of 1.1 removed is that of optimizing a given function of several arguments $C(\nu)$ where there are many restrictions, so that no job is attempted on a machine before the operations it requires on all earlier machines have been completed. Page (16) has pointed out that with this formulation sorting is a much simpler special

case of the scheduling problem. 'If x_i is the key of the i th item, then the sorting problem can be regarded as equivalent to minimizing a function $f(x_i, x_j, \ldots, x_p)$ which is zero if $x_i \leq x_j \leq \ldots \leq x_p$ and takes positive values if any of the inequalities is reversed'. Sorting is a process frequently performed by hand and by machine and consequently many methods exist all of which avoid the generation of all possible permutations of the orders of the x_i - a feature of the black box problems. Page proposes the use of sorting methods for attacking black box problems on the principle that 'if no fairly simple and short method of finding the best order can be given we would like first, a method which will usually, if not always, produce a fairly good order for a first trial, and second, a method for deriving a better order from the one we have'. Of the three sorting methods, selection, merging and exchanging which follow the first two attempt to produce a good initial solution whilst the latter tries to improve upon an existing approximate solution.

3.31 Selection

For permutation problems the selection method of sorting can be applied to build up an approximate solution permutation in the following manner. Each element of 1, 2, . . ., n is tried as the initial element of a permutation of one element and submitted to the cost function. The element, i_1, which gives the lowest cost is then chosen as the first element of the approximate solution being formed. The cost function is thus applied n times at this stage. Those elements not yet allocated (n - 1 in number) are now considered for the next (i.e. second) position

of the permutation by applying the cost function to permutations of two elements. Each of these permutations will be of the form (i_1, j) where $j \in I_n \sim \{i_1\}$ and the permutation possessing lowest cost is retained. The next and subsequent positions in the permutation are filled in a similar fashion; at each stage the cost function will be applied one time fewer to permutations having one more element and finally a complete permutation will be reached and adopted as an approximate solution. The selection method is in fact an application of local rules of the type in 3.1 and possesses similar characteristics. The following sorting methods of exchanging and merging are not local rules and their application usually gives better results.

3.32 Exchanging

This method attempts to improve upon an existing approximate solution $P_n = (i_1, i_2, \ldots, i_n)$. The cost of the permutation is compared with the cost associated with the permutation with the n th and (n - 1) th elements interchanged. If the permutation with the interchange is deemed better (i.e. has lower cost) than that without the interchange then the elements i_n and i_{n-1} are left interchanged, otherwise they are placed in the positions they occupied before the switch. The process is repeated with positions (n - 1) and (n - 2) as candidates for the interchange and continued until finally positions 2 and 1 of the permutation have been considered. A series of possible interchanges has then been attempted upon the permutation and if any improvement upon the original cost has been achieved, the series is repeated. The process terminates when one complete pass through the permutation fails to give an improvement in the cost of the approximate solution.

3.33 Merging

Merging builds up an approximate solution by forming ordered pairs of elements (k_1, k_2), (k_3, k_4), then ordered quartets, ordered octets and continues until a single complete permutation is reached. The ordered pair (k_1, k_2) is adopted if the cost of the part permutation (k_1, k_2) is less than the cost associated with (k_2, k_1). (In the event of a tie an arbitrary choice is made). Ordered quartets are formed from pairs of ordered pairs. If (k_1, k_2) and (k_3, k_4) are ordered pairs then k_1 is chosen as the first element of the ordered quartet if the cost of (k_1, k_3) is less than the cost of the part permutation (k_3, k_1). Suppose that k_3 is chosen. The second position of the quartet will be taken to be either k_1 or k_4 and is chosen by considering the costs of the two part permutations (k_3, k_1, k_4) and (k_3, k_4, k_1). If the cost of (k_3, k_1, k_4) is less than that associated with (k_3, k_4, k_1) then k_1 is chosen, otherwise k_4 fills the second position. Supposing k_1 to be selected the costs of (k_3, k_1, k_2, k_4) and (k_3, k_1, k_4, k_2) are determined and the part permutation with the lowest cost taken as the ordered quartet. A similar procedure leads to the formation of the ordered octets and so on.

3.34 Characteristics of Sorting Methods

The selection method of sorting applied to discrete optimization problems readily illustrates a feature of many heuristic approaches. When the black box under investigation does not yield easily to analysis another black box is substituted for it. The second black box is generally simpler and hopefully more cooperative. Its characteristics should however approximate to those of the

original. The selection method replaces the black box which accepts a permutation of (1, 2, . . ., n) by one which will accept a subpermutation of (1, 2, . . ., n). Ashour's method in 3.21 for the job-shop scheduling problem substitutes a black box which accepts subpermutations and produces lower bounds based upon them. It would be expected that this method would be superior to selection since some consideration is given to elements that have not been assigned whereas selection does not utilise any such knowledge.

3.341 The 'Goodness' of Solutions

All the sorting methods described possess the disadvantage that, in general, there is no guarantee as to the merit of the particular resulting approximate solution and its associated cost. However, for certain classes of cost functions the merging method can be guaranteed to produce an optimum solution. Smith (13) has shown that if $\overline{P}_n = (\overline{i}_1, \overline{i}_2, \ldots, \overline{i}_n)$ and $P_n = (i_1, i_2, \ldots, i_n)$ are permutations of 1, 2, . . ., n and C is a cost function over the set of all permutations of 1, 2, . . ., n , a sufficient condition that $C(\overline{P}_n) \leq C(P_n)$ for all P_n is that:

(A) there is a real-valued function g of ordered pairs of elements such that if P_n is any permutation and P_n' the permutation obtained from P_n by the interchange of the j - th and (j+1) - th elements, then $C(P_n) \leq C(P_n')$ if $g(i_j, i_{j+1}) \leq g(i_{j+1}, i_j)$ and

(B) $\overline{P}_n$ is such that the i-th element precedes the j-th if $g(i,j) < g(j,i)$.

If the function g is known then an optimum permutation may be achieved by ordering the elements of the permutation according to the corresponding function values. Johnson's complete solution

for the scheduling of jobs upon two machines specifies g by $g(i,j) = \min \{d(i,1), d(j,2)\}$. Thus job i precedes job j in an optimum solution if $\min \{ d(i,1), d(j,2)\} < \min\{d(j,1), d(i,2)\}$. There is no extension of this rule to more than two machines except in special cases. Smith has determined appropriate functions for particular cost functions and Rau (15) shows how to determine the g function when $C(i_1, i_2, \ldots, i_n)$ has the form:

$$C(i_1,i_2, \ldots,i_n) = \sum_{k=1}^{n} h(i_k)\, g_k\, (i_1, \ldots, i_k)$$

where h is a positive function of one integer variable, and each g_k is a function of k integer variables.

The determination of the appropriate function g is however, in general, not trivial; indeed such a function may not exist! Page has shown that the merging method does have the advantage that if such a function exists, merging will produce an optimum solution without the necessity of the function g being discovered. Clearly it will do so for the case n = 2, and (for larger n) as the algorithm proceeds the elements of the permutation will fall into the optimum order for the subsets of the permutation that are considered. Merging could be used in a different fashion; if one suspects that a function C possesses a function g (as specified above) experimental runs of the merging algorithm compared with complete enumeration results (upon small problems) would (if the results differ) prevent effort being expended searching for a non-existent function g.

The sorting techniques can be applied to cost functions which do not possess an appropriate g function. They do not then guarantee

an optimum solution but quite often 'good' results are achieved. Page reports that for the job-shop scheduling problem the computing times for the methods increase slowly with the number of machines and merging and exchanging computing times increase by about a factor of 3 for each doubling of the number of jobs. He stated that 'the sorting type methods are quite general' and 'all that is required is an unambiguous specification of the method of calculating of the cost function'. The use of exchanging upon the resulting permutation of the merging procedure is also of help in the cases where the approximate solution produced by the latter method is not particularly 'good'. The main advantage of the exchanging method is that the greater part of computation expended is used in achieving a decrease in the cost of the permutation being explored. Only one 'fruitless' cycle of calls of the cost function is made. (This was of course one of the major drawbacks of the backtracking algorithm). The storage requirements of the algorithm are modest but exchanging should really be used only upon a permutation in the 'neighbourhood' of the optimum as it appears that improvements gained are in general small ones.

3.4 Monte Carlo Methods.

A simple approach to finding the permutation of 1, 2, . . ., n which minimizes some particular cost function would be to sample randomly from the n! possibilities. At each stage the cost associated with the permutation generated could be determined and the permutation which gave the lowest cost taken as an approximate solution. One of the disadvantages of this random

sampling is that no use is made of previous observations in subsequent sampling and thus information that might have proved useful is disregarded. Page (17) suggests that some measure of likeness between permutations could be formulated so that like permutations would exhibit similar characteristics, in this case have similar costs. If this were the case the sampling of permutations could be adjusted so that once a 'good' permutation has been discovered the sampling should be conducted among permutations that are near to this permutation, or in effect in a neighbourhood of the 'good' permutation. The concept of a metric in the space of permutations is suggested by this approach.

3.41 The Metric in the space of permutations

If P^n is the set of all permutations of (1, 2, . . ., n) then the function $s : P^n \times P^n \to R$ is a metric on the set P^n if it satisfied the axioms

1. $s(x,y) \geq 0$ for all $x, y \in P^n$.
2. $s(x,y) = 0$ if and only if $x = y$.
3. $s(x,y) = s(y,x)$ for all $x,y \in P^n$.
4. $s(x,z) \leq s(x,y) + s(y,z)$ for all $x,y,z \in P^n$.

For $x \in P^n$ and $r \geq 0$ the subset $B(x,r) = \{ y : s(x,y) < r \}$ of P^n is defined as the open ball , centre x and radius r. A subset A of P^n is said to be a neighbourhood of $x \in P^n$ if A contains some non-empty open ball of centre x. A neighbourhood of a permutation is thus defined.

If $x = (x_1, x_2, . . ., x_n)$ and $y = (y_1, y_2, . . ., y_n) \in P^n$ then several metrics can be defined. The euclidean metric

$$s(x,y) = \left\{ \sum_{i=1}^{n} |x_i - y_i|^2 \right\}^{\frac{1}{2}}$$

and the metric

$$s_1(x,y) = \max_{1 \leq i \leq n} |x_i - y_i|$$

are both possibilities which may be used to define likeness between permutations. However, for cost functions like than of the job-shop scheduling problem it is reasonable to suppose that orders (or permutations) containing the same subsequences of jobs will tend to have cost functions near to one another. The euclidean metric gives a distance of $\sqrt{150} \underline{\sim} 12.3$ between the permutations (1,2,3,4, 5,6,7,8) and (6,7,8,4,5,1,2,3) whilst the distance between (1,2,3,4,5,6,7,8) and (5,4,8,7,6,1,2,3) is $\sqrt{130} \underline{\sim} 11.4$; the first two permutations do however appear to be more like each other as far as subsequences are concerned! The maximum difference metric also fails to exhibit the subsequence property as can be seen from

$$s_1((1,2,3,4,5,6,7,8), (8,1,2,3,4,5,6,7)) = 7$$

whilst $s_1((1,2,3,4,5,6,7,8), (1,2,3,4,8,7,6,5)) = 3.$

A measure that does give consideration to subsequences of elements has been proposed by Page. The distance between two permutations is taken as the number of elements in the first permutation that are not followed by the same element as they are in the second. Thus the number of breaks in the order of the permutations is counted. In the examples above

$$s((1,2,3,4,5,6,7,8), (6,7,8,4,5,1,2,3)) = 2$$

$$s((1,2,3,4,5,6,7,8), (5,4,8,7,6,1,2,3)) = 5$$

$$s((1,2,3,4,5,6,7,8), (8,1,2,3,4,5,6,7)) = 1$$

$$s((1,2,3,4,5,6,7,8),(1,2,3,4,8,7,6,5)) = 4$$

This measure of likeness satisfies the four axioms of a metric space and the distance, s, between any two permutations of the elements (1,2, . . ., n) is thus integer valued and less than n. All permutations of (1,2, . . . n) thus lie on the perimeters of the closed balls $B'(x,r) = \{ y : s(x,y) \leq r \}$ for any $x \in P^n$ and $0 \leq r < n - 1$.

Rephrasing all permutations of (1,2, . . ., n) belong to the family of open balls $B(x, r + \delta) = \{ y : s(x,y) < r + \delta \}$ for $x \in P^n, \delta > 0$ and $0 \leq r < n - 1$.

The open balls $B(x, r + \delta)$ thus form a neighbourhood of the permutation x. This allows a Monte Carlo approach to be adopted in which initially permutations are generated in a neighbourhood of a known good permutation. The neighbourhood is first chosen as the open ball of radius $q + \delta$, for some integer q and $\delta > 0$. The cost of each permutation is calculated and if it is lower than that of the known good permutation it replaces the known good permutation. If N samples are taken without improving upon the base permutation the value of q is halved and the process repeated. Termination occurs when permutations at a maximum distance of 2 from the base are to be sampled. It can be seen that when it appears that a particularly good solution has been reached the search concentrates amongst those permutations near to it.

Page has shown that the measure of distance as defined readily lends itself to a search of this nature. No great difficulty is

found in generating permutations within a distance of another permutation and it has been shown that solutions produced by Chain Monte Carlo are normally better than those produced by crude sampling. He recommends that neither crude nor Chain Monte Carlo be used for problems for which satisfactory deterministic methods are available. They might however be used as a last resort, or to give a good quick solution and techniques such as the Chain Method should be used for increasing the power.

Chapter 4.

An Interactive Approach

As has been illustrated, there exist cost functions which do not yield to analytical solution, or for which the amount of computation required for the determination of an optimum solution is excessive. One must then accept the results of heuristic methods which give a 'good' solution and attempt to back up the value obtained with an assessment as to how good the solution is. The heuristic method suggested here is an interactive approach which utilises the marriage of human intuition with the facilities provided by a modern multiprogramming computer to improve upon the 'good' solutions supplied by approximate methods of the type described in chapter 3. The idea is for the human to employ his judgement and/or intuition to determine those areas in the solution space which might prove worthwhile and to unleash the computational effort of the computer upon them. Experience of the success obtained (or lack of success) in a particular area would provide a feedback and might then suggest other possibilities.

4.1 Necessary Computing Facilities

Basic computing facilities required for such an approach were provided by the IBM 360/67 computer at Newcastle and economics determined that a multiprogramming operating system be used since interaction requires comparatively large periods of 'think' time by the user. Terminal facilities were also necessary and the early stages of the research were conducted using an IBM 2741 terminal typewriter whilst in later stages IBM 2260 character

displays were available. In order to provide reasonable response whenever the user required service it was also essential for the controlling operating system to time-slice the many programs it was running. This necessity was satisfied by the TSS (Time Sharing System) of IBM under which the early development of the interactive approach took place, and also by MTS (the Michigan Terminal System) which succeeded TSS.

The hardware and system software facilities were thus available in Newcastle for an interactive approach and a system entitled IMPACT ('An Interactive System for the Manipulation of Permutations for an Attack upon a class of Combinatorial Problems') was designed and implemented.

4.2 Design Objectives of IMPACT

Several objectives were formulated in the design of IMPACT. An ability for IMPACT to be applicable to a wide range of discrete optimization problems was immediately obvious. A set of powerful heuristics was desirable and interactive access was essential in order to permit decision taking and provide immediate feedback from the system to the decision taker.

The access to the system was to be easy to use so that the user could easily concentrate his intellectual effort into the problem-solving and not be encumbered with troublesome system details. Furthermore whilst the system was to be easy to use it should also be difficult to abuse in the sense that any typing or logical errors by the user should not result in a catastrophe. Such events should be detected as minor errors and be catered for by IMPACT so that the user could be prompted to remedy his mistake.

Initially all the facilities that were eventually incorporated into IMPACT were not anticipated. Extendability was thus a necessity as was flexibility in the sense that IMPACT ought to easily lend itself to tailoring for different operating systems. (This situation actually occurred as IMPACT was being constructed; the early stages were conducted under IBM's TSS and the later stages under the Michigan Terminal System (MTS)).

4.3 Achievement of Design Objectives

To allow flexibility and to attempt to achieve machine-independence it was decided that IMPACT ought to be written in a high level language. Under the version of IBM's TSS available at Newcastle at that time the only high level language provided was FORTRAN and this language was thus adopted. One disadvantage of this language is its lack of dynamic storage allocation. This was not catastrophic however, and the ability to compile subroutines separately and link them at load time (a facility not easily invoked with certain other languages) proved particularly useful. A change from TSS to MTS at Newcastle has subsequently meant that several other languages (e.g. Algol, APL, PLI) are now available. It is not unlikely that certain features of these languages (e.g. APL's vector operations) would have been useful. Recourse to ASSEMBLER language has been permitted only where absolutely necessary, namely to access certain system functions. The current version of IMPACT uses only three such routines; one provides information as to whether an IMPACT session is interactive or not, another gains access to computing time used by IMPACT, whilst the third allows the use of the ATTENTION

(or response request) button on the terminal being used.

Extendability is provided for by modularity in the programming of IMPACT. Each facility provided occupies its own module and has access to a common data base (large areas of COMMON blocks). A skeleton of a main program provides a switching network between these modules and hence additional features may be easily incorporated by the writing of an appropriate module and a slight modification to the main program.

Modularity also made possible the objective of being able to attack a wide class of combinatorial problems. In order to use IMPACT upon a different problem of the class defined in Chapter 1, a user must write a suitable module to calculate the cost of a particular input to his particular black box function. This and a suitable routine to place any data in the common data base within IMPACT allows use of many of the powerful heuristics that exist in the system.

Use of corrective programming within IMPACT allows the system to protect the user from himself. Checking is performed to ensure that errors by the user are trapped. Typically, if the user enters what he considers to be a permutation of 1 to n (where n is some positive integer) a suitable subroutine TESTPE performs a test to see that this is so. Errors result in a meaningful diagnostic message being printed out and if the problem solving effort is conversational a chance to re-enter is provided for the user. Nonconversational errors result in the termination of the session.

Corrective programming also helps to make IMPACT easy to use

as does the use of a single command-parameter analyser. This means that, with very few exceptions, the format of the user's input is uniform throughout IMPACT. Positional parameters (i.e. of the form a_1, a_2, a_3 where a_1 is the value of the first parameter for the command, a_2 is the value of the second etc.) are used. Keyword parameters (i.e. of the form para 1 = a_1, para 3 = a_3, para 2 = a_2 in which the order of the parameters is unimportant) are specifically excluded. The implementation of IMPACT was simplified by such a restriction but the modular nature of IMPACT would allow the future introduction of keyword parameters, since only the parameter string processor would require reconstruction. Ease of use also stems from a ready feedback from IMPACT to the user and from the provision of facilities which permit the user to interrupt IMPACT at will. Interrogation by the user is then available as is modification of certain major variables. Finally the user may instruct IMPACT to continue the interrupted process or to abandon this process and initiate another (possibly different) one.

IMPACT possesses a powerful set of heuristics, some of which have been described in Chapter 3 whilst others grew as interactive problem solving with the system suggested them. Currently facilities that exist are biased towards discrete optimisation problems where the input vector to the cost function is to be a permutation. The extendability of IMPACT easily lends itself to the provision of other heuristics. Some of the more important heuristics are discussed later but a complete list of them is given in Appendix 1, together with brief explanations and examples of their usage.

4.4 Requirements of IMPACT from the Operating System

IMPACT appears to the operating system as one large program and an interactive session is invoked by instructing the operating system that the program is to be executed and is to have access to certain files. In MTS this is achieved by means of the MTS command.

```
$run impact   1 = datafile  2 = recordfile  4 = savefile
              5 = hash
```

which instructs that the object program in the file named 'impact' is to be loaded and executed; data is to be taken from the file 'datafile' and a set of command words is to be taken from the file 'hash'; 'recordfile' is a file onto which a permanent record of the problem-solving session is to be kept; 'savefile' is a file onto which items of information may be stored.

When IMPACT is running only a small portion of the compiled program will be actually kept in main storage since in essence IMPACT consists of a set of subroutines which share a common data base and are linked by a main program which performs the function of allowing the user to switch from one subroutine to another. Any subroutine called which does not reside in main storage is 'paged' into main storage from secondary drum storage by the operating system. The virtual storage requirement for IMPACT is currently about 85 pages (1 page = 4096 bytes) and only a small proportion of this (10 - 20 pages) is required in core at any one time. In a particularly interactive session the large amounts of 'think' time would

mean that almost all of IMPACT would reside on secondary storage should the main storage be required by other users' programs in MTS. The core storage requirements for IMPACT are thus not excessive.

Central processor usage time for IMPACT is dependent upon which facilities the user requires and how heavily he uses them. Most facilities provided require computing times of the order of a few milliseconds, whilst some involving powerful search techniques can demand computing times of a few minutes or more. Controls are however provided for interrupting or policing such time - consuming processes. As mentioned earlier the operating system must supply 'hooks' from which IMPACT can pick up three pieces of information concerned with 1) computing time used, 2) whether or not the session is conversational and 3) an attention handling facility.

4.5 Usage of IMPACT: Command Mode

Once the user has initiated the execution of IMPACT and has provided basic data to specify the particular problem he is attacking he can invoke any of a set of heuristics by simply specifying a particular command word whenever IMPACT places him in command mode. Command mode is recognised by IMPACT outputting the line,

\# \# \#

to the 2741 terminal and unlocking the keyboard. Commands consist of the name of a particular operation to be carried

out and may require certain parameters.

In command mode the user may enter a command name and be prompted for parameters, or he may enter the name and the parameters together. In the latter case the command name must be separated from the parameters by a comma; the parameters must be separated amongst themselves by commas and terminated by a semi-colon. A command name consists of from one to eight alphanumeric characters, the first of which must be alphabetic. An effort has been made to make command names meaningful, e.g. 'CMC' is used to invoke a Chain Monte Carlo technique. Any parameters entered are positional parameters and a single parameter may be an integer number or a permutation (part permutation) name. A permutation name must be up to four alphanumeric characters the first of which must be alphabetic. The lack of keyword parameters means that occasionally one must be prompted for certain parameters. A few commands (e.g. the MERGE command) will accept other special characters. (A description of any such peculiarities may be found in 'Command Descriptions' in Appendix 1.)

4.51 Interrupt Mode and Local Commands

The ability to interrupt execution of a command provides a powerful facility in IMPACT.

Various points in IMPACT have been tapped with tests to determine whether or not the attention button has been depressed. (The reason for implementation of this nature is to ensure that any command may only be interrupted at certain fixed

points so that adverse or unusual effects may not be introduced). A short time after the pressing of the attention button, IMPACT will inform the user that his attention request is being serviced. (From the user's point of view it appears that his request is being serviced here, whereas the request was actually serviced earlier by IMPACT). A second press of the attention button before the apparent servicing of the first press will return the user to the operating system (MTS) environment; in which case the MTS command '$restart' will cause IMPACT to resume. Certain commands are specified as being interruptable and the depression of the attention button places the user in interrupt mode. In such a mode a meaningful message will be displayed and will be followed by a single line

?

and the unlocking of the keyboard. The user may then enter local (or sub-)commands which allow the display and modification of major variables. As an example, the state of a search may be ascertained, modified (or perturbed) and resumed. (Examples of such techniques are given in Chapter 5).

When in interrupt mode information about local commands may be ascertained by inputting the local command 'HELP'. The local commands are handled in a more simple manner than the main commands and more information about them is available in 'Command Descriptions' in Appendix 1.

A single attention interrupt during the processing of any of the commands specified to be non-interruptable will initially appear to be ignored by IMPACT. A message

'YOU RANG, SIR?'

will however be printed before the user is next put into command mode and serves to remind the user that his attempted interruption was not allowed.

4.52 System Commands.

There are four classes of commands within IMPACT

1) Storage allocation commands which allow permutations or part permutations to be defined, deleted or displayed.

2) Commands for invoking heuristic methods which require that the user supply routines for calculating the cost of a permutation and for calculating a lower bound for a part permutation.

3) Sub-commands which are available whenever the attention interrupt facility is invoked.

4) Commands which are peculiar to the particular cost function being investigated. These have been incorporated in order to provide additional facilities in IMPACT. Typically, such facilities have been used by the author for measuring the value of a solution found by a heuristic technique, for looking inside a cost function, and for testing out ideas. Although one of the basic concepts of IMPACT was to provide a set of heuristics into which a user could simply insert his own cost function and thus arrive at some 'solution' without any exploitation of the cost function, the author was not averse to using any such knowledge. The extendability of IMPACT permitted the introduction of these special commands. They can also be made interruptable without undue difficulty.

The storage allocation commands allow IMPACT to be interrogated by the user in order to ascertain various features of the interactive session. Typically, the best solution found so far may be viewed (the RECAP command allows this); the permutation which the interaction is currently considering may be viewed and manipulated (BASE, POPCAP commands); permutations may be defined and displayed (GIAN, CATALOG commands); the CPU time used over an interval can also be determined (TIME command).

A variety of facilities are provided by the permutation manipulation commands. A single permutation specified by the user may be submitted to the cost function and the resulting cost displayed (HUNCH command). Elements in a particular permutation may be interchanged (INTCH) as may blocks of elements (MOVE). (The blocks need not be of the same size). Parts of permutations may be reversed (REV) or cycled (CYCLE) and specific elements in the permutations may be tried in other positions (WEAVE, TONFRO). In each of the above after the manipulation has been performed the resulting permutation is subjected to the cost function, and the user is informed as to the cost of the new permutation. At any stage in an interactive session the user will be informed if there has been any improvement in the best value found so far. Random permutations may be generated (SHUFFLE), as may permutations that are part random in the sense that certain elements of a base permutation are not to be disturbed from their original position (RANFIX). Various heuristic techniques based upon methods described in Chapter 3 are also provided. Some of these are based upon sorting techniques (SELECT, EXCHANGE, MERGE) whilst others adopt a Monte Carlo

approach (CMC, DISPER). Methods which permit complete enumeration are also included. One (PERLEX) allows the generation in a lexicographic manner of all permutations of 1 to n, or restricts this generation to part of an original permutation. Others allow the use of branch-and-bound approaches with different methods of performing the branching. One method of branching utilises a straightforward backtracking method of the kind described in Chapter 2 and its command word is TRAKBAK. The other permits an approach developed interactively and is described more fully in Chapter 6. The branch-and-bound techniques require, of course, that the user provide suitable routines for calculating lower bounds. Highly developed interactive facilities are also provided in these approaches and such facilities are described in 'Interactive Techniques' in Chapter 5.

4.6 Aids to Ease of use of IMPACT

IMPACT provides powerful facilities which allow users to manipulate permutations in order to resubmit them to heuristic algorithms. It is thus essential for techniques to have a result that is accessible to the whole system. It is also unreasonable, slow and very difficult for a user to enter complete permutations of 1 to n (for n anything other than small). Both of these problems are overcome by the provision of a facility for giving permutations names, in effect by defining a permutation.

4.61 Permutation Definition and the concept of the Current Active Permutation (CAP).

Permutations or part permutations may be defined either explicitly or implicitly. Explicit definition is achieved by the use of the command GIAN ('Give It A Name') and at any later stage a named permutation may be retrieved for use as input parameters to a command. Certain names are predefined and may be referred to by the user but may not be destroyed (See 'Predefined Permutations').

To define permutations implicitly the concept of a current active permutation is used. This is essentially a storage mechanism for permutations which the user may use and retrieve without the necessity of having to define them explicitly. As an illustration, the user may not know whether or not a permutation will be worth storing until after it has been submitted to the cost function. If he should then decide to retain the permutation it might be tedious to reconstruct it. A certain number (an IMPACT parameter at generation time, and currently three) of permutations are stored in a push-down stack manner and may be referred to by CAP 1, CAP 2, . . . etc. (CAP 1 being the most recently implicitly defined permutation, CAP 2 the one before that, and so on). Whenever a new implicitly defined permutation enters the stack the permutation which was CAP 1 becomes CAP 2, the old CAP 2 becoming CAP 3 and so on. (The permutation at the lowest level is discarded if necessary). Also manipulated with this stack of permutations is a stack with their associated cost. These costs may be referred to as CC1, CC2 . . . etc., in the same

manner as above.

Most of the commands available have as result a permutation and cost which become CAP 1 and CC 1 respectively, thus causing stack manipulation. (Detailed descriptions, about these commands, are given in 'Command Descriptions').

If after the use of a command which created a new member of the CAP family, the user wishes to revert to the situation (as far as CAP 1 is concerned) immediately preceding the issuing of the command he may do so by use of the command POPCAP. There is, of course, no means of restoring permutations which had to be discarded (because of the push-down nature of the stack) so the situation retrieved may be different in this respect.

4.62 Predefined Permutations

Certain names have been preassigned within IMPACT so that the user may easily form and manipulate permutations. (They may not be overridden). These names are BEST, BC, NATU, NTO1, PADD and the generic names CAP k and CCk (where k represents a digit in the range 1 to 9.)

The generic names have previously been explained. By referring to BEST in a string of parameters to a command the user will obtain the permutation corresponding to the lowest cost found so far; BC refers to the cost of this solution. NATU will result in the natural permutation (1, 2, . . ., n) being supplied and NTO1 will supply the reverse of NATU, i.e. (n, n-1, . . ., 2, 1). PADD is used to indicate that the parameters supplied to date are to be padded out with those elements of 1 to n which have not yet

been entered. The ordering of the elements which provide the padding is the natural order. As an example, if n = 8 the parameters 6,4,8,PADD; are equivalent to 6,4,8,1,2,3,5,7;. The parameters PADD; are equivalent to NATU;

4.63 Timing Considerations

IMPACT was designed to perform under a multi-programmed time-sharing operating system and experiments described later were carried out under MTS. It was found that the CPU time recorded by the system for various operations varied widely according to the loading of the system, (amount of drum-to-core transfers ('paging') and the number of disk accesses.)However, within MTS the use of an appropriate supervisor call in an Assembler Language subroutine provided access to two CPU times for a job within the system. These were the time spent in supervisor state (taking care of interrupts and general housekeeping) and the time spent in problem state (a more useful measure of time spent doing useful work for the program). The time spent in supervisor state was not surprisingly found to reflect the loading of the system (and fluctuate widely) whilst the time spent in problem state was reasonably consistent for the same operations (within ± 5%). When IMPACT was run overnight on the 360/67 the loading upon the system was low because of the low level of multi-programming which was taking place at that time and it was found that the time spent in supervisor state by IMPACT was negligible. Accordingly the problem state time was recorded for timing measurements, and figures quoting times are measurements of problem state time only. (The problem state time to load IMPACT is approximately six seconds).

4.64. Hard Copy Permanent Record Facility.

At the end of a problem-solving session a user may have no permanent record of the progress he made upon a particular problem. (He may, for example, have been using IMPACT from a character display). If he used a typewriter terminal his terminal sheet will provide a record of some form but is likely to contain typing errors and possibly terminal hardware errors. A more permanent machine-readable record of the session is kept by IMPACT and recorded onto a disk file. This record may be used for a variety of purposes. One suggestion put forward by Morton (18) is that scrutiny of it might prove insight into the problem-solving approaches used by humans and thus suggest more powerful heuristic techniques. The COMMENT command which allows the user to explain his decision making process also proves useful in this respect. Judicious use of the attention button also allows one to ask for information which might be too voluminous for terminal output but which will be recorded onto the disk file and hence be available for a more suitable output device, e.g. a line printer, character display, or graph plotter.

Chapter 5.

Interactive Techniques

The two previous chapters have described certain permutation manipulation techniques and heuristics. In order to allow convenient interaction in tackling permutation problems it was recognised that the somewhat simple techniques described required further extension to provide the user with more power. This chapter describes some of the extensions provided and their implementation where appropriate. The division of this chapter from chapter 3 is not clearly defined since the provision of some primitive facilities was suggested by the requirements of the author as he investigated some permutation problems. (Additional information about certain of the commands may be gleaned from 'User's Guide to IMPACT', Appendix 1).

5.1 The Motivation for the Display of Lower Bounds

The preliminary investigation into the behaviour of the branch-and-bound algorithm when applied to job - shop scheduling made the author apprehensive about using backtracking upon large ($n > 12$) problems. Initial attempts upon such problems therefore used the more primitive techniques provided by the heuristic approaches and some good results were obtained for some problems. However, the author was not able to assess whether such results were obtained by good fortune or by some particular decision mechanism invoked by intuition. A weakness also existed in the inability to specify exactly how good were the

particular solutions obtained. Furthermore, the heuristics employed provided local optima (in some sense) and there was no indication as to whether a search ought to be made in the neighbourhood of a permutation which differed greatly from that of the permutation produced by the heuristics. It was realised that access to the lower bounds associated with a part permutation was desirable and accordingly an appropriate command 'BOUNDS' was implemented. Upon entering 'BOUNDS' and providing a part permutation $P_j = (i_1, i_2, \ldots, i_j)$ to IMPACT the lower bounds associated with the part permutations $(i_1, i_2, \ldots, i_j, k)$, where $k \in \{I_n \sim P_j\}$ are displayed. In order to allow the user to easily assimilate the information provided, the bounds are sorted and printed in ascending order. If the user decides that he has seen enough of such information the remainder may be suppressed by the depression of the ATTENTION button. Figure 11 illustrates the use of the command.

```
## ## ##
bounds, 7, 3;
(   2  -      142)
(   1  -      142)
(   5  -      145)
(   6  -      147)
(   4  -      149)
(   8  -      151)  Lower bounds for part permutations beginning
                    with (7, 3)


## ## ##
bounds
   ENTER A SUBPERMUTATION
7,3;
(   2  -      142)  Same lower bounds but attention button
(   1  -      142)  pressed here
(   5  -      145)<
REST OF THE BOUNDS SUPPRESSED BY ATTN
```

Figure 11: Use of the 'BOUNDS' Command

5.2 User Controls over Backtracking

A measure of goodness of an approximate solution can be obtained from data available from a backtracking approach. An indication of the potentially fruitful regions for the search is also provided. The backtracking approach in IMPACT has been modified for the above reasons. The command 'TRAKBAK' allows the user to specify a part permutation upon which backtracking is to be performed, thus allowing the user to restrict the amount of computation expended and also providing a facility for following up ideas or 'hunches'. The maximum number of vertices to be examined in such a search must also be specified by the user; this provides some control over the tendency for backtracking to require large amounts of computing time. Storage space has had to be allocated for backtracking information and currently trees or subtrees of up to 35 levels may be searched. (The lack of dynamic storage allocation in the FORTRAN language is shown as a disadvantage here). Further controls are discussed in 'Tree Interrogation and Adjustment' from which it is obvious that those interactive facilities that were most used by the author are the most highly developed.

5.3 Further Developments of Existing Heuristics

Certain heuristics have been developed further than their description in earlier chapters. As an example, in the use of the exchanging technique the original specification meant that the complete permutation had to be subjected to pairwise interchanges. Frequently in investigations using this technique one wished to have $P_n = (i_1, i_2, \ldots, i_{n-1}, i_n)$ as a starting permutation and additional knowledge would show that in order to better the best

cost so far found the elements i_{n-1} and i_n should not be disturbed in the original permutation. However, the exchanging mechanism might at times recognise the cost of the permutation $(i_1, i_2, \ldots, i_{n-1}, i_n)$ to be greater than that of the permutation $(i_1, i_2, \ldots, i_n, i_{n-1})$ and would consequently interchange i_n and i_{n-1}. To allow the user to instruct IMPACT not to do this additional parameters to the exchange command are provided and when these are utilised exchanging is only performed upon part of the permutation specified. (The costs are of course calculated upon the complete permutation). Figure 12 illustrates the difference between exchanging upon a full permutation and part of a permutation.

```
## ## ##
time
   219158          121119          340277          0
```

Initialising call of 'TIME'

```
## ## ##
exchange,8,7,6,5,4,3,2,1;)
 NEW TARGET FOUND         } 'exchanging' from (8,7,6,5,4,3,2,1).
  EXCHANGING GIVES COST   128
PERMUTATION
  8   7   6   5   4   2   3   1
```

Number of calls of the cost function

```
## ## ##
time
          93              84              177             10
```

Cpu time in problem state in milliseconds

```
## ## ##
newstate
  TARGET VALUE RESET TO 99999
```

Resetting target value

```
## ## ##
time
          55              53              108             0
```

```
## ## ##
exchange,8,7,6,5,4,3,2,1,7,3;
 NEW TARGET FOUND
  EXCHANGING GIVES COST   131
 PERMUTATION
   8   7   5   6   3   4   2   1
```

Specifying that exchanging is only to be performed between the positions of elements 7 and 3

```
## ## ##
time
          93              96              189             8
```

Only 8 calls of the cost function this time.

Figure 12: Differences in the use of the EXCHANGE command

Some of the heuristics employed may also be modified by the user interpreting a permutation in a slightly different manner. This point is discussed further in 7.0.

5.4 Tree Interrogation and Adjustment

5.41 Interruption of a Search

During the course of backtracking through the tree of permutations, situations may arise where the computational effort of the search is directed in a part of the tree which might give a small decrease in the lowest cost found so far. The potential payoff from a successful search (which might involve a considerable amount of computation) in this region may be small and the computational effort might be better employed in a different part of the tree. Facilities have therefore been provided for allowing one to interrupt the backtracking procedure and examine those still potentially fruitful parts of the tree. The tree can be adjusted if desired and the search resumed as before or even abandoned.

Whenever the backtracking is interrupted by a depression of the attention button the number of vertices (of the tree) that have been examined is displayed, providing an indication of the progress of the search to date. Also displayed is the number of complete permutations that have been examined, giving an indication of whether the computational effort has been expended in a low (near to the leaves) or a high (near the root) part of the tree. Local commands provide the ability to display the tree remaining for investigation. 'HOWF' informs the users about the depth of this tree (the number of allocated elements in the permutation) and also about this part permutation.

As an example, IMPACT might respond to an attention with

```
YOU'VE INTERRUPTED THE BACKTRACKING AFTER     39 VERTICES AND
              4 COMPLETE PERMUTATIONS
?
```

entering 'HOWF' might then result in

```
howf
 THE DEPTH OF THE SEARCH IS    5
 AND THE PERMUTATION BEING EXPLORED IS    5   2   8   6   7
  WITH LOWER BOUND =     117
?
```

The stack representation of the tree can be displayed by inputting 'STAC'; only those vertices that are still active, i.e. have lower bounds less than the current targetvalue, are displayed

```
?
stac
    ID      BACK      BOUND      LEVEL
    24         3        119          5
    22         1        116          4
    21         7        117          4
    20         3        118          4
    18         6        119          3
    13         1        114          2
    12         4        114          2
    11         6        114          2
    10         8        115          2
     9         7        117          2
     7         7        117          1
?
```

Table 6: The Stack representation of a tree.

The above information is interpreted in the following manner: the permutation beginning with (5,2,8,6,3) is in position number 24 in the stack and has a lower bound of 119, whilst that beginning with (5,4) is in position number 12 and has a lower bound of 114. The state of the search is thus depicted by figure 13. Parts of the tree that are still active are represented by nodes and their lower bounds are attached to them.

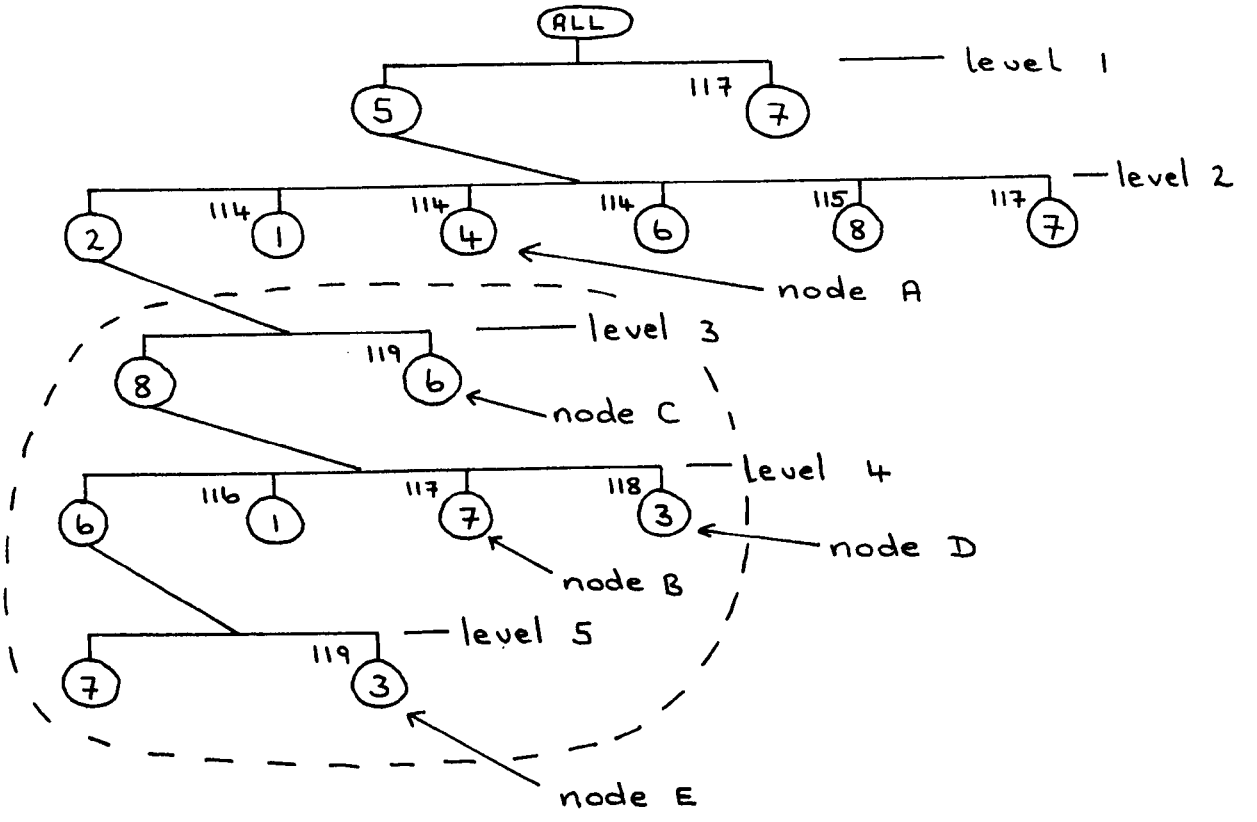


Figure 13: The tree corresponding to Table 6.

5.42 Tree Drawing.

The same information that is printed by the commands HOWF and STAC may be displayed more graphically by use of the local command DRAW. This command will 'draw' the tree associated with the information. Figure 14 shows the tree with the root printed first, agreeing with the orientation as was introduced in chapter 2.

```
?
draw down
          117
1( 5)---( 7)
   .
   .
   .      114    114    114    115    117
2( 2)---( 1)---( 4)---( 6)---( 8)---( 7)
   .
   .
   .      119
3( 8)---( 6)
   .
   .
   .      116    117    118
4( 6)---( 1)---( 7)---( 3)
   .
   .
   .      119
5( 7)---( 3)
   .
   .
   .
  117   ← Lower bound of part permutation being investigated
↑
level
```

Figure 14: Tree drawn conventionally.

A more convenient manner of printing, from an interactive problem solving viewpoint, is illustrated by figure 15.

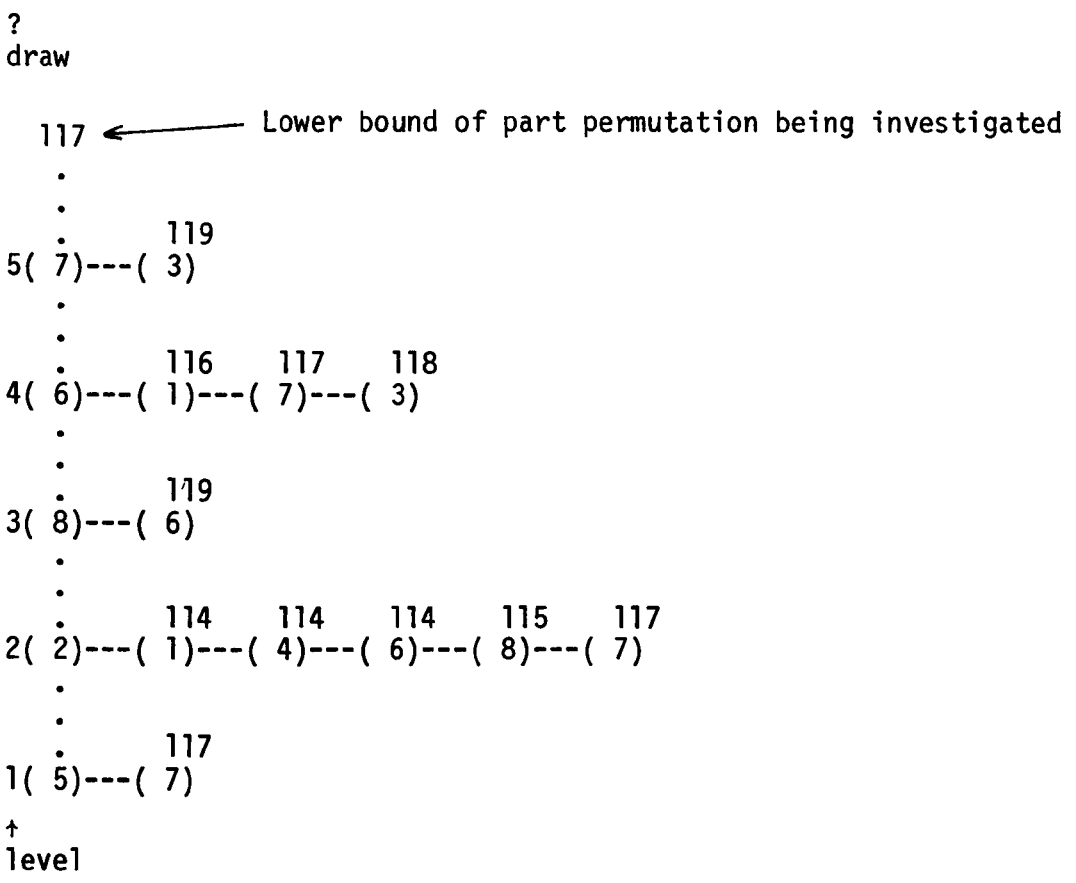


Figure 15: Tree drawn root last

Nodes lower down the tree are displayed first, and thus the part of the tree that will be examined soonest is seen first by the user. A parameter to the DRAW command gives the user the choice as to which way the tree is to be displayed. Use of the ATTENTION button during printing of the tree will terminate the printing although a drawing of the tree will be stored away automatically onto the recording file and will not be suppressed. Thus if one wishes only to store a drawing for later viewing the DRAW command can be issued and followed immediately by a press of the ATTENTION button.

5.43 Masking of Parts of a Tree

Parts of the tree may be discarded or pruned in various fashions. A simple method is to use the 'MASK' command in order to specify particular nodes to be rejected. Implementation is performed by altering the lower bound associated with the node to be masked to a value higher than the best cost found so far. Thus to discard the nodes labelled A and B in figure 13 one should specify that positions 12 and 21 in the stack are to be masked. The tree would then be as in figure 16. IMPACT makes no record of parts of the tree which have been discarded in this manner and thus it is the user's responsibility to remember should he require later examination of any nodes which have been masked.

```
?
mask
 ENTER IDS OF VERTICES TO MASK OUT OF THE SEARCH
12,21;
 MASK DONE
?
draw
```

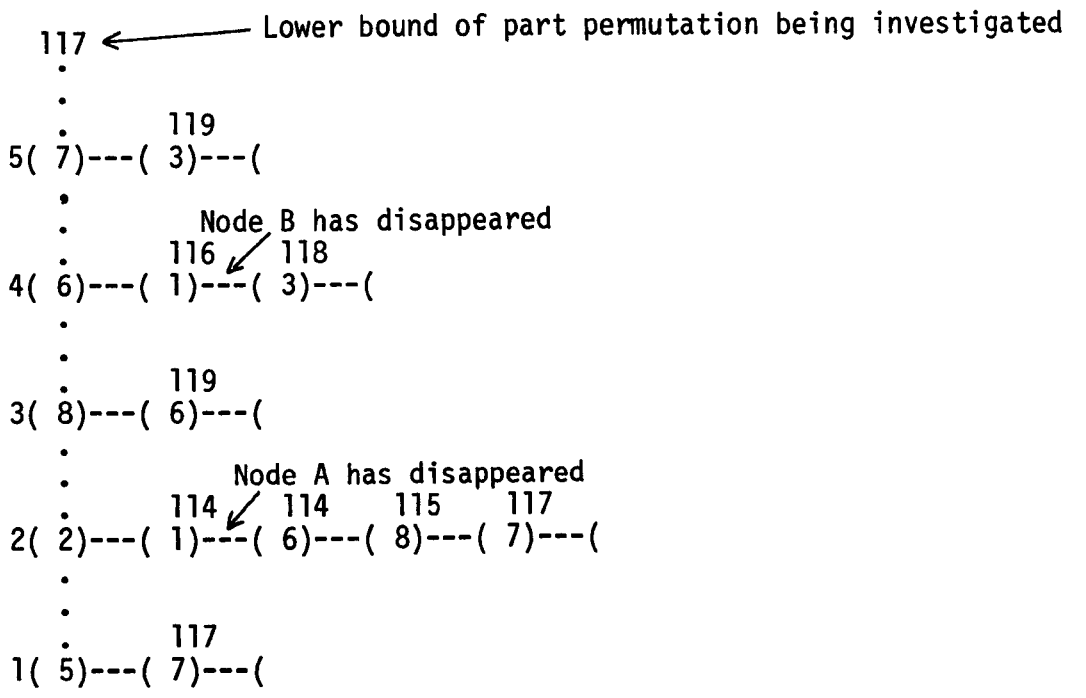


Figure 16: Tree after 'MASK'ing of nodes A and B

5.44 Alteration of the Targetvalue

A more powerful method of reducing the amount of computation in a search is to use the local command 'TARG'. This allows the target-value (the value with which the lower bounds are compared) to be altered to a value chosen by the user. If at some stage the cost of the best solution found so far is C_b then altering the target-value to $C_t = C_b - \delta$ will cause only those nodes with a lower bound less than C_t to be examined. δ would normally be positive (but need not be so) and thus fewer nodes of the tree would be examined. It does not follow that no permutations giving a cost between C_t and C_b will be found since a part permutation of n-2 elements may have a lower bound less than C_t and thus warrant investigation. The costs of the two resulting permutations may be such that the smaller of them lies between C_t and C_b and it would be foolish to ignore it.

In the example if the best cost discovered so far is 120 then using TARG to search for values better than 118 would result in nodes C, D and E in figure 13 being thrown away, leaving figure 17. Implementation is performed by adjusting the value with which the lower bounds are compared rather than by altering values in the stack. Thus after a time if the manoeuvre appears to be fruitless in the sense that no improvement in C_b is made the attention button may be used and TARG reset. A certain amount of the information can thus be retrieved.

```
?

targ

 ENTER NEW TARGET VALUE)
118;                    }   User Command and IMPACT's confirmation
  TARGETVALUE ALTERED FROM      120  to   118


?
draw

  117
   .
   .
   .        Node E has disappeared
5( 7)---(<--
   .
   .          Node D has disappeared
   .       116    117
4( 6)---( 1)---( 7)---(<--
   .
   .
   .
3( 8)---( <----------Node C has disappeared
   .
   .
   .       114    114    114    115    117
2( 2)---( 1)---( 4)---( 6)---( 8)---( 7)---(
   .
   .
   .       117
1( 5)---( 7)---(
```

Figure 17. The effect of altering the targetvalue

In practice the author found the TARG facility useful for making large jumps up the tree to levels where the potential payoff (with reference to the lower bounds) could be greater. The possibility of switching to another part of the tree where the payoff could also be small can thus be avoided.

The TARG command is also useful for limiting the amount of information displayed by the STAC or DRAW command. The stack of lower bounds is of the order of n^2 elements and the display of all of

these may be time-consuming or not required by the person supplying the interaction. One method of curtailing the display of such information is to suppress it by the use of the attention button. A more selective method is to use TARG and thus only display those nodes which have lower bounds less than the value specified. The targetvalue may of course be reset to its previous value should it be desired. Thus the user may see which parts of the tree he will be pruning before committing himself to such a course, and hence the possibility of being too severe may be avoided. Figure 18 is the original tree retrieved by resetting the targetvalue to its previous value.

```
?
targ                          )
 ENTER NEW TARGET VALUE       } ← User Command
120;                          )
  TARGET VALUE ALTERED FROM      118  TO     120
?
draw


  117
   .
   .
   .      119
5( 7)---( 3)---(
   .
   .
   .      116    117    118
4( 6)---( 1)---( 7)---( 3)---(
   .
   .
   .      119
3( 8)---( 6)---(
   .
   .
   .      114    114    114    115    117
2( 2)---( 1)---( 4)---( 6)---( 8)---( 7)---(
   .
   .
   .      117
1( 5)---( 7)---(
```

Figure 18: Original Tree retrieved by resetting the targetvalue to its previous value.

5.45 Jumping up the Tree

The two previous methods of tree pruning have the disadvantage that any nodes which are rejected because they are unlikely to yield a sufficiently high payoff cannot be retrieved easily within IMPACT. An alternative means of curtailing a search is available in the 'JUMP' command and does not have the above drawback. Use of the command allows the search to jump to a higher level of the tree, disregarding (either temporarily

or permanently) the subtree jumped over. On invoking the command one is prompted whether information jumped over is to be saved or not. If it is to be saved then it must be named by the user and stored on a disk file. It then may be later retrieved by its name. In the example, if the search is to jump to level 3 then the part of the tree encircled in figure 13 would be jumped over and the next subtree examined would be based upon the part permutation (5,2,i) where i is the next element in the filial set of (5,2), i.e. 6.

Figure 19 illustrates such an effect. The subtree jumped over has been saved and named 'FRED'; it can be retrieved later by the 'RESUME' command (see appendix 1) and when retrieved will be as in figure 20.

```
?
jump
 ENTER LEVEL TO JUMP TO
3;                          User - IMPACT dialogue
 SHALL WE SAVE THE INFORMATION JUMPED OVER?
yes
 ENTER A NAME TO DEFINE IT
fred
  INFORMATION STORED AND NAMED FRED
?

draw
```

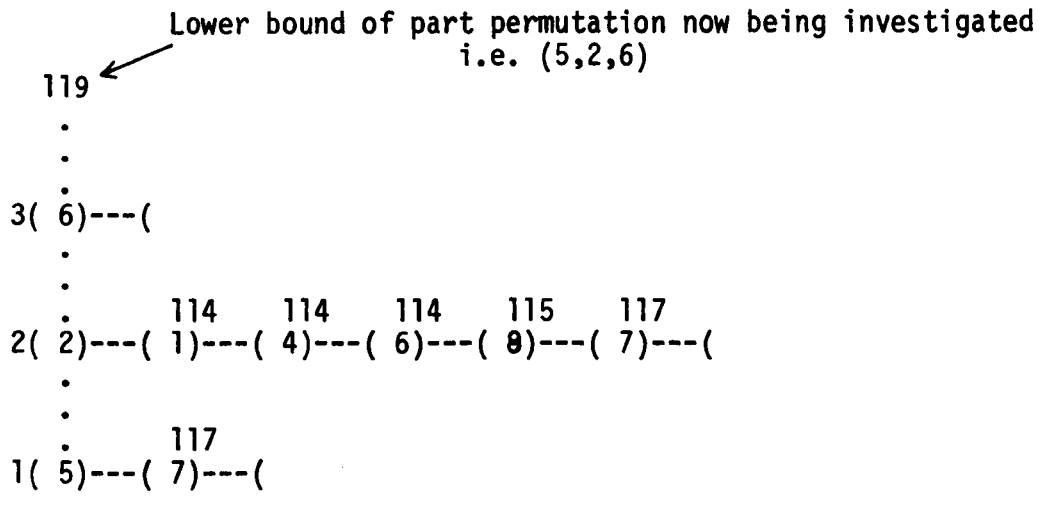


Figure 19: The effect of 'JUMP'ing up the tree.
Note: Only three levels now.

```
  ## ## ##
resume
 ENTER NAME OF THE TREE TO RETRIEVE
fred
 RETRIEVED TREE NAMED 'FRED'
   SEARCH RESTARTS WITH 1000 VERTICES AS INITIAL MAXIMUM
 YOU'VE INTERRUPTED THE BACKTRACKING AFTER       0 VERTICES AND
?                                                0 COMPLETE PERMUTATIONS
howf
 THE DEPTH OF THE SEARCH IS      5
 AND THE PERMUTATION BEING EXPLORED IS     5   2   8   6   7
  WITH LOWER BOUND=      119
?
draw


  119
   .
   .
   .      119
5( 7)---( 3)---(
   .
   .
   .      116    117    118
4( 6)---( 1)---( 7)---( 3)---(
   .
   .
   .
3( 8)---(
   .
   .
   .
2( 2)---(
   .
   .
   .
1( 5)---(
```

Figure 20: Retrieval of the tree previously 'JUMP'ed over

The JUMP command is implemented by adjusting the pointers for the level of the search and for the information stack. Information temporarily discarded can be saved in a compact form. The jump from level 5 to level 3 means that the part of the tree discarded can be specified by (5,2,8,6,7), the part permutation that was being examined, and the branches of the original tree that lay between the two levels concerned, i.e. (level 4, node 1 with

bound 116, node 7 with bound 117, node 3 with bound 118) and (level 5, node 3 with bound 119). Table 7 shows the information stored on the disk file

```
                Depth of the tree
## $list  -q \          Name of tree
>          1            ↓ FILE FOR SAVING DUMPED TREES
>          3     SAV FRED
>          4      ---->5   0 <---------- Last element (explained in 6.21)
>          5        5   2   8   6   7 <---------- Part permutation
>          6                3       118   4 )
>          7                7       117   4 } Stack of information
>          8                1       116   4 }
>          9                3       119   5 )
## END OF FILE
```

Table 7: Information saved on 'JUMP'ing up the tree

The ability to jump and save the part of the tree jumped over allows the user to make the search more like branching from the lowest bound without losing all of the attractions of back-tracking.

Chapter 6.

Interaction and the Job-Shop Scheduling Problem

Although IMPACT was designed to permit interaction to achieve good results without intimate knowledge of the particular cost function being investigated, simple interactive sessions showed that any extra knowledge could greatly improve the quality of the results obtained and consequently in the investigation of the job-shop scheduling problem existing knowledge of the problem was utilised. This knowledge proved extremely useful and severe weaknesses in existing enumerative approaches were highlighted and to a certain extent removed.

Early in the interactive investigation of the job-shop scheduling problem it was realised that the inputting of a permutation to the cost function and the display of the resulting cost did not provide great insight into what was happening inside the cost function. However, the display of earliest finishing times mentioned in Chapter 1 would reveal more of the behaviour of the cost function. Upon modifying IMPACT to provide this facility, use of the facility revealed that what was proving useful to the user was a display of the bottlenecks in the processing of the jobs through the machines. In effect, the critical path in the processing was required.

6.1 The Critical Path for a sequence of jobs

Given a job-shop problem with n jobs and m machines, any permutation of the n jobs gives a duration or cost which can be calculated according to the rule given earlier. In the evaluation of this duration n + m - 1 job-machine times are summed. Thus there are

at least n + m - 1 job-machine pairs which form a critical path in the sense that an increase of k units ($k \geq 0$) in any one of these times would mean an increase in the duration of the schedule of k units. (The figure n + m - 1 is arrived at from the fact that any duration is the sum of n entries from the m columns and m entries from the n rows less one since one entry is common to both (see fig. 2)).

As an example, in the 5/3 problem in the table 1 the table of earliest finishing times in figure 2 is

	Machine		
Job	I	II	III
3	(3)	34	54
2	(16)	46	72
4	(43)	70	84
5	(94)	(106)	(124)
1	95	118	(143)

Figure 21. A critical path

The times circled are the ones for the job-machine pairs which form the critical path. (They may be found by simply tracing back the arrows in figure 2).

It does not follow that a decrease of k units in any of the times which are critical will mean a decrease of k units in the duration since other elements in the matrix may then become critical. e.g. if d(5,3) were decreased by 7 units to 11 units the above table would change to

		Machine I	II	III
	3	3	34	54
Job	2	16	46	72
	4	43	70	84
	5	94	106	117
	1	95	118	137

Figure 22: the effect of decreasing d(5,3) by 7 units

and thus the duration and critical path have changed. Decreasing an element on the critical path by k units can decrease the duration of the schedule by a maximum of k units.

This leads to the concept of slack. For a particular permutation we define the slack associated with an element (a job-machine pair) as being the maximum amount by which the value of the element may be increased without increasing the duration associated with the permutation. (The slack may also be thought of as the maximum amount by which the processing of the job on the machine may be delayed without affecting the completion time of the whole process). Thus any element on the critical path has zero slack. Conversely any element with zero slack must lie on the critical path.

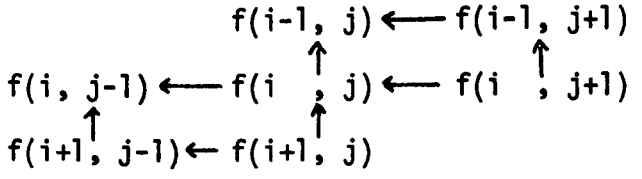


Figure 23: Algebraic display of earliest finishing times

Consider the earliest finishing times depicted in the above figure. Suppose that the element (i, j+1) lies upon the critical path.

Since $f(i, j+1) = \max \left[f(i,j), f(i-1,j+1)\right] + D(i,j+1)$

if the value of D(i,j) is increased by an amount a then the value of f(i,j) will be increased by an amount a and the value of f(i, j+1) will not be increased providing that

$$f(i-1, j+1) \geq f(i,j) + a$$

i.e. providing that the critical path still passes through (i-1, j+1).

It follows therefore that if the critical path passes through the job-machine pair (i, j+1) the maximum amount that element D(i,j) can be increased is

$f(i - 1, j+1) - f(i, j)$ if this value is positive, otherwise 0.

i.e. if we denote the slack associated with element (i, j) as S(i, j) then

$$S(i,j) \leq \max \left[f(i-1, j+1), f(i,j)\right] - f(i,j) \qquad \text{-- (1)}$$

However increasing the value of D(i,j) may affect the critical path in another way. If the critical path passed through element (i+1, j) then a similar argument to the above gives

$$S(i,j) \leq \max \left[f(i,j), f(i+1, j-1)\right] - f(i,j) \qquad \text{-- (2)}$$

In the first case above, if the job-machine pair (i, j+1) does not lie on the critical path then

$S(i, j+1) = b$ where $b > 0$

By increasing the values of f(i,j) and f(i-1, j+1) by b the value f(i, j+1) will be increased by b and hence will be brought onto the critical path.

Thus (1) can be rewritten as

$$S(i,j) \leq \max \left[f(i,j), f(i-1, j+1)\right] - f(i,j) + S(i, j+1) \qquad \text{-- (3)}$$

and similar considerations for the element (i+1,j) give

$$S(i,j) \leq \max \left[f(i,j),\ f(i+1,\ j-1) \right] - f(i,j) + S(i+1,\ j) \quad -- \quad (4)$$

Combining (3) and (4) gives the result that

d) $$S(i,j) = \min \left\{ \max \left[f(i,j),\ f(i+1,\ j-1) \right] + S(i+1,\ j), \right.$$
$$\left. \max \left[f(i,j),\ f(i-1,\ j+1) \right] + S(i,\ j+1) \right\} - f(i,j)$$

This gives an algorithm for calculating the slack associated with a particular job-machine pair under a particular permutation since the above holds for $1 < i < n$.

Special cases are:

a) $i = 1,\ j = 1$ and $i = n,\ j = m$. $S(1,1) = 0$ and $S(n,m) = 0$ since the first event and the last event lie upon the critical path.

b) $j = m,\ 1 < i < n$. In this case d above breaks down to

$$S(i,m) = \max \left[f(i,m),\ f(i+1,\ m-1) \right] - f(i,m) + S(i+1,\ m)$$

c) $1 \leq j < m,\ i = n$

$$S(n,\ j) = \max \left[f(n,j),\ f(n-1,\ j+1) \right] - f(n,j) + S(n,\ j+1)$$

e) $j = 1,\ 1 < i < n$

$$S(i,1) = \min \left\{ \max \left[f(i,1),\ f(i-1,\ 2) \right] - f(i,1) + S(i,2), \right.$$
$$\left. S(i+1,\ 1) \right\}$$

f) $i = 1,\ j = m$

$$S(1,\ m) = \max \left[f(1,m),\ f(2,m-1) \right] - f(1,m) + S(2,m)$$

g) $i = 1,\ 1 < j < m$

$$S(1,j) = \min\left\{ \max\left[f(1,j),\ f(2,j-1) \right] - f(1,j) + S(2,j),\ S(1,\ j+1) \right\}$$

Thus the slack can be calculated and a suitable order of computation is a,b,c,d,e,f,g which is depicted below.

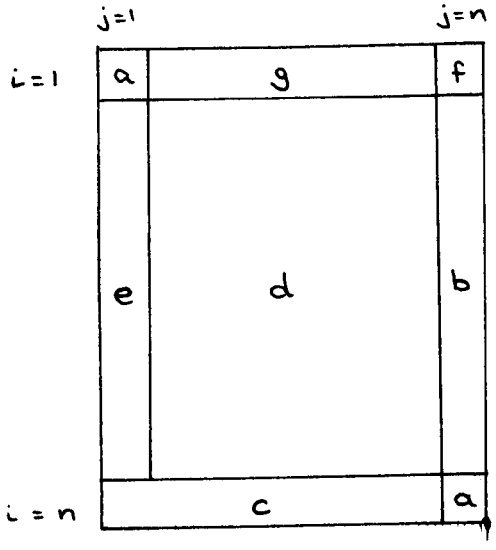


Figure 24:
An order for calculation of slack

Example: In the problem given in the 1.11 the earliest finishing times for the permutation (3,2,4,5,1), were as in the table below.

		j = 1	2	3
p_i	i			
3	1	3	34	54
2	2	16	46	72
4	3	43	70	84
5	4	94	106	124
1	5	95	118	143

table 8: The earliest finishing times of 1.11

$F = \{ f(i,j) \}$

In the above table element f(i,j) denotes the earliest finishing time of element p_i

on $p_n = \{p_1, p_2, \ldots, p_n\}$ upon machine j.

Now S(1,1) = 0 and S(5,3) = 0.

From c,

$$S(5,2) = \max \left[f(5,2),\ f(4,3) \right] - f(5,2) + S(5,3)$$

$$= \max\ (118,\ 124) - 118 + 0 = 6$$

$$S(5,1) = \max \left[f(5,1),\ f(4,2) \right] - f(5,1) + S(5,2)$$

$$= \max\ (95,\ 106) - 95 + 6 = 17.$$

From b,

$$S(4,3) = \max \left[f(4,3),\ f(5,2) \right] - f(4,3) + S(5,3)$$

$$= \max\ (124,\ 118) - 124 + 0 = 0$$

$$S(3,3) = \max \left[f(3,3),\ f(4,2) \right] - f(3,3) + S(4,3)$$

$$= \max\ (84,\ 106) - 84 + 0 = 22$$

$$S(2,3) = \max \left[f(2,3),\ f(3,2) \right] - f(2,3) + S(3,3)$$

$$= \max\ (72,\ 70) - 72 + 22 = 22$$

$$S(1,3) = \max \left[f(1,3),\ f(2,2) \right] - f(1,3) + S(2,3)$$

$$= \max\ (54,\ 46) - 54 + 22 = 22$$

From d,

$$S(4,2) = \min \left\{ \max \left[f(4,2), f(5,1) \right] + S(5,2),\ \max \left[f(4,2), f(3,3) \right] + S(4,3) \right\} - f(4,2)$$

$$= \min \left\{ \max\ (106,95) + 6,\ \max\ (106,84) + 0 \right\} - 106 = 0$$

$$S(3,2) = \min \left\{ \max \left[f(3,2), f(4,1) \right] + S(4,2),\ \max \left[f(3,2), f(2,3) \right] + S(3,3) \right\} - f(3,2)$$

$$= \min \left\{ \max\ (70,94) + 0,\ \max\ (70,72) + 22 \right\} - 70 = 24$$

$$S(2,2) = \min \left\{ \max \left[f(2,2),\ f(3,1) \right] + S(3,2),\ \max \left[f(2,2), f(1,3) \right] + S(2,3) \right\} - f(2,2)$$

$$= \min \left\{ \max\ (46,43) + 24,\ (46,54) + 22 \right\} - 46 = 24$$

From e,

$$S(4,1) = \min \left\{ \max \left[f(4,1), f(3,2) \right] - f(4,1) + S(4,2), S(5,1) \right\}$$

$$= \min \left\{ \max\ (94,70) - 94 + 0,\ 17 \right\} = 0$$

$$S(3,1) = \min\{\max\left[f(3,1), f(2,2)\right] - f(3,1) + S(3,2), S(4,1)\}$$

$$= \min\{\max(43,46) - 43 + 24, 0\} = 0$$

$$S(2,1) = \min\{\max\left[f(2,1), f(1,2)\right] - f(2,1) + S(2,2), S(3,1)\}$$

$$= \min\{\max(16,34) - 16 + 24, 0\} = 0$$

From g

$$S(1,2) = \min\{\max\left[f(1,2), f(2,1)\right] - f(1,2) + S(2,2), S(1,3)\}$$

$$= \min\{\max(34,16) - 34 + 24, 22\} = 22$$

Thus the matrix S is

p_i	i	j = 1	2	3
3	1	0	22	22
2	2	0	24	22
4	3	0	24	22
5	4	0	0	0
1	5	17	6	0

Table 9: The slack associated with the job-machine times

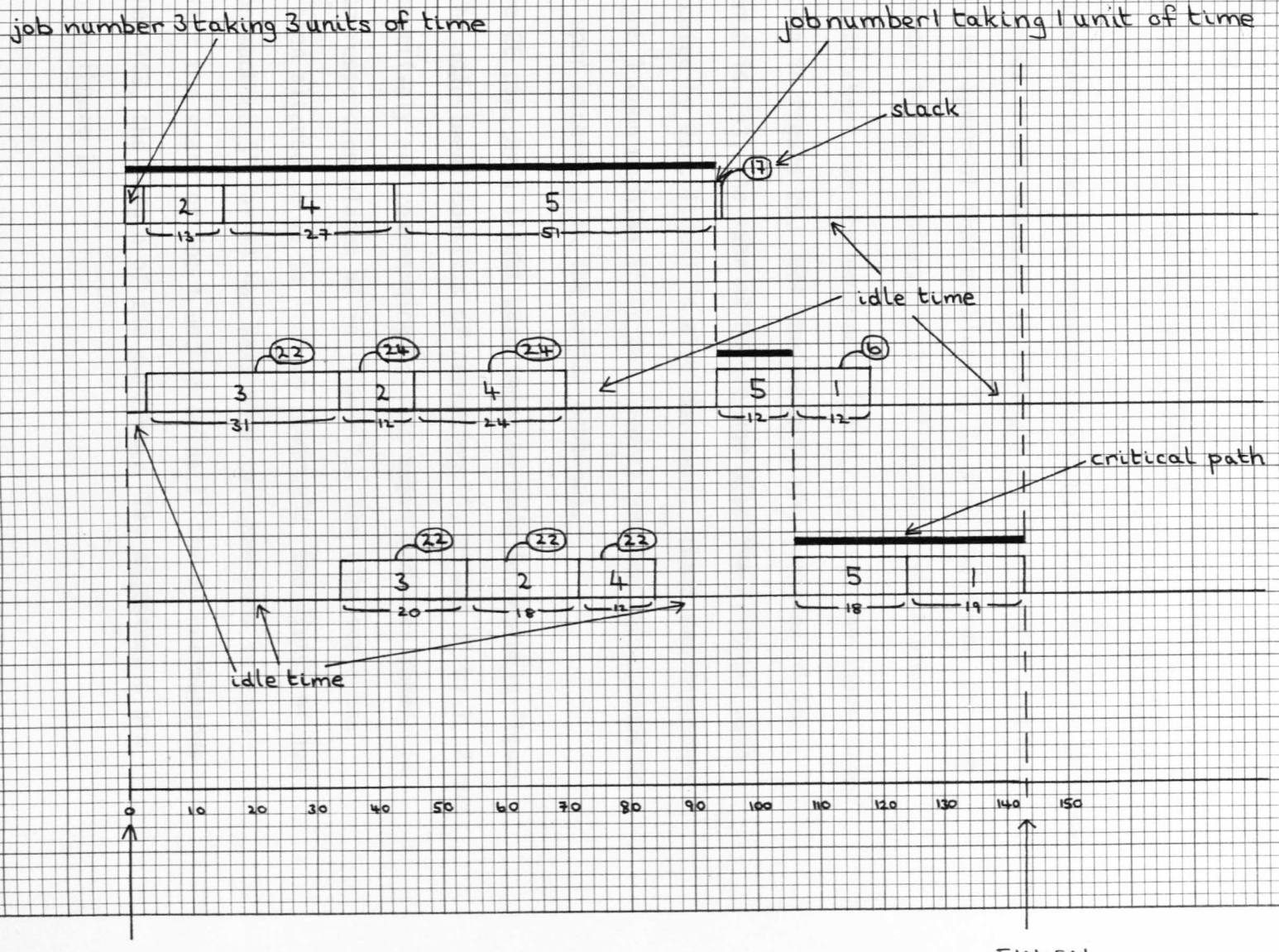
and the critical path can be easily observed. This table in conjunction with table 1 is equivalent to figure 25. Figure 25 is almost identical to figure 1 except that the critical job-machine elements have been marked by dark lines and the slack associated with each job-machine element is shown.

6.11 The inverted problem and its association with the slack of a job-machine element

The consideration of the inverted problem for the scheduling of n jobs upon m machines provides further insight into the concept of slack. The example given in Chapter 1 when converted to the inverted problem requires that 5 jobs are to be sequenced

Events Sequence for 5 jobs upon 3 machines with the critical path outlined.

Figure 25.



upon 3 machines 1', 2', 3' where the times taken by each job upon the machines are as in the table below.

Job	Machine		
	1'	2'	3'
1	19	12	1
2	18	12	13
3	20	31	3
4	12	24	27
5	18	12	51

Table 10: The inverted problem of table 1

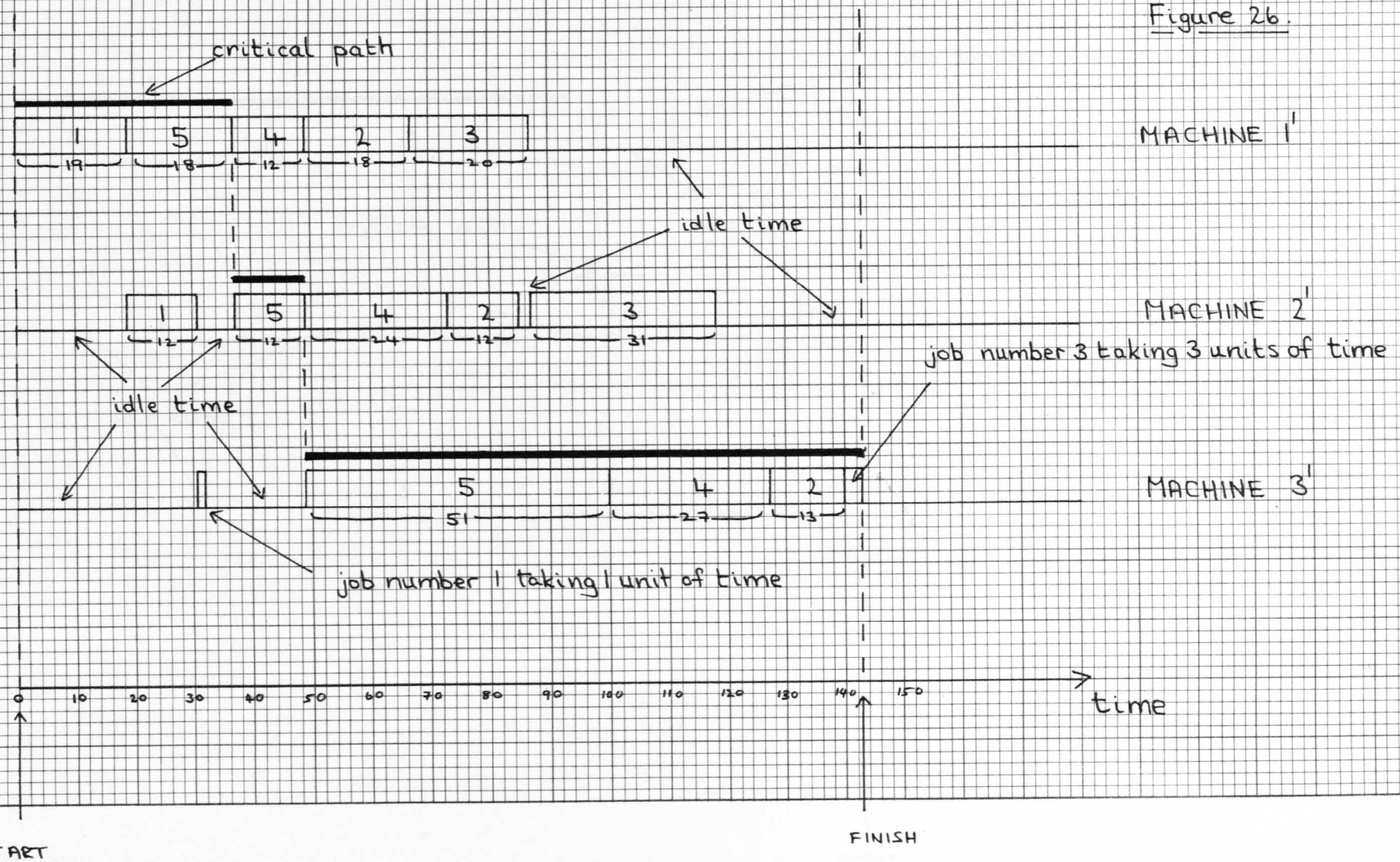
As explained in 2.41 the sequence (3,2,4,5,1) for the normal problem corresponds to the sequence (1,5,4,2,3) for the reverse problem. It is also apparent that machines 1, 2, 3 in the normal problem correspond to the machines 3', 2', 1' in the reverse problem.

The Gantt chart for the sequence (1,5,4,2,3) with the above times is given in figure 26. Not unexpectedly the duration of the sequence is 143, the same as before and also the critical path (indicated by dark lines) is also the same; (when one considers that job 1 upon machine 1' in the reverse problem corresponds to job 1 upon machine 3 in the normal problem).

Figure 26 can, by a slight adjustment be made to resemble figure 1. The adjustment consists of replacing 1', 2' and 3' by 3, 2 and 1 respectively and rearranging so that these machines appear in the corresponding places as in figure 1. The time scale has also been altered so as to 'run in reverse'; the finish of job 3 upon machine 3' (or 1) has been made to correspond to time 0 whilst the start of job 1 upon machine 1' (or 3) corresponds to time 143, the length of the duration. The adjustments are shown in figure 27.

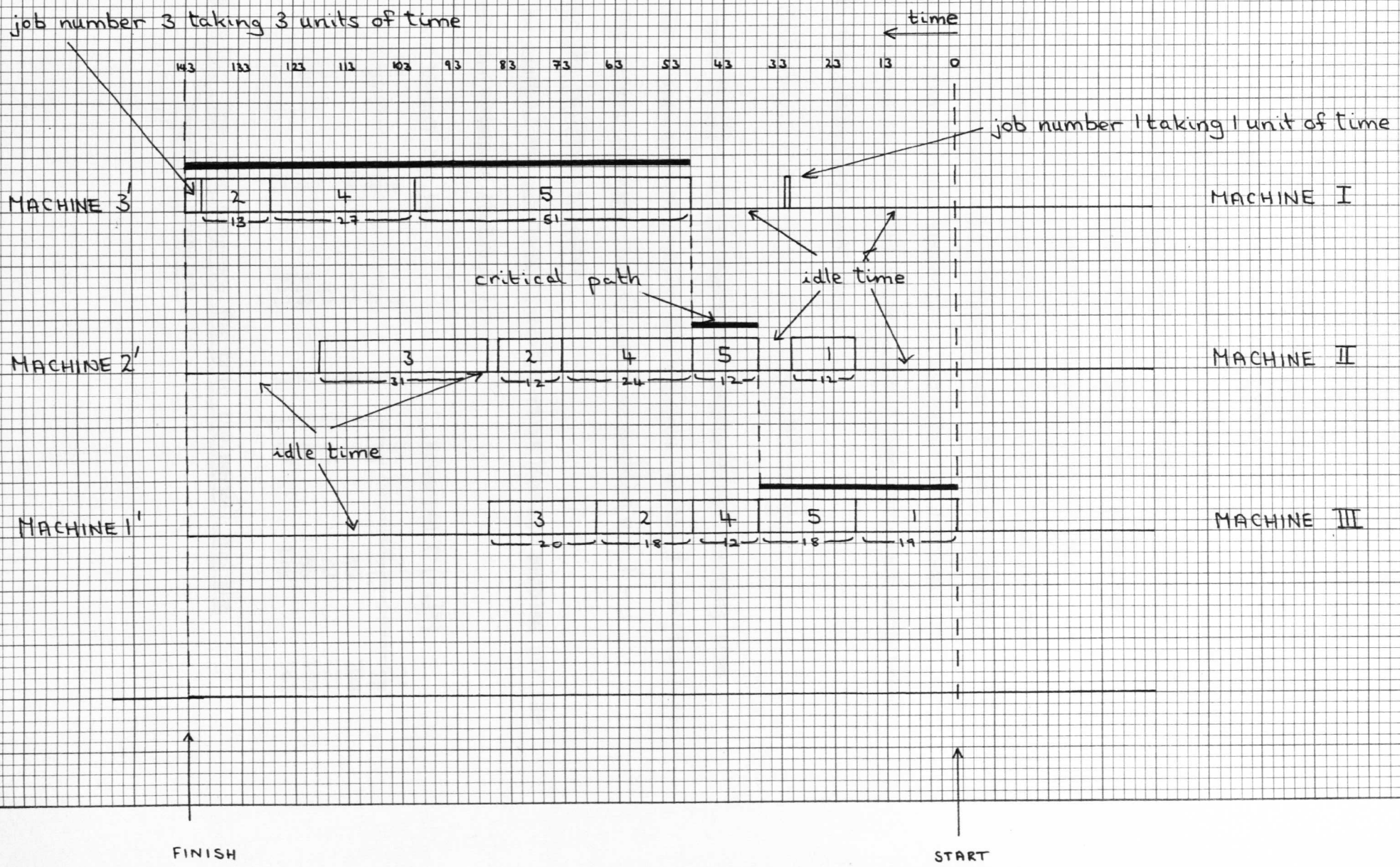
The Sequence of Events for the Reversed Problem of 5 jobs upon 3 machines.

Figure 26.



The Reversed Problem with the time scale adjusted.

Figure 27.



It should be noted that in the adjusted figure the critical path remains unaltered whereas the starting times of processing of jobs not upon the critical path do not remain unaltered. In effect these times are the latest that the jobs could commence without affecting the total duration of the sequence. Figure 25 gives the earliest starting times of such jobs and hence an alternative means of calculation of the slack can be seen. Suppose f(i,j) is the earliest finishing time of job i upon machine j under some sequence in the normal problem and that f'(i,j') is the earliest finishing time of job i on machine j' under the reversed sequence in the inverted problem. If the duration of the total sequence is T then the slack associated with the job-machine pair (i,j) is

$$\text{abs}\{(f(i,j) - d(i,j)) - (T - f'(i, m+1-j))\}.$$

The term f(i,j) - d(i,j) represents the earliest starting time of job i upon machine j under the sequence, whilst T - f'(i,m+1-j) is the latest starting time of job i on machine m + 1 - j under the reversed sequence.

As an example the earliest finishing times for our problem earlier under the sequence (3,2,4,5,1)

	Normal f(i,j)				Reversed f'(i,j')		
	1	2	3		1'	2'	3'
3	3	34	54	1	19	31	32
2	16	46	72	5	37	49	100
4	43	70	84	4	49	73	127
5	94	106	124	2	67	85	140
1	95	118	143	3	87	118	143

Table 11: The same sequence for both the normal and the reversed problems

and the earliest finishing times for the reverse problem under the sequence (1,5,4,2,3) are in table 11. The duration of both sequences is of course 143 units.

In our earlier notation for slack,

$$S(3,1) = \text{abs}\{f(3,1) - d(3,1) - (143 - f'(3,3)\}$$

$$= \text{abs}\{3 - 3 - (143 - 143)\} = 0$$

$$S(3,2) = \text{abs}\{f(3,2) - d(3,2) - (143 - f'(3,2'))\}$$

$$= \text{abs}\{34 - 31 - (143 - 118)\}$$

$$= \text{abs}(3 - 15) = \text{abs}(-22) = 22$$

$$S(3,3) = \text{abs}\{54 - 20 - (143 - 87)\} = 22$$

$$S(2,1) = \text{abs}\{16 - 13 - (143 - 140)\} = 0$$

$$S(2,2) = \text{abs}\{46 - 12 - (143 - 85)\} = 24$$

$$S(2,3) = \text{abs}\{72 - 18 - (143 - 67)\} = 22$$

$$S(4,1) = \text{abs}\{43 - 27 - (143 - 127)\} = 0$$

$$S(4,2) = \text{abs}\{70 - 24 - (143 - 73)\} = 24$$

$$S(4,3) = \text{abs}\{84 - 12 - (143 - 49)\} = 22$$

$$S(5,1) = \text{abs}\{94 - 51 - (143 - 100)\} = 0$$

$$S(5,2) = \text{abs}\{106 - 12 - (143 - 49)\} = 0$$

$$S(5,3) = \text{abs}\{124 - 18 - (143 - 37)\} = 0$$

$$S(1,1) = \text{abs}\{95 - 1 - (143 - 32)\} = 17$$

$$S(1,2) = \text{abs}\{118 - 12 - (143 - 31)\} = 6$$

$$S(1,3) = \text{abs}\{143 - 19 - (143 - 19)\} = 0$$

The above calculations result in an identical table to table 9. Consideration of the reversed problem thus helps to provide insight into the scheduling cost function.

The display of earliest finishing times and the associated slack thus provides a reasonably compact convenient method of allowing the IMPACT user to peep inside this particular cost function should he so desire. The PATH command was implemented for this purpose and the slack so displayed is calculated in the first method described. (By providing similar facilities for other cost functions, a user could utilise the framework of IMPACT for other problems. This capability is discussed further in 7.3).

It should be noted that the slack for an element is dependent upon the permutation. Also the amounts of slack for two elements may not be independent. In the example quoted, the start of processing of job 1 on machine 1 and the start of processing of job 2 upon machine 3 can both be delayed by their respective amounts of slack; whereas the same does not hold for the start of processing of job 2 on machine 3 and of job 4 on machine 3. In the latter case the total increase in the two slack values must be less than 22 units, the slack of each element.

6.2 Interactive Experiences

Initially several job-shop scheduling problems were generated in the manner described earlier with each machine being equally heavily loaded. Problems generated ranged from 12 to 30 jobs upon 3 to 10 machines and were tackled interactively. As experience suggested further facilities for IMPACT they were implemented and their effectiveness assessed.

The experiences that follow are illustrated and explained by sample interactive sessions with IMPACT. The use of the 'COMMENT' command allows what would have been interactive sessions to be described in a convenient fashion. User supplied input is in lower case and the system's replies are in upper case letters.

```
## $run impact*  1=data12/3 2=-junk 4=-q 5=hash
##EXECUTION BEGINS
    ENTER START VALUE FOR R.N.G.
  12345.9

## ## ##
data
```

MACHINES

	1	2	3
1	63	69	3
2	99	34	97
3	69	111	153
4	126	103	85
5	17	107	152
6	36	63	0
7	92	94	79
8	117	113	40
9	7	73	166
10	135	69	10
11	111	39	58
12	128	125	157
TOTALS	1000	1000	1000

```
 ## ## ##
comment
```

The first problem attempted was one of scheduling 12 jobs upon

3 machines; the author was not too adventurous at this stage!

```
$

 ## ## ##
bounds,;
   (   9  -   1080)
   (   6  -   1099)
   (   5  -   1124)
   (   1  -   1132)
   (   2  -   1133)
   (  11  -   1150)
   (   3  -   1180)
   (   7  -   1186)
   (  10  -   1204)
   (   4  -   1229)
   (   8  -   1230)
   (  12  -   1253)


 ## ## ##
comment
A spin-off from the programming of the backtrack procedure had been

the command 'BOUNDS', which allowed one access to the lower bound

for any part permutation.  The use of this facility in this case

revealed that the lowest lower bound was 1080 and that this

corresponded to placing job number 9 first.

$

 ## ## ##
select
  WANT TO ENTER ANY PARAMETERS?
no
  SELECTION CALLED WITH NO PARAMETERS
*SELECTION GIVES COST    1595 PERMUTATION
    6    1   11   10    2    7    8    4    9    5    3   12
 NEW TARGET FOUND


 ## ## ##
exchange,capl;
 NEW TARGET FOUND
  EXCHANGING GIVES COST   1407
 PERMUTATION
    6   1   11   10    2    9    7    5    8    4    3    12


 ## ## ##
comment
Use of the selection technique gave a permutation with cost
```

1595 which was improved upon by exchanging to give a value

of 1407.
$

```
 ## ## ##
merge
  ENTER STRINGS TO BE MERGED
+
*MERGING GIVES COST    1097  PERMUTATION
 9    5    3    2    4    1    7    8   12    6   10   11
 NEW TARGET FOUND


 ## ## ##
exchange,capl;
 NEW TARGET FOUND
  EXCHANGING GIVES COST  1080
 PERMUTATION
   9    5    3    2    4    1    7    8   12    6   11   10


 ## ## ##
comment
```

Merging proved very encouraging and gave a permutation with cost

1097. When this was subjected to exchanging a permutation with

the cost of 1080 was found. The example served to illustrate

the relative merits of the heuristic techniques but proved

disappointing to the author since the optimum was achieved without

any intuition or flashes of brilliance upon his part.
$

```
 ## ## ##
hunch,best;
  GIVES COST   1080


 ## ## ##
path,9,10;
THE PATH BETWEEN ELEMENTS     1  AND    12
```

JOB	BUILD UP OF TIMES			SLACK TIMES		
9	7	80	246	0	0	0
5	24	187	398	1	19	0
3	93	298	551	1	19	0
2	192	332	648	1	19	0
4	318	435	733	1	19	0
1	381	504	736	1	19	0
7	473	598	815	1	19	0
8	590	711	855	1	19	0
12	718	843	1012	1	12	0
6	754	906	1012	1	56	0
11	865	945	1070	1	56	0
10	1000	1069	1080	1	1	0

critical path

```
## ## ##
comment
The 'PATH' command illustrates the critical path for the optimum

sequence found.
$
 halt
      181 CALLS OF THE COST FUNCTION WERE MADE


   THAT'S ALL FOLKS
##EXECUTION TERMINATED
```

It was hoped that the next problem to be tackled would be more

difficult as it consisted of scheduling 15 jobs upon 3 machines.

```
## $run impact* 1=data15/3 2=-junk 4=-q 5=hash
##EXECUTION BEGINS
   ENTER START VALUE FOR R.N.G.
 12345.9


## ## ##
data
```

	MACHINES		
	1	2	3
1	62	68	43
2	107	102	61
3	138	124	100
4	18	2	73
5	30	70	99
6	115	105	85
7	57	36	40
8	35	50	90
9	23	101	100
10	13	85	108
11	113	17	7
12	64	49	78
13	18	111	6
14	79	70	110
15	125	11	0
TOTALS	997	1001	1000

```
 ## ## ##
bounds,;
 (   4  -    1020)
 (   8  -    1085)
 (   7  -    1093)
 (  10  -    1098)
 (   5  -    1100)
 (  12  -    1113)
 (   9  -    1124)
 (  13  -    1129)
 (   1  -    1130)
 (  11  -    1130)
 (  15  -    1136)
 (  14  -    1149)
 (   2  -    1209)
 (   6  -    1220)
 (   3  -    1262)

 ## ## ##
merge
  ENTER STRINGS TO BE MERGED
+
*MERGING GIVES COST    1052 PERMUTATION
   4   8   5   10   3   9   6   12   2   7   1   11   14   13   15
 NEW TARGET FOUND

 ## ## ##
exchange,capl;
 NEW TARGET FOUND
  EXCHANGING GIVES COST   1047
 PERMUTATION
   4   5   8   10   3   9   6   12   2   7   1   11   14   13   15
```

```
##
##
##
comment
```

In this problem the lowest lower bound was calculated as being 1020 and this would require job number 4 to be placed first. The second lowest bound was 1085, a value that was beaten by the use of the merging technique. This means that effectively the problem has been reduced to one of scheduling 14 jobs since the value of 1052 could only be bettered if job number 4 was placed first.

Use of exchanging improved upon the merging permutation to give a permutation with a cost of 1047. At this stage a lack of power was felt by the author since heuristics were needed which would allow one to specify particular elements of the permutation to be constructed; (In this case the first element was to be specified as being 4). In the absence of any such heuristics, lower bounds for permutations beginning with 4 were requested.

```
$


##
##
##
bounds,4;
(   10  -    1043)
(    5  -    1047)
(    8  -    1052)
REST OF BOUNDS SUPPRESSED BY ATTN


##
##
##
comment
```

Continuing further along these lines a value of 1043 was reached fairly quickly.

```
$


##
##
##
```

```
bounds,4,10;
  (    5  -    1043)
  (    7  -    1043)
  (   14  -    1043)
  (   12  -    1043)
  (    1  -    1043)
  (    8  -    1043)
  (    9  -    1043)
  (   13  -    1046)
  (   11  -    1058)
  REST OF THE BOUNDS SUPPRESSED BY ATTN


 ## ## ##
base
   *CC IS       1047   *CAP IS
    4   5   8   10     3    9   6   12   2   7
    1  11  14   13     15


 ## ## ##
intch,5,10;
  NEW TARGET FOUND
   COST     1046 PERMUTATION
   4   10   8   5   3   9   6   12   2   7   1   11   14   13   15


## ## ##
exchange, capl;
 NEW TARGET FOUND
  EXCHANGING GIVES COST  1043
 PERMUTATION
   4   10   8   5   3   9   6   12   2   7   1   14   11   13   15


 ## ## ##
comment
```

Further problems were tackled and fortunately provided greater difficulty than the two already mentioned. One such problem consisted of scheduling 25 jobs upon 3 machines.

$

```
  ## ## ##
merge
   ENTER STRINGS TO BE MERGED
+
*MERGING GIVES COST   1074  PERMUTATION
  19   9   13   23   24   6   1   2   4   3   7   8   5   11   12
  10   15   16   14   22   17   18   20   25
NEW TARGET FOUND



  ## ## ##
exchange,capl;
  NEW TARGET FOUND
   EXCHANGING GIVES COST 1041;
 PERMUTATION
  19   9  23  13  24   6   1   2   4   3   7   8   5   11   12
  10  15  16  14  17  22  18  20  25


  ## ## ##
bounds,;
 (  19  -    1028)
 (   9  -    1036)
 (  23  -    1041)
 (  24  -    1042)
 REST OF THE BOUNDS SUPPRESSED BY ATTN


  ## ## ##
comment

Investigation of the lower bounds revealed that to better the cost

of the permutation obtained from the application of exchanging upon

the result of the merging solution, only permutations beginning

with (19) or (9) need be considered.

$


  ## ## ##
switch


  ## ## ##
bounds,;
 (   25  -    1028)
 (   19  -    1037)
 (   14  -    1042)
REST OF THE BOUNDS SUPPRESSED BY ATTN
```

```
## ## ##
comment
```

At this time it was felt that one ought to be able to consider either the normal job-shop problem, or the inverted problem. An appropriate command 'SWITCH' was therefore incorporated into IMPACT. This command alters the data so that the inverted problem is considered. A second use of the command returns one to the original problem.

The use of this command in this instance allowed the author to ascertain (by the 'BOUNDS' command) that the choices for the first position of the inverted problem (which is of course the last position of the normal problem) could be similarly restricted. Following up this investigation a permutation with a cost of 1028 was found and was seen to be optimal. The combination of the ability to ask for lower bounds and to tackle the inverted problem was thus seen to be useful.

```
$


## ## ##
merge
   ENTER STRINGS TO BE MERGED
+
*MERGING GIVES COST   1060 PERMUTATION
  25  14   3  12  16   5   8   4   2   1   7  20  18  22   6  10
  13  15  11  17  21  19  24  23


## ## ##
exchange, capl;
   EXCHANGING GIVES COST  1041
 PERMUTATION
  25  14  16   3  12   5   8   4   2   1   7  20  18  22   6  10
  13  15  11  17  21  19  24  23


## ## ##
intch,19,23;
 NEW TARGET FOUND
  COST     1029 PERMUTATION
  25  14  16   3  12   5   8   4   2   1   7  20  18  22   6  10
  13  15  11  17  21  23  24  19
```

```
 ## ## ##
exchange,capl;
  NEW TARGET FOUND
    EXCHANGING GIVES COST   1028
PERMUTATION
  25  14  16   3  12   5   8   4   2   1   7  20  18  22   6  10
  13  15  11  17  21  23  24  19

 ## ## ##
halt
   447 CALLS OF THE COST FUNCTION WERE MADE
  THAT'S ALL FOLKS

##EXECUTION TERMINATED
```

A 30 jobs 3 machines problem was similarly treated and solved without undue difficulty. The success proved encouraging and stimulated the provision of further interactive facilities for the scheduling problem.

The above pointed to the ease with which extensions could be made to IMPACT.

6.21 Lower bounds and their connection with the inverted problem

It had been pointed in Chapter 2 that the use of the branch-and-bound technique upon the inverted problem could greatly reduce the amount of tree-searching necessary. The combination of the ability to ask for lower bounds for a part permutation (the BOUNDS command) and the ability to consider the inverted problem (the SWITCH command) allowed the author to recognise a severe weakness in the procedure for the calculation of the lower bounds as described in 2.2. This weakness lies in the statement

'As the last job to be processed has not yet been decided the job, out of those not yet assigned, that occupies the last m-k machines for the least time is chosen.'

This means that in the calculation of the lower bounds for a part permutation a different job might be chosen as last in the calculation of the lower bound $g^{(k)}$ than that chosen as last in the calculation for the lower bound $g^{(k+1)}$. Thus the calculated lower bound could be lower than that that could be obtained with just a little more computation. Furthermore this weakness might not be recognised until a large number of the currently unassigned elements had been allocated. Removing this drawback would thus improve the performance of the branch-and-bound method as far as the total number of vertices searched is concerned, although of course slightly more effort would be expended per vertex examined. To strengthen the bounds even further, the observation of good starting elements (with respect to lower bounds) for the inverted problem revealed that the choice of the last element for the normal problem could often be more restricted than all those elements of 1 to n which are not specified to be in the starting

part permutation. The first elements for the inverted problem are of course the last elements for the normal problem and thus if a permutation is known which has a cost C, and the use of lower bounds for the inverted problem indicate that only elements $\{\ell_1, \ell_2, \ldots, \ell_j\}$ placed first can possibly result in a permutation with a cost less than C, then only the set of elements $\{\ell_1, \ell_2, \ldots, \ell_j\}$ need be considered for the last position of the normal problem. Similarly of course the choice for starting elements of the normal problem can be restricted. It is also possible that the set of elements $\{\ell'_1, \ell'_2, \ldots, \ell'_{j'}\}$ for the starting position may not be distinct from $\{\ell_1, \ell_2, \ldots, \ell_j\}$.

6.22 Additional facilities for the 'BOUNDS' command

It was realised at this stage that the BOUNDS command ought to allow the user to specify not only a starting part permutation but also (optionally) the element with which to finish the permutation. (The possibility of being able to specify a part permutation with which to finish the permutation was considered at this time. Further analysis of the job-shop scheduling problem was however called for in this respect and it was convenient to perform this analysis in parallel with the interaction. The results of such analysis are discussed later in 6.241).

Implementation of this facility was easily achieved because of the modular nature of IMPACT. The routine for calculating lower bounds was modified so that if one requested lower bounds for a part permutation beginning with $(i_1, i_2, \ldots, i_j)$ and ending

with (k), a very high value was returned for the invalid part permutation (i_1, i_2, . . ., i_j, k). This is by no means the most efficient method but its implementation required little re-programming and consequently did not disrupt the pursuance of the investigation.

Some more problems were generated at this stage and examined interactively, and some of the existing problems were re-examined. The size of the problems ranged from 20 to 30 jobs upon 3 to 10 machines and the new facility allowed one to determine more exactly where the computational effort ought to be directed. The facilities for instructing IMPACT to examine these regions were rather weak however, and a consequence of the above implementation meant that the backtracking procedure could be instructed to search that part of the tree beginning with (i_1, i_2, . . ., i_j) and finishing with (k). The fact that the bound for the part permutation (i_1, i_2, . . ., i_j, k) is given a high value means that this part of the tree will not be examined, which is as desired. (A trivial modification to the 'accounting' procedure for counting the number of vertices examined had of course to be made). The avoidance of reprogramming was particularly useful since the interactive facilities of search interruption, tree interrogation and pruning were still available. Upon using the new facility, good (sometimes optimum) results were obtained with a comparatively small amount of computing time. Tree pruning was used to direct the search into the promising areas. The results were good in the sense that they were near to the optimum (within a few percent).

The important result of the interaction was that frequently optimum solutions were being determined and being recognised as such fairly quickly. The verification stage mentioned in 2.424 as being the reason why backtracking was as effective (or ineffective) as branching from the lowest bound had thus been curtailed in enough of the examples to offer encouragement.

6.23 The normal or the 'reverse' problem?

When backtracking, as described above, was performed a decision had to be made whether to attack the normal problem or the inverted one. To allow a choice to be made during an interactive session an appropriate command 'SWITCH' would be used to switch from the normal problem to the reverse problem, or vice-versa. When interaction was taking place if it appeared that no success was being achieved by backtracking through a normal job-shop problem, the author would interrupt the search and begin backtracking upon the inverted problem. Often this would result in success. Initially this backtracking was performed in 'manual' fashion as illustrated by figure 28.

Frequently the number of still potentially fruitful descendents of a part permutation would become large and the manual organisation became tedious. Use of the SWITCH command and manual backtracking upon the inverted problem could prove fruitful since the author could take advantage of his experiences before the switch was made. As an example, a situation might occur in which a part permutation beginning with, say, (7,1,23,14) would have descendents worthy of investigation in 8, 19 and 27; i.e. the part permutations (7,1,23,14,8),

```
## ## ##
bounds,;
 (   7 -    114)
 (   4 -    117)
 (   2 -    119)
 (   1 -    121)
 (   6 -    127)
 (   3 -    127)
 (   5 -    132)
 (   8 -    136)

## ## ##
switch


## ## ##
fixbound,;
 ENTER ELEMENT TO OCCUPY LAST POSITION
7;
 (   5 -    114)
 (   4 -    121)
 (   1 -    126)
 (   6 -    127)
 (   8 -    128)
 (   2 -    133)
 (   3 -    138)
 (   7 - 999999)

## ## ##
fixbound,5;
 ENTER ELEMENT TO OCCUPY LAST POSITION
7;
 (   6 -    114)
 (   1 -    114)
 (   4 -    114)
 (   8 -    115)
 (   3 -    120)
 (   2 -    120)
 (   7 - 999999)

##/## ##
switch


## ## ##
fixbound,7;
 ENTER ELEMENT TO OCCUPY LAST POSITION
5;
 (   2 -    114)
 (   1 -    115)
 (   4 -    119)
 (   3 -    121)
 (   8 -    127)
 (   6 -    132)
 (   5 - 999999)

## ## ##
```

Figure 28: An example of manual backtracking. (This example uses n=8 only to illustrate the principle involved. In practice n was larger).

(7,1,23,14,19) and (7,1,23,14,27). However, consideration of the inverted problem with the last element fixed as 7, (the first element in the normal problem) could reveal that the elements 8, 19 and 27 would have to be placed very early in the permutation to be able to better the lowest cost found so far. Thus any permutation beginning with these elements could not end with (14,23, 1,7). In other words, use could be made of preclusion if backtracking was to be performed from both ends of the permutation in some manner. In this way some knowledge of not only the last element of the permutation but the last j elements could be exploited.

The reduction in necessary tree searching when this approach was taken manually suggested that an automatic method would be worth testing and the following algorithm was devised for use within IMPACT.

6.24 Branching from both ends of the permutation.

This method has advantages that the lower bounds calculated can be stringent and preclusion is exploited. The method is termed 'branching from both ends of the permutation' and in essence a permutation is built up as follows:-

The first element of the permutation is filled by calculating the lower bounds for each candidate for this position and the element with the lowest lower bound is chosen. The n th position of the permutation is similarly filled by calculating the lower bounds(for position n) for each as yet unassigned elements, and by choosing that element with the lowest lower bound. (In the calculation of the lower bounds advantage can of course

be taken of the knowledge of the first element of the permutation). The next step is to allocate the second position in the permutation in a similar manner, again utilising any knowledge of the positions fixed so far. The (n-1) th position is then filled and then positions 3, n-2, 4, etc. When a complete permutation has been built up its cost may be calculated and if this is lower than the best cost so far found then a record is made of it. The formation of the initial permutation is thus simply a selection procedure. Backtracking may be performed by considering the complete permutation as two permutations each having n/2 elements and systematically removing the end elements from each of these alternately until a position is reached which is deemed worthy of further investigation. The search then proceeds as before and terminates when all possibilities have been investigated.

The complete scheduling tree for n = 4 to be searched under this method may be represented as in figure 29. The box

1	3

represents those permutations that begin with element 1 and finish with element 3, whilst

1,4	3

represents those permutations beginning with 1,4 (in that order) and ending with element 3.

As an example of the method, the figure 30 depicts the parts of the tree examined for the problem of scheduling 6 jobs upon 3 machines, where the times taken for each job upon each machine are as in Table 4. Of interest in the search is the fact that it is possible that the lower bound calculated for a permutation beginning with element i and ending with element j might be

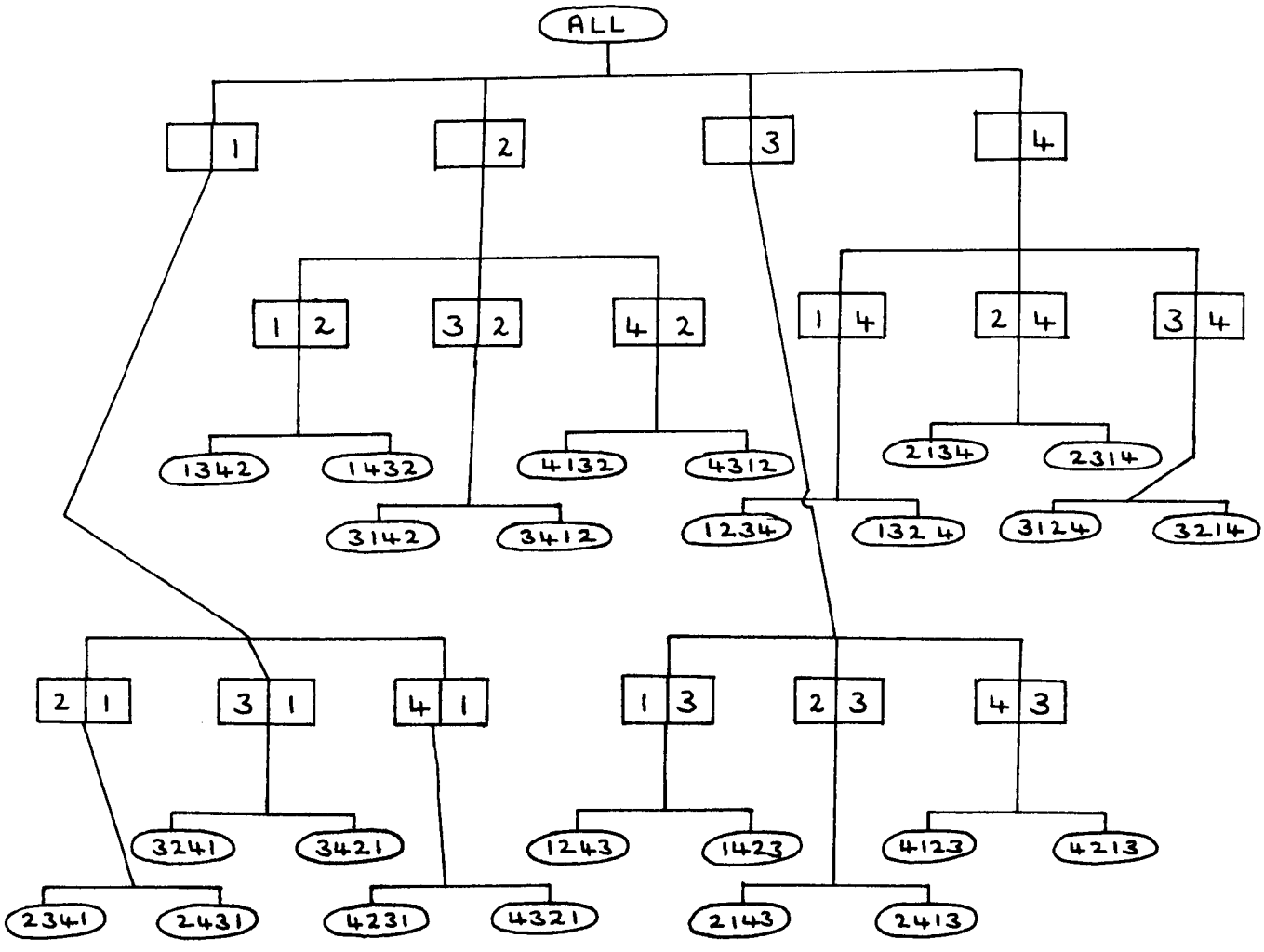


Figure 29: The permutations of (1,2,3,4) arranged in a manner suitable for a 'branch-from-both-ends' search.

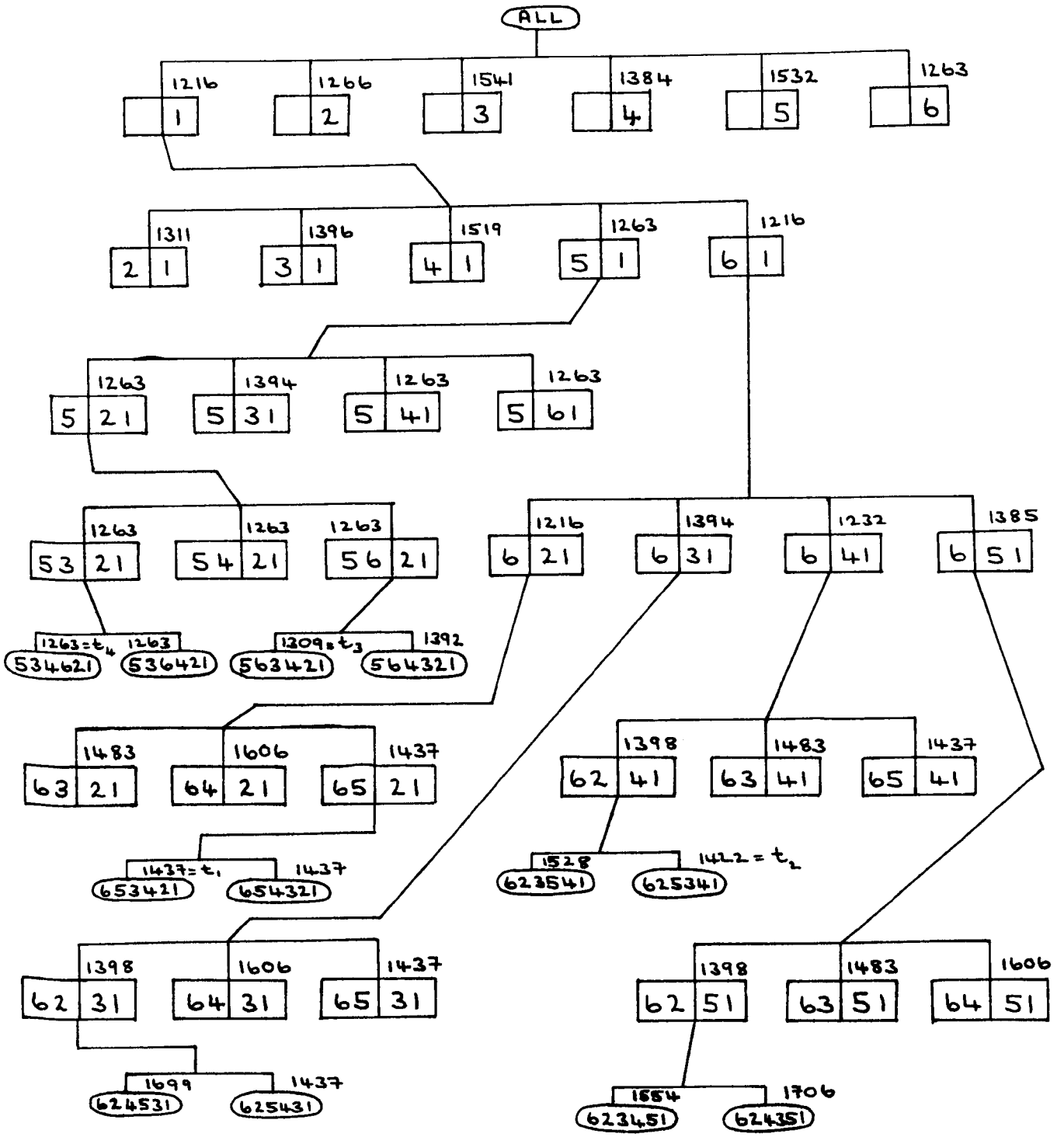


Figure 30: A scheduling tree examined under branching from both ends of the permutation.

lower than the bound calculated for a permutation ending with element j. This anomaly arises because of the method of the calculation of the lower bounds but could be easily cleared up by modifying the bounds routine so that the bound for a permutation beginning with $(i_1, i_2, \ldots, i_r)$ and ending with $(\ell_q, \ell_{q-1}, \ldots, \ell_1)$ is at least as high as its 'predecessor', the permutation (in the inverted problem) beginning with $(\ell_1, \ell_2, \ldots, \ell_q)$ and finishing with $(i_{r-1}, i_{r-2}, \ldots, i_1)$.

No implementation difficulties will be caused by such a modification since the lower bound of the predecessor is available just before the calculation of the lower bound for a part permutation. The difficulty can be avoided in a more convenient manner by strengthening the lower bound still further.

6.241 A further strengthening of lower bounds

The manner of calculation of the set of lower bounds $g^{(1)}, \ldots, g^{(m)}$ described earlier utilises knowledge of r + 1 elements or assigned jobs, namely jobs $i_1, i_2, \ldots, i_r$ and ℓ_1. In effect no use is made of the knowledge that jobs $\ell_q, \ell_{q-1}, \ldots, \ell_2$ immediately precede job ℓ_1 in the permutation and this is a weakness of the bounds described. If $g^{(s)}$, $1 \leq s \leq m$, is one of the lower bounds calculated then the method of calculation assumes

i) there will be no more idle time upon machine s, and

ii) the last job will progress without any idle time from machine s through to machine m. Since the last q jobs $(\ell_q, \ldots, \ell_2, \ell_1)$ to be processed are known, consideration of the reverse problem allows one to calculate the finishing times of job ℓ_q upon each machine under the sequence

$(\ell_1, \ell_2, \ldots, \ell_{q-1}, \ell_q)$, and hence the idle time upon each of these machines (whilst processing these jobs) can be determined. Furthermore machine s in the normal problem corresponds to machine (m + 1 - s) in the reverse problem, and thus any idle time upon machine s can be determined and added to the lower bound $g^{(s)}$. Similarly any idle time during the progress of job ℓ_q through machines s to m can also be determined and added to $g^{(s)}$ to produce a more realistic lower bound.

In calculating the amount of idle time to add onto a lower bound $g^{(s)}$ it is convenient to consider the total idle time on the path determined by the start of processing of job ℓ_q upon machine s, the subsequent processing of jobs $(\ell_{q-1}, \ldots, \ell_2, \ell_1)$ upon this machine and finally the processing of job ℓ_1 upon machines s+1, s+2, ..., m. By consideration of the reverse problem the idle time on machine s' is

$$f'_{P_q}(l_q,s') - \sum_{t=2,q} D(\ell_t,s') - \sum_{t=s',m'} D(\ell_1,t)$$

In effect the idle time on this path is the finishing time under the reverse problem of job ℓ_q upon machine s' under the sequence P_q, less the processing times of jobs ℓ_{q-1}, . . ., ℓ_2 upon machine s', and less the processing time of job ℓ_1 upon machines s', s'+1, . . ., m'. Machine s' in the reverse problem corresponds to machine m + 1 - s' in the normal problem and thus the idle time upon machine s in the normal problem during processing of jobs $\ell_q, \ldots, \ell_1$, is

$$J^{(s)} = f'_{p_q}(\ell_q, m+1-s) - \sum_{t=2,q} D(\ell_t, m+1-s) - \sum_{t=1,m+1-s} D(\ell_1,t)$$

This correction when added to the formula for lower bounds can substantially strengthen the lower bounds.

The strengthened lower bounds as described above have been implemented and tested in IMPACT. The implementation was performed in a manner convenient from an experimental viewpoint. In effect an existing routine was modified fairly easily. It is not pretended that such a modification is as efficient as it could possibly be from the amount of computation necessary to calculate a lower bound. Its merit lay in the fact that substantial reprogramming was avoided and hence the problem solving effort was not disrupted. The strengthened bounds can however be more efficiently calculated and easily illustrated as follows

$$g^{(k)} = f_{p_r}(i_r,k) + \sum_{s\epsilon I_n \sim (P_r \cup P_q)} d(i_s,k) + f'_{p_q}(\ell_q,k')$$

where $P_r = (i_1, i_2, \ldots, i_r)$

$P_q = (\ell_1, \ell_2, \ldots, \ell_q)$

$k' = m+1-k$

and $f'_{p_q}(\ell_q, k')$ represents the earliest finishing time of job ℓ_q upon machine k' under the part permutation P_q, for the reverse problem.

This formula is seen to be symmetric no matter whether the reverse or the normal problem is considered. Formulae following for calculating lower bounds when dominating jobs are considered do not at first sight appear to have symmetry in this respect. The term $\sum_{t=1}^{q} d(\ell_t, m)$ in $H^{(u)}$ however is equivalent to $f'_{p_q}(\ell_q, 1')$ since no idle time occurs upon the first machine until all n jobs have been processed upon that machine. The lower bounds can thus be seen to be symmetric with respect to either the reverse or the normal problem.

The strengthened lower bounds are depicted in figure 31.

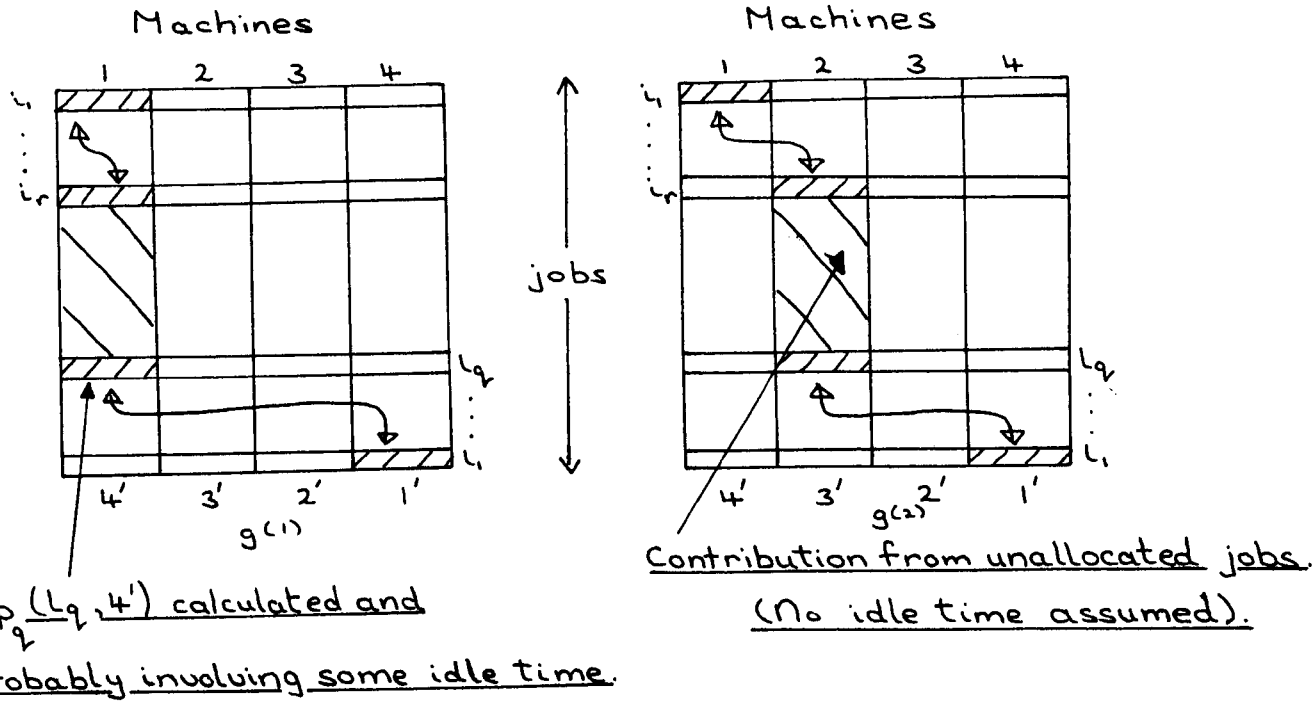
6.242 An additional set of Lower Bounds

Lomnicki reports that whilst testing the backtracking branch-and-bound approach to the job-shop scheduling problem it was observed that some of the problems which required an excessive number of vertices to be searched had a common feature. This was that one of the jobs in the problems was such that its processing times were dominating the times taken by the other jobs. By the use of some appropriately designed bounds Lomnicki found that these particular problems could be solved fairly quickly.

In the case of three machines, the following lower bounds were designed for a permutation beginning with

$P_r = (i_1, i_2, \ldots, i_r)$:

$h^{(1)} = f_{p_r}(i_r, 1) + d(k_1, 1) + d(k_1, 2) + d(k_1, 3)$



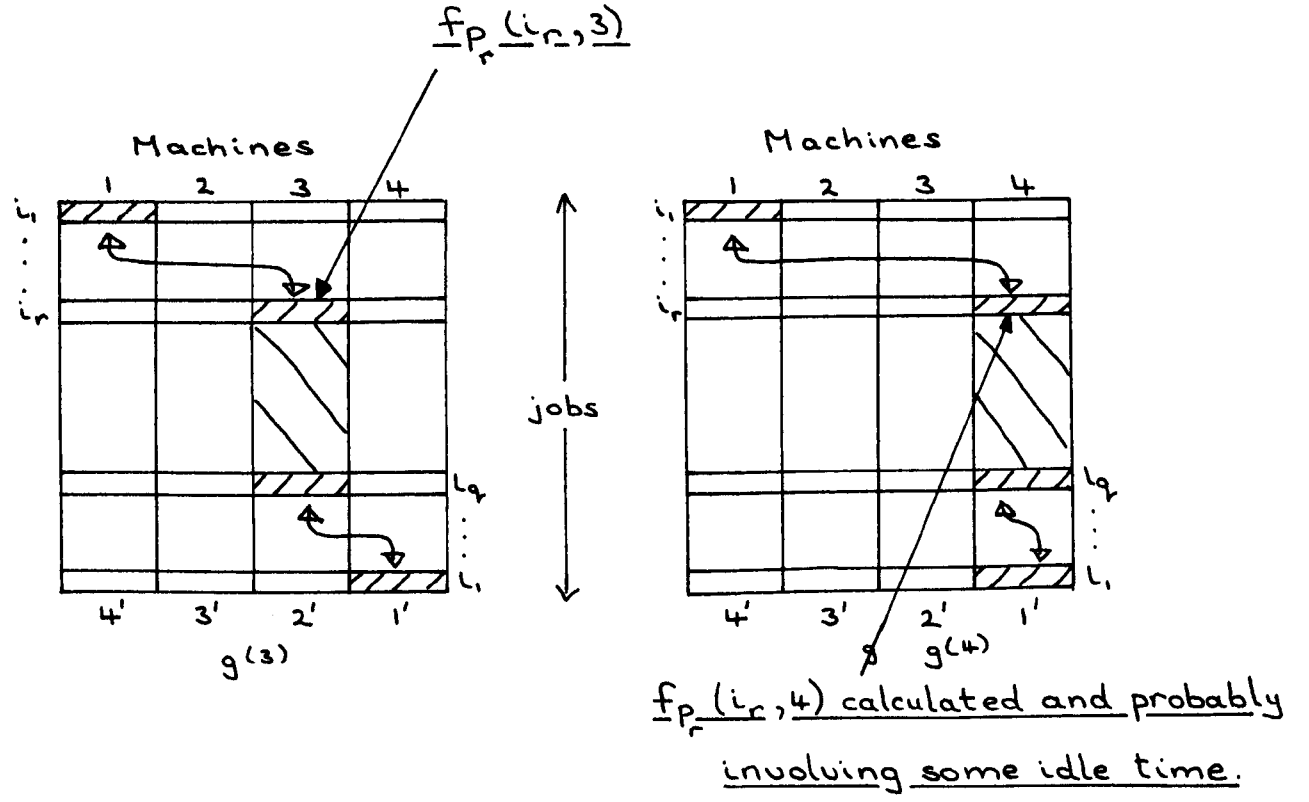


Figure 31 : An illustration of the lower bound $g^{(k)}$ when $m = 4$.

$$+ \sum_{v \neq i_1, \ldots, i_r, k_1} \min\{d(v,1), d(v,3)\}$$

where k_1 is the job, v, which is not one of $i_1, \ldots, i_r$ and which maximises

$$d(v,1) + d(v,2) + d(v,3).$$

$$h^{(2)} = f_{P_r}(i_r,2) + d(k_2,2) + d(k_2,3)$$

$$+ \sum_{v \neq i_1, \ldots, i_r, k_2} \min\{d(v,2), d(v,3)\}$$

where k_2 is the job, v, which is not one of $i_1, \ldots i_r$ and which maximises

$$d(v,2) + d(v,3).$$

These bounds can be interpreted in the following manner. $h^{(1)}$ is the time taken for job i_r to finish upon the first machine and bearing in mind that the job k_1 dominates the other entries the contribution from this job is $d(k_1,1) + d(k_1,2) + d(k_1,3)$. The remaining contributions are from the as yet unallocated jobs. As we do not know whether these jobs will precede or follow job k, we place them where their contribution will be smallest, thus choosing the minimum of d(v,1), d(v,m). (If $d(v,1) < d(v,m)$ we are effectively placing job v before job k_1).

Similarly $h^{(2)}$ is the time for job i_r to be finished upon the second machine plus the contribution from the job which dominates the other jobs upon the second and third machines, namely $d(k_2,2) + d(k_2,3)$. The contributions from the remaining

unallocated jobs are made on the assumption that as we do not know whether or not these jobs precede k_2 we place them in the position where their contributions are smallest, i.e. we choose for each of them the minimum of (d(v,2),d(v,3)).

The above bounds can be generalised for the case of m machines giving m - 1 lower bounds where $h^{(u)}$, the u-th lower bound is calculated as

$$h^{(u)} = f_{p_r}(i_r,u) + \sum_{t=u}^{m} d(k_u,t)$$

$$+ \sum_{v \neq i_1, \ldots i_r, k_u} \min\,(d(v,u),d(v,m))$$

where k_u is the job, v, not one of $i_1, \ldots, i_r$

which maximises $\sum_{t=u}^{m} d(v,t)$, and u = 1, . . . m-1.

(The reason why there are only m-1 lower bounds and not m is that $h^{(m)}$ would coincide with $g^{(m)}$).

A paper by McMahon and Burton (19) suggests almost identical bounds which take into account job-dominance. They however prefer to calculate $h^{(u)}$ as

$$h^{(u)} = f_{p_r}(i_r,u) + \max_{k_u \neq i_1, \ldots i_r} \left\{ \sum_{t=u}^{m} d(k_u,t) \right.$$

$$\left. + \sum_{v \neq i_1, \ldots i_r, k_u} \min\,(d(v,u),d(v,m)) \right\} .$$

This lower bound requires more effort in computation but one would expect it to be superior. It has not however been adopted in this report.

The modification to the bounds to fit for a permutation beginning with $(i_1, \ldots, i_r)$ and ending with $(\ell_q, \ell_{q-1}, \ldots, \ell_2, \ell_1)$ is to calculate the bounds as

$$H^{(u)} = f_{P_r}(i_r, u) + \sum_{t=u}^{m} d(k_u, t)$$

$$+ \sum_{\substack{v \neq i_1, \ldots, i_r, k_u, \ell_1, \ldots \ell_q}} \min \left\{ d(v,u),\ d(v,m) \right\}$$

$$+ \sum_{t=1}^{q} d(\ell_t, m)$$

where k_u is the job, v, which is not one of $i_1, \ldots, i_r, \ell_1, \ldots, \ell_q$ which maximises $\sum_{t=u}^{m} d(v,t)$. It can be seen that the lower bound simply allows for the positioning of the last q jobs after the dominating job k_u.

Figures 32 and 33 illustrate the formulae for the lower bounds. Figure 33 illustrates lower bounds $H'^{(k)}$ for $k = 2, \ldots, m-1$ where dominating jobs upon the reverse problem are considered. (The case $k = 1$ is not allowed since $H'^{(1)}$ would be identical to $H^{(1)}$).

6.3 Implementation of branching from both ends of the permutation

As was the case for the simple backtracking algorithm this branching method lends itself to push-down stack implementation. In this case two stacks are utilised and the size of each is approximately half the size of the one required for simple backtracking and consequently the storage requirements of the two methods are comparable. In implementing branching from both

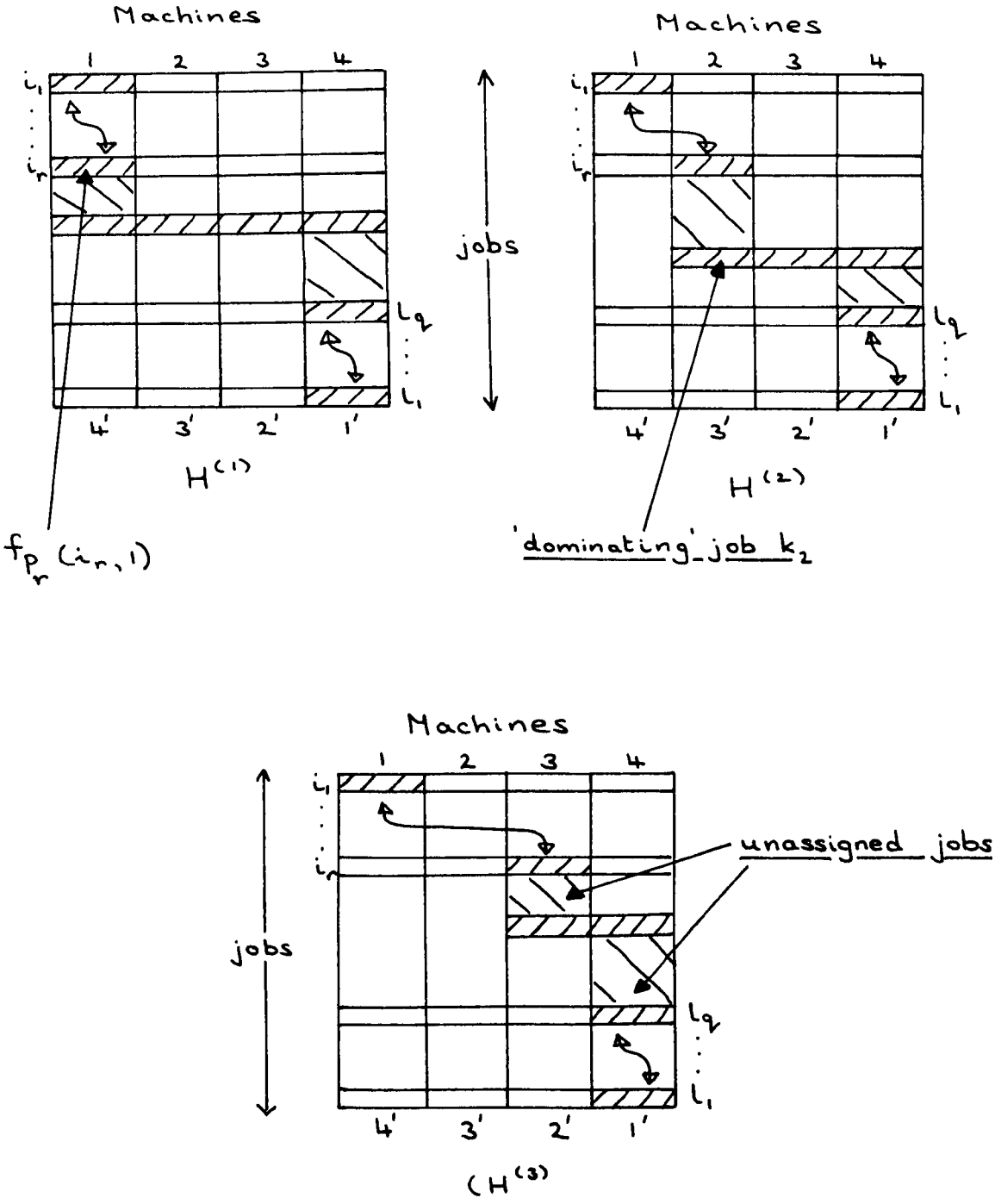


Figure 32: An illustration of the lower bounds $H^{(k)}$ when $m = 4$.

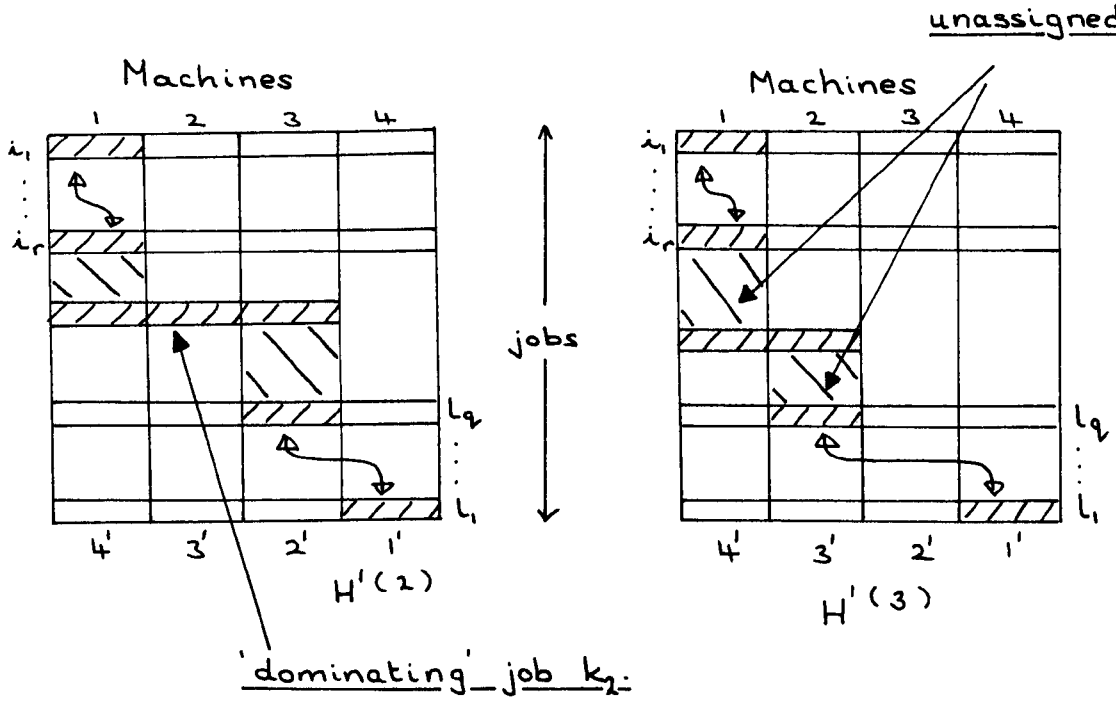


Figure 33: The lower bounds $H'^{(k)}$ for $m = 4$.

ends of the permutation two choices were immediately available; the above method of utilising two stacks, or a method which interpreted a part permutation $(i_1, i_2, \ldots, i_j)$ in a different fashion. This requires that the part permutation be considered as a permutation beginning with $(i_1, i_3, \ldots, i_{j-1})$ and ending with $(i_j, i_{j-2}, \ldots, i_4, i_2)$ (for j even). It can be seen that the corresponding routines for calculating lower bounds associated with a part permutation, and for calculating the cost associated with a complete permutation would require alteration. This approach may have been easier to implement for the author but was not adopted since it might have clouded the author's thinking about the lower bounds for the job-shop scheduling problem. Such an interpretation of an input permutation would however widen the range of existing heuristics and this facet is discussed later in 7.0.

Tree pruning methods similar to those for simple backtracking have also been implemented and further information about these techniques is given under the command 'BOTHENDS' in Appendix 1.

6.4 Behaviour of the 'Branch from both ends' approach

A number of 'small', hopefully difficult problems were generated in order to assessthe performance of the new algorithm. Ten 12 job - 3 machine problems were generated with the loadings the same upon each machine. A comparison of the simple backtracking algorithm and the branch from both ends approach was made. In the case of the latter method the approach was tested using the strongest bounds available as described in 6.25 and 6.26, and also using the bounds of 6.21 strengthened only to allow the lower

bound for a part permutation to be at least as high as its predecessor (see 6.24).

The measure of computation required for each method was taken to be the number of vertices searched. Because the new approach calculates potentially more powerful bounds the amount of computation per vertex would be higher. In fact for the new method with the simpler bounds it was higher by about 13% whilst an increase of about 38% was observed with the strengthened lower bounds. The aim of the investigation was to determine what improvement was gained from the new method and the strengthened lower bounds. Simple backtracking being the only available similar approach was used as a crude yardstick in assessing the difficulty of a particular problem.

For all three approaches both the normal and the inverted problem were considered, although it was suspected that the new branching approach with the strengthened lower bounds would behave uniformly for a particular problem no matter whether the inverted or the normal problem was considered. The heuristic approach of applying the exchanging technique to the resulting permutation from merging was also used in these examples in an attempt to determine what might be achieved by tree pruning. The heuristics were applied to both the normal and the reverse problems and the best results chosen.

The results of the investigation are given in tables 12, 13, 14 and 15. It can be seen that in some of the simple backtrack attempts the search was terminated after 50,000 vertices had been examined; an '*' marks those examples in which this occurred. The column 'found after' indicates how many

Problem number	Backtracking upon the normal problem: Value Found	Found After	Total Examined	Backtracking upon the inverted problem: Value Found	Found After	Total Examined
1	1074	462	696	1074	9619	50,000*
2	1123	137	50,000*	1123	1060	1,227
3	1068	87	87	1068	1869	1,869
4	1172	18,500	50,000*	1169	1043	40,429
5	1107	18,138	30,216	1107	3041	3,041
6	1105	43,830	50,000*	1105	564	564
7	1090	81	50,000*	1090	88	120
8	1075	13,192	17,181	1075	478	959
9	1099	1,061	1,061	1099	5771	11,225
10	1072	1,076	1,076	1072	79	79
Mean		9,656	20,532*		2361	10,951*

Average number of vertices searched per sec $\underline{\sim}$ 460.

The * indicates that the maximum number of vertices to be examined, was reached.

Table 12: The behaviour of the backtracking algorithm on problems of scheduling 12 jobs upon 3 machines.

Problem Number	Value found by the heuristic
1	1083
2	1135
3	1071
4	1170
5	1121
6	1135
7	1090
8	1120
9	1099
10	1072

Table 13: The results produced for the 12 jobs 3 machines problems by the heuristic of exchanging upon the solution produced by merging

	Searching upon the normal problem			Searching upon the inverted problem		
Problem Number	Value Found	Found After	Total Examined	Value Found	Found After	Total Examined
1	1074	527	2,806	1074	1,620	5,903
2	1123	1,380	2,915	1123	1,581	2,150
3	1068	564	564	1068	494	494
4	1169	2,528	15,431	1169	17,961	30,754
5	1107	336	1,717	1107	2,676	2,828
6	1105	10,280	10,309	1105	10,615	10,615
7	1090	84	179	1090	411	421
8	1075	3,818	3,979	1075	2,957	3,472
9	1099	619	628	1099	1,652	1,652
10	1072	125	125	1072	1,853	1,853
Mean		2,026	3,865		4,182	6,014

Average number of vertices searched per sec $\simeq$ 405.

Table 14: The behaviour of branching from both ends of the permutation on problems of scheduling 12 jobs upon 3 machines

	Examination of the normal problem			Examination of the inverted problem		
Problem Number	Value Found	Found After	Vertices Examined	Value Found	Found After	Vertices Examined
1	1074	231	1074	1074	564	2410
2	1123	146	1361	1123	221	761
3	1068	67	67	1068	134	134
4	1169	1326	8693	1169	1085	3700
5	1107	615	767	1107	374	481
6	1105	138	167	1105	138	167
7	1090	65	160	1090	361	371
8	1075	698	884	1075	842	1344
9	1099	500	509	1099	459	459
10	1072	65	65	1072	65	65
Mean		385	1375		424	989

Rate of examination of vertices $\simeq$ 333 per sec

Table 15: Branching from both ends of the permutation with improved bounds upon 12 jobs/3 machines problems

vertices were examined before the best value finally found was discovered. The difference between this entry and the total number of vertices searched indicates what verification was necessary, and thus what computation could not be avoided by tree cutting should the optimum be required. The sessions were run in an 'interactive' mode, i.e. from a terminal. They were, however, not truly interactive since no feedback was provided by the author. The interactive facilities were used solely to inspect the state of the search from time to time by means of the attention button and local commands. The author was thus able to determine where computational effort was being expended. The average rate of investigation of vertices was calculated but all of the rates could be increased since at times computational efficiency was sacrificed (slightly) for ease of experimentation and implementation.

Example number 4 was a difficult one for the simple backtracking algorithm since no matter whether the normal or the reverse problem was considered an enormous amount of computation would have to be expended. In the case of the reverse problem some 40,000 vertices had to be examined in order to verify that the optimum had been found, whilst in the normal problem 50,000 vertices were examined without even finding the optimum! In this particular problem the heuristic method gave a value better than that found under backtracking from the normal problem. Branching from both ends of the permutation with the original bounds showed that the problem could be solved more quickly, but of real interest was the fact that under this method the verification stage required some 13,000

vertices no matter whether the reverse or the normal problem was considered. The reduction in verification was thus due to either preclusion facilitated by the new method of branching, or the better performance of the lower bounds because of the new method of branching, or both. The branching from both ends method with the strengthened lower bounds achieved even more spectacular success upon this problem; resulting in a maximum of about 8,500 vertices being searched. Similarly example number 6 yielded to the new method. This example illustrates the differences that can occur depending upon whether the normal or the reverse problem is considered. The simple backtracking approach solved the inverted problem in a small number of nodes (564) whereas the normal problem had not been terminated after 50,000 vertices had been examined. Branching from both ends with the simpler bounds achieved a compromise in this problem. No matter whether the normal or the reverse problem was considered some 10,000 vertices were required for the solution of the problem. Branching from both ends with the strengthened bounds however solved either problem in just 167 vertices. It can be seen that the average number of vertices necessary for solution of the problems had been drastically reduced by the new branching method and the strengthened bounds.

The algorithm was tested further upon some problems of scheduling 5,6,7,8,9,10,11,12,13 and 14 jobs upon 3,5 and 7 machines. The data was generated as before with no excess loading upon any machine, and the results are given in Appendix 2. For the case of three machines it appeared initially that the addition of an

extra job was not greatly increasing the necessary computation. The number of vertices examined increased by about a factor of 2 or 3 for each additional job. There were of course great variations in the number of vertices necessary for the solution of problems of the same size. The examples of 12 jobs upon 3 machines in this investigation appeared to be very well behaved compared with the results of table 15. This peculiarity was attributed to the particular examples themselves. The values in table 15 included three problems which were not well-behaved. It also appeared that about twice as much computational effort was being expended in the verification stage for a problem as opposed to finding an optimum. This suggests that branching from the lowest bound would not behave substantially better than backtracking. It might also mean that further improvement could be achieved in the calculation of the lower bounds.

Example number 4 for the scheduling of 14 jobs upon 3 machines was interesting in that a phenomenal amount of effort was necessary in verifying that an optimum had been found. The data and an optimum solution for this problem are given in Appendix 2. Upon further interactive investigation of this problem it was seen that the majority of the verificational effort was expended in attempting to find a solution with a cost of 1094 or 1095, when the optimum solution was 1096. To achieve the figures of 1094 the third machine would require no idle time once job number 13 was accepted by it. Apparently this cannot be achieved but it appears that there is no quick way of establishing this.

In the case of three machine problems it thus appeared that most problems would succumb without excessive computation but occasionally one might present difficulties.

For the 5 - machine and 7 - machine problems it seemed that more effort was required for a particular problem but that the number of vertices necessary for solution again increased by a factor of 2 to 3 for the addition of a job. The verification stage to search stage ratio again appeared to be about 2:1. As more computational effort was required for 5 - machine problems than for 3 - machine problems, and more again as far as 7 - machine problems was concerned, only up to 11 jobs were considered for the 5 - machine problems and up to 10 jobs for the 7 - machine problems Again an occasional difficult problem was encountered (examples 1 and 4 for the 11/5 problems, and examples 5, 9 and 10 for the 10/7 problems). The number of vertices examined per second appears to be constant within problems involving the same number of machines once consideration is taken of set-up costs for placing the data in a convenient fashion. The number of vertices examined per second is approximately 280, 220 and 175 for 3, 5 and 7 machines respectively.

The initial investigation into the method of branching from both ends of the permutation indicated a substantial improvement in the use of branch and bound for job-shop scheduling problems. Tree-pruning facilities similar to those of chapter 5 have therefore been implemented and are described further in Appendix 1. Larger (n=30, m=5) problems have also been tackled interactively and optimum solutions obtained within a short elapsed time ($\underset{\sim}{\sim}$ 40 minutes) and using a small amount of computing time (1 - 2 minutes).

Chapter 7

7.0 The Extendability of IMPACT

As was suggested in 6.5 the interfacing of a suitable routine between the searching procedure and the cost function could greatly alter the branching process followed. Such a routine would effectively take an output permutation from the search and transform it (by the application of a fixed permutation) into another permutation to be used as input to the cost function. If the original permutation is $P_n = (i_1, i_2, \ldots, i_n)$ and T_n is the permutation $T_n = (t_1, t_2, \ldots, t_n)$, then another permutation $J_n = j_1, j_2, \ldots, j_n$ may be defined so that $j_s = i_{t_s}$.

As an example, if n = 10 and $P_{10} = (1, 2, \ldots, 9, 10)$ defining

$$t_k = \begin{cases} (k+1)/2 & \text{for k odd} \\ n+1 - k/2 & \text{for k even} \end{cases}$$

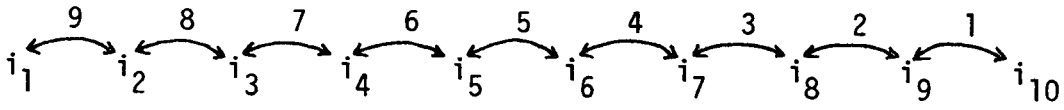
gives T = (1,10,2,9,3,8,4,7,5,6) and

hence $J_{10} = (i_1, i_{10}, i_2, i_9, i_3, i_8, i_4, i_7, i_5, i_6)$

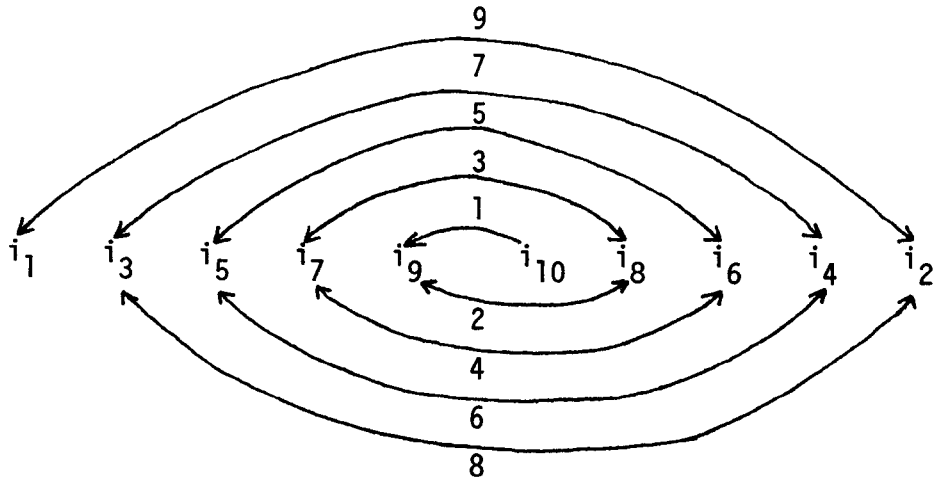
$= (1, 10, 2, 9, 3, 8, 4, 7, 5, 6)$

This transformation would thus allow the normal backtracking procedure to be used to branch from both ends of the permutation. (The appropriate procedure for calculating lower bounds would of course require modification). In a similar fashion merging would be transformed into merging from both ends of the permutation. Exchanging behaves in a different fashion however. Figure 34 shows that exchanging would be performed upon pairs of elements whose positional distance apart in the input permutations are 1, 2, 3, . . ., n-2, n-1 (in that order). The order of attempted interchanges is

shown by the numbers above the arrows.



Output permutation and interchanges tried



Input permutation and the actual interchanges tried

Fig. 34: The effect of exchanging when a different interpretation is placed upon the output permutation

The existing heuristics may thus be easily extended by the introduction of a suitable transformation permutation.

7.1 The Application of IMPACT to other discrete optimisation problems

IMPACT may be used for attempting permutation problems where all input permutations are not feasible. The job-shop scheduling problem where the restriction of no-passing is relaxed may be

taken as an example. In such problems the cost function provided should return an arbitrarily high value should the input permutation be infeasible. Care must however be taken since certain routines will have the undesirable feature of producing a large number of infeasible permutations. In the case of the extended job-shop scheduling problem the number of feasible solutions that need to be considered for n jobs upon m machines is $(n!)^{m-2}$; converting to a permutation problem would mean the permutation was of length (nm) and thus the price of using the heuristic of generating random permutations is high; the chance of generating a feasible permutation that we wish to examine is

$$\frac{(n!)^{m-2}}{(nm)!} \leq \frac{1}{(m!)^{n}(n!)^{2}}$$

Some of the other routines are however usable. The exchanging algorithm would not misbehave if the initial input permutation is feasible since the nature of this heuristic would mean that infeasibilities would not be introduced.

A use of IMPACT not mentioned previously in this report is for attacking discrete problems in which the cost function is not well-defined. Data could be liable to error or the optimal solution might be required to some cost function subject to certain constraints upon the solution. These constraints might be difficult to state since they may rely on the subjective judgement of a works manager who 'doesn't like that machine placed there'. An approach that could be adopted would be to use IMPACT to produce a number of good solutions in the hope that one of them will satisfy the manager's aesthetics.

7.2 A branch-and-bound approach for the problem of assigning facilities to locations

Gavett and Plyter (20) have attempted to solve this assignment problem by adopting a branch-and-bound approach. Their method is based on that of Little et al for the travelling salesman problem and only the main points of the method will be described here. The problem of assigning n facilities to n locations is converted to one of solving an assignment problem of minimising

$$\sum_{i,j=1}^{N} a_{ij}\, x_{ij}$$

such that

$$\sum_{i=1}^{N} x_{ij} = 1 \qquad \forall\, j$$

$$\sum_{j=1}^{N} x_{ij} = 1 \qquad \forall\, i$$

where the x_{ij} take integer values

$$0 \leq x_{ij} \leq 1,$$

the a_{ij} are matrix entries and $N = \frac{1}{2}n\,(n-1)$. A further restriction is imposed in that the solution to the assignment problem must be a feasible solution to the original problem. As an example the problem of assigning 4 plants to 4 locations is transformed to the assignment problem of selecting 6 elements from the associated table 16 so that no two elements are chosen from the same row or column.

	A	B	C	D
A	0	6	7	2
B	6	0	5	6
C	7	5	0	1
D	2	6	1	0

Distance Matrix D

	1	2	3	4
1	0	28	25	13
2	28	0	15	4
3	25	15	0	23
4	13	4	23	0

Flow Matrix F

	1-2	1-3	1-4	2-3	2-4	3-4
A-B	168	150	78	90	24	138
A-C	196	175	91	105	28	161
A-D	56	50	26	30	8	46
B-C	140	125	65	75	20	115
B-D	168	150	78	90	24	138
C-D	28	25	13	15	4	23

The associated assignment problem

Table 16: Conversion from one assignment problem to a different one

In the example the locations are labelled A, B, C, D for clarity and the entries in the expanded table are simply the products of entries in the two matrices. The labelling of the rows and columns illustrates how the entry was determined. The entry 75 = 5 x 15,for instance, corresponds to placing the facilities 2 and 3 at locations B and C (not necessarily in that order).

The method proposed by Gavett and Plyter differs from that used in the travelling salesman problem in two respects. The cost matrix is manipulated in order to maximise an initial lower bound and during the tree construction certain elements in the cost matrix are eliminated because certain rows and columns are inadmissible.

The results of this investigation are given in Table 17. The times quoted are for the running of a FORTRAN II program upon an

IBM 7074 computer and it is claimed that the largest value of n that could be conveniently handled was n = 8. It is not clear whether the restriction was due to storage limitations or to the computational effort required; although the table suggests the latter.

n	Computing Time
4	3 sec
5	15 sec
6	45 sec
7	14 min
8	42 min

Table 17: The results of Gavett and Plyter

The authors give no data for the individual problems, nor any indication as to how dependent upon data the process is. They claim that their preliminary program requires a good deal of refinement and speculate that with both major and minor changes in the program the computing time may be reduced by a significant factor. Some difficulty was also encountered in keeping track of the proper permutation of the facility pair at each node of the tree. A method which has no such problems, appears to be easier to understand and which 'plugs' easily into IMPACT is now described.

The problem has been shown to be easily expressed as a permutation problem in 1.4 and lower bounds upon the cost of any permutation beginning with $(i_1, i_2, \ldots, i_r)$ can be calculated without undue difficulty.

7.21 Lower Bounds for the Problem of assigning Facilities to Locations

If no elements in the problem have been assigned then a lower bound to the cost of any assignment can be determined by forming the sum of the cross products of the elements of the flow matrix F and the elements of the distance matrix D. There are $\frac{1}{2}n(n-1)$ cross products to be formed and each element is to be used once and once only so that the sum of the products is minimised. It can be easily seen that such a sum will be minimised if the elements of F are sorted into ascending order, those of D sorted into descending order and then corresponding elements multiplied. In the example given in table 3 the F values sorted ascendingly are

0 0 0 1 2 2 3 4 5 5

and the D values are (in descending order)

3 2 2 2 2 1 1 1 1 1

and thus a lower bound is

0 x 3 + 0 x 2 + 0 x 2 + 1 x 2 + 2 x 2 + 2 x 1 + 3 x 1 +

4 x 1 + 5 x 1 + 5 x 1

$\therefore$ A lower bound is 25

Now consider a part permutation $(i_1, \ldots, i_r)$, the cost function is

$$\sum_{s=1}^{n} \sum_{t=s+1}^{n} f_{st} d_{\ell_s \ell_t}$$

where ℓ_s denotes the destination of facility s

The cost function can be rewritten as

$$\sum_{s=1}^{r} \sum_{t=s+1}^{n} f_{st} d_{\ell_s \ell_t} + \sum_{s=r+1}^{n} \sum_{t=s+1}^{n} f_{st} d_{\ell_s \ell_t}$$

$$= \sum_{s=1}^{r} \sum_{t=s+1}^{r} f_{st} d_{\ell_s \ell_t} + \sum_{s=1}^{r} \sum_{t=r+1}^{n} f_{st} d_{\ell_s \ell_t}$$

$$+ \sum_{s=r+1}^{n} \sum_{t=s+1}^{n} f_{st} d_{\ell_s \ell_t}$$

$$= \sum_{s=1}^{r} \sum_{t=s+1}^{r} f_{st} d_{i_s i_t} + \sum_{s=1}^{r} \sum_{t=s+1}^{n} f_{st} d_{i_s \ell_t}$$

$$+ \sum_{s=r+1}^{n} \sum_{t=s+1}^{n} f_{st} d_{\ell_s \ell_t}$$

(since the first r elements are known)

The first term of the above can be calculated precisely. A minimum value for the second term can be determined in r stages, each stage corresponding to a different value for s (= 1, 2, . . ., r). Consider s = k, say, the term gives

$$f_{k,k+1} d_{i_k \ell_{k+1}} + f_{k,k+2} d_{i_k \ell_{k+2}} + \ldots + f_{k,n} d_{i_k \ell_n}$$

which corresponds to the cross product of the unallocated elements of row k of the flow matrix F and the unallocated elements of row i_k of the distance matrix D. These unallocated elements are known and their minimum contribution will, as before, occur if the largest-smallest products are formed. Such a procedure causes no computational difficulties.

The third term of the cost function corresponds to the sum of the cross-products of the unallocated elements of the matrix and the minimum contribution is again easily determined as before.

Example. For the table 3 in 1.4, the lower bounds for a part permutation beginning with 3,5 are calculated by writing the cost of the permutation $(3,5,\ell_3,\ell_4,\ell_5)$ as

$$f_{12}^{5\times 2} d_{35} + f_{13}^{2\times} d_{3\ell_3} + f_{14}^{4\times} d_{3\ell_4} + f_{15}^{1\times} d_{3\ell_5}$$

$$+ f_{23}^{3\times} d_{5\ell_3} + f_{24}^{0\times} d_{5\ell_4} + f_{25}^{2\times} d_{5\ell_5}$$

$$+ f_{34}^{0\times} d_{\ell_3\ell_4} + f_{35}^{0\times} d_{\ell_3\ell_5}$$

$$+ f_{45}^{5\times} d_{\ell_4\ell_5}$$

The contribution of the first term is 10.

The free elements of row 3 of the distance matrix have values 1, 2 and 1 and their minimum contribution gives 2 x 1 + 4 x 1 + 1 x 2 = 8.

The free elements in row 5 of the distance matrix have values 3, 2 and 1 and the minimum contribution is 3 x 1 + 0 x 3 + 2 x 2 = 7.

The remaining unassigned elements of the flow matrix have values 0, 0 and 5 and those of the distance matrix are 1, 2 and 1. The minimum contribution is thus 0 x 2 + 0 x 1 + 5 x 1 = 5.

The lower bound for a permutation beginning with (3,5) is thus 10 + 8 + 7 + 5 = 30.

The complete tree for the above problem is given in figure 35.

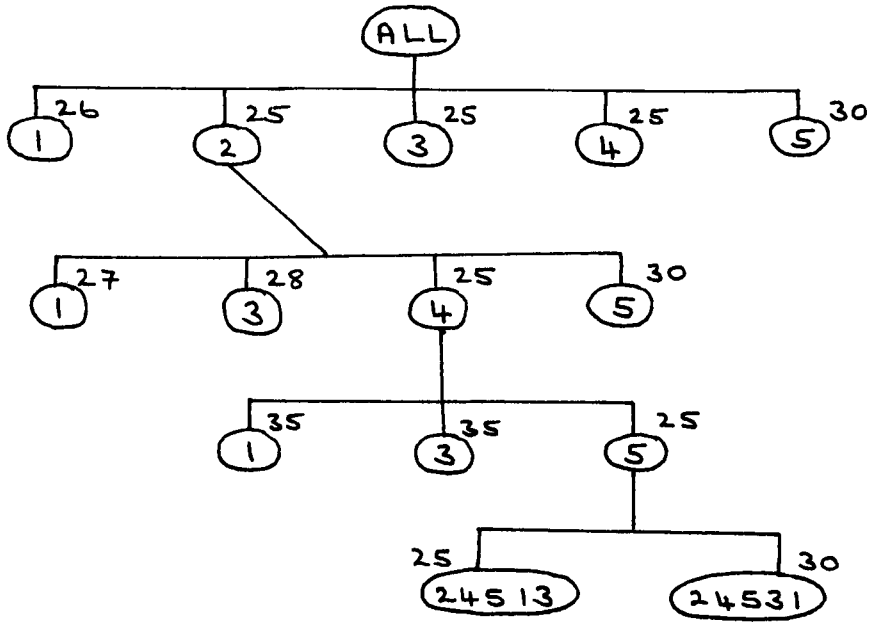


Figure 35: Tree for the assignment problem of 5 plants and Locations

7.22 Performance of the Algorithm

The algorithm was tested upon several sets of examples supplied in the paper of Nugent et al (21). The data for the examples are given in Appendix 4, together with the solutions found by backtracking. It can be seen that the layout of the plants for the cases when n = 6 and n = 8 would allow advantage to be taken of symmetry but the general nature of the backtracking approach implemented in IMPACT does not utilise such information. The amount of computation expended is thus intended to illustrate the quality of the lower bounds. Table 18 gives the results of the investigation.

Table 18: Results of backtracking upon the assignment problem

Problem Size	Vertices Examined to find Optimum	Total Vertices Searched	Time Taken (Secs)
5	15	15	0.1
6	85	129	0.5
7	112	339	1.7
8	36	3116	18.9

Even on these small examples it appears that the computation necessary rises steeply with the size of the problem. The total number of vertices to be searched in verifying that an optimum had been found in the 8 plants problem was 3116 - 36 = 3080. This indicates that the lower bounds are not stringent enough. This feeling was supported by the behaviour of the algorithm when used interactively upon Nugent's larger problems. In these problems 12, 15 and 30 plants were to be assigned to locations and interruption of the search revealed that the state of the tree being examined would be as in figure 36. It can be seen that at high levels of the tree the lower bounds are small but increase in general with every level of the tree. The lower bounds are thus feeble in that very little (if any) of the tree is eliminated early in the search. It thus appears that with the present lower bounds the algorithm is limited to small problems ($n \lesssim 10$) and furthermore the display of a tree does not prove to be of great help in suggesting where a particular solution might be improved.

Values given by Nugent for the larger problems (n = 12, 15, 20 and 30) were much superior to those produced by various heuristics available in IMPACT although the author did not exert a lot of effort in attempting to better Nugent's results. In this respect the set of heuristics available in IMPACT could be used as a yardstick to assess the performance of special purpose heuristics.

```
TREE BEING EXAMINED AFTER     2312 VERTICES
 TARGETVALUE IS      290 AND LAST ELEMENT IS TO BE    0


  281
   .
   .
   .      284    286
7( 6)---(10)---( 9)---(
   .
   .
   .      271    277    281    282
6( 7)---(12)---( 9)---( 6)---( 8)---(
   .
   .
   .      261    264
5(11)---( 8)---(12)---(
   .
   .
   .      251    254    255    259    259    261    264
4( 5)---( 6)---( 9)---(11)---( 1)---( 7)---(12)---( 8)---(
   .
   .
   .      249    249    251    251    251    252    253    253    254
3( 4)---( 8)---( 1)---( 5)---(12)---(11)---( 9)---( 7)---(10)---( 6)---(
   .
   .
   .      246    250    250    251    251    253    254    256    257    258
2( 3)---( 1)---(10)---( 4)---( 6)---( 5)---( 9)---( 8)---(11)---( 7)---(12)---(
   .
   .
   .      244    244    244    244    244    244    244    244    244    249
1( 2)---(12)---( 4)---( 5)---( 3)---( 1)---( 8)---( 9)---(10)---(11)---( 7)
                                                          249
                                                     ---( 6)---(
```

Figure 36: A typical tree on interruption of a branch and bound search for the problem of assigning facilities to locations

7.3 Other Man-Machine Approaches: the Travelling Salesman Problem

A recent paper by Krolak, Felts and Marble (22) has dealt with a man-machine approach to the travelling salesman problem and some good results are claimed by the authors. The results are good in the sense that in the particular problems tackled (100 - 200 cities) lower costs were obtained interactively in a shorter amount of cpu time than certain heuristic methods, that had previously been advocated. A first attempt was made in the interaction to organise the data to suggest a tour. The philosphy was that if a salesman must visit a number of cities which are far apart, and in each city there are two or more customers who are fairly close together, in general, the time spent planning the tour yields the greatest savings in mileage reduction if a correct decision is made in the order to visit the cities, rather than in deciding which order to visit the customers in each city. An attempt was thus made to find the regions around which a high density of customers are located. This idea is similar to that in 3.41 where consideration was given to groups of sequences of jobs, but in this case the order within a group is disregarded for the moment.

The method of finding clusters of cities utilised a solution to the assignment problem associated with the travelling salesman problem in which the expression

$$\sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij}x_{ij} \text{ is minimised subject}$$

$$\text{to } 0 \leq x_{ij} \leq 1 \qquad \forall\ i \text{ and } j$$

$$\sum_{j=1}^{n} x_{ij} = 1 \qquad \forall\ i$$

$$\text{and } \sum_{i=1}^{n} x_{ij} = 1 \qquad \forall\ j$$

The optimal assignment is not necessarily a feasible solution to the travelling salesman problem but does tend to indicate local order and each set of cities connected by the assignment was taken as a cluster by Krolak et al. The clusters were then replaced by geometric centres and the assignment problem solved with a reduced number of 'cities', in this case 'county seats'. The problem solver was able to ask for repeated application of this process and thus could be provided with a plot of the initial solution. This compares with the idea used in the job-shop scheduling where it was realised that the cost of a permutation was insufficient for the problem solver, and that the results ought to be displayed in some meaningful fashion. For the job-shop problem the earliest finishing times and the critical path were displayed; the travelling salesman problem does of course lend itself to a more graphical display. Also supplied to the problem solver initially in the interactive approach to the travelling salesman problem was information suggesting how to piece clusters of towns together. It is claimed that if the assignment problem solved in the first stage is modified by assigning large values to those entries selected by the first solution the solution to the modified problem often gives links which connect the subtours. Thus the computer is delegated the role of suggesting alternatives to the user. In the job-shop problem the alternatives were made more precise by the display of appropriate lower bounds, or parts of a tree.

Krolak et al claim that the problem solver uses his advantage over the computer in that he can 'see' the whole problem. That is he can envision a solution that accounts for interactions

between all of the regions that must be connected. He can also consider a few good solutions provided by the heuristic algorithms available and try to use good features of each in order to correct errors made in incorrectly connecting regions to each other. The intuition provided here by the problem solver appears to be more than was needed in the job-shop problem. The reason for this is that the job-shop problem yielded more to analysis and thus any intuition was directed in this direction. It then ceased to be intuition and became fact.

Krolak's final stage in the interaction is to use simple tour improvement routines in the hope of mopping up any oversights by the problem solver. An attempt is also made to define some stopping criterion by the determination of a lower bound to see what possible maximum reduction in tour length could be achieved. No claim is made that any indication of how to obtain a reduction is provided by the bound.

Information was displayed by Krolak et al by the use of the graph plotter and they claim that the problem would be a perfect application for a computer graphics system. This is one of the main reasons why the travelling salesman problem was not investigated extensively in this report.

7.4 Conclusions

7.41 An Appraisal of IMPACT

The interactive features of IMPACT proved particularly useful to the author; the ability to interrupt and interrogate a search was perhaps the most valuable. There are some features which would have been useful but which were not implemented since their construction would have retarded the progress of the experimental solution of the discrete optimisation problems. Keyword parameters would have made the man-machine interface less artificial but would have necessitated major reconstruction of the command analyser in IMPACT. Conditional interruptions to some of the processes appear inviting; e.g. 'examine 2,000 vertices of the scheduling tree unless an improvement is made, in which case .....'

The development of IMPACT took place when the only available terminal was an IBM 2741 (typewriter) terminal. Consequently, IMPACT is oriented towards the physical characteristics of this mechanical device. In later stages of the work described 2260 character display terminals were available and it was possible to display information noiselessly and more quickly. The full power of such devices has not however been exploited by IMPACT, but only interface routines would require modification in order to do so. The availability of more powerful terminals (graphical displays with a light pen) would again greatly increase the facilities for a problem solver using IMPACT.

7.42 The value of interaction

The main result of the investigation showed that interactive sessions could provide insight into the behaviour of certain black box

functions. In the case of the job-shop scheduling problem such insight enabled what was thought to be an impractical algorithmic approach (the branch-and-bound method) to be tuned up so that it is now useful for larger (of the order of 30 jobs) problems. By useful it is meant that the optimum may be found, or a good solution (within a few per cent of the optimum) may be determined, within a reasonable amount of computing time. In the case of a good solution an indication of its goodness can also be provided, as can a description of how this value might be improved upon. The tuning up of the branch-and-bound approach was achieved by making the bounds calculated more stringent (or realistic) and by adopting a slightly different branching procedure. The limitations of the approach suggested are not apparent and there seems to be no reason why the method should not give good results for larger problems.

It is felt that the insight gained into the cost function could not have been achieved as easily without the use of a terminal system which provided interaction. The easy access to the powerful facilities provided allowed one to assimilate information much more quickly and more easily than would have occurred in conventional batch usage. Ideas were readily tested out and the ability to 'single-shot' through some of the search procedures was particularly useful. A lesson that became apparent as the investigation into the job-shop scheduling problem progressed was that extra effort in strengthening lower bounds could be well worthwhile in terms of the amount of searching that could be avoided because of the more powerful bounds. This perhaps

explains why weaknesses existed in the bounds as described by Lomnicki. He tested his ideas out on small problems ($n \leq 10$) and used hand computation. Consequently he would have been restricted to a small number of small problems since the concentration required and infallibility necessary in his arithmetic would be prohibitive. He would thus be unlikely to encounter many difficult problems and would be even less likely to be able to ascertain why they were difficult.

Appendix 1. User's Guide to IMPACT

Command Descriptions

Informative and Definitive Commands

HELP In an attempt to cater for absent-minded users (of whom the author is one), the HELP command asks IMPACT to display the names of all available commands (but no subcommands). The routine is not interruptable and has no effect upon CAP.

Example Usage

```
help

   COMMANDS AVAILABLE IN IMPACT ARE

HELP        RECAP       COMMENT     PAUSE       TIME
RECONOFF    RESET       HALT        GIAN        CATALOG
DELETE      RENAME      BASE        NEWSTATE    POPCAP
HUNCH       INTCH       MOVE        REV         CYCLE
WEAVE       TONFRO      SHUFFLE     DISPER      CMC
RANFIX      PERLEX      SELECT      MERGE       EXCHANGE
BOUNDS      TRAKBAK     BOTHENDS    FIXBOUND    SIO
OLDBILL     PATH        MODIFY      DATA        SWITCH
RESUME
```

RECAP

This command allows the user to display the best solution found so far. The permutation BEST (see Predefined Permutations in 4.62) and its associated cost are displayed. The routine is not interruptable, requires no parameters and has no effect on CAP.

COMMENT

Comments may be entered from the terminal by the user. The lines entered after the command line are recorded onto the permanent

record file provided the recording facility is being used at that time. A dollar sign '$', in the first position of a line (column number 1) turns off the comment facility. The facility was implemented so that the user could record any ideas as he progressed through a problem. He might thus be able to explain why a particular method was adopted.

The routine is not interruptable and has no effect on CAP.

PAUSE

This command provides an exit from IMPACT to the operating system under which IMPACT is running (in this case MTS). The command is available only for a conversational run of IMPACT and places the user in the MTS environment in a state that may be resumed later by the use of the MTS command '$restart'. The command was used by the author mainly as an aid to debugging but could be of use for other reasons (e.g. manipulating or saving an MTS file, or communicating with the system operator). There are no parameters to the command, it is not interruptable and does not affect CAP.

TIME

This command allows one to gauge how much time has been spent in computation since the last time the command was used. The number of calls of the cost function which have occurred in this period is also displayed. The cpu time is displayed in two parts, the time spent in the problem state and the time spent in the supervisor state, the units being milliseconds.

The routine is not interruptable, requires no parameters, and has

no effect on CAP. For further information see 'Timing Considerations' in Chapter 4.

RECONOFF

This command is useful in connection with the permanent record that is being kept. It allows the user to switch the record keeping on or off at will. The command has no parameters since the user will be prompted. It is not interruptable and does not affect CAP.

RESET

Provides the facility for redefining CAP and CC without invoking the cost function. Parameters are a permutation and a cost. The CAP is replaced by the new permutation (other levels of CAP being left unaltered) and similarly CC is replaced by the new cost. The routine is not interruptable.

example usage: (n = 6)

RESET, 4,5,2,1,6,3,1507;

The above would reset CAP1 to (4,5,2,1,6,3) and CC1 to 1507.

HALT

The HALT command is used to terminate the problem-solving session. Certain statistical information may be provided after prompting by IMPACT and a return to the operating system (MTS) is then made. All temporary information (named permutations, CAP etc.) is lost.

The routine is not interruptable.

GIAN (Give It A Name).

This allows a permutation (or part permutation) to be defined by attaching a name to it. The elements of the permutation can then be retrieved by specifying the name that was attached. Parameters are the permutation name and the permutation (in that order). Certain names (see Predefined Permutations in 4.62) may not be used and an attempt to use them will result in i) termination of the job in batch, or ii) corrective prompting for terminal usage.
The routine is not interruptable, and has no effect on CAP.

example usage: (n = 6)

```
GIAN, FRED, 2,1,5,4,3,6;
```

The above would cause the permutation (2,1,5,4,3,6) to be stored and it could later be referred to as 'FRED'.
GIAN, ABCD, CAP1; causes the permutation which is the current active permutation to be stored and given the name ABCD. If no current active permutation exists then the error is recognised and appropriate action taken.
CAVEAT USER: No check is made that the parameters submitted as a permutation or part permutation are valid in the sense that they do not have repeated, missing or out-of-range elements.

IMPLEMENTATION NOTE If the user's storage for named permutations is incapable of holding the permutation named then this is recognised as a user error and the corresponding action is taken.

CATALOG

Allows the user to examine the contents of named permutations. The command CATALOG without parameters will display the names and contents of all permutations which the user has defined. Use of the command followed by a permutation name will result in the contents of the named permutation being displayed. If no such name exists corrective action will be taken in conversational mode, whilst execution will cease in batch.

The routine is not interruptable and does not affect CAP.

example usage: In the situation formed by the GIAN examples earlier, the command

CATALOG, FRED;

would result in IMPACT displaying

FRED 2 1 5 4 3 6

whilst the command

CATALOG

(with no parameters) would cause IMPACT to display

FRED 2 1 5 4 3 6

ABCD ℓ_1 ℓ_2 ℓ_3 ℓ_4 ℓ_5 ℓ_6

where $\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6$ was the current active permutation at the time ABCD was defined.

DELETE

Allows one to 'forget' a permutation named earlier, thus deleting it from storage and freeing the space. The user must supply as a parameter the name of the permutation to be deleted. A non-existant name is

treated as an error and corrective action taken if possible. The routine is not interruptable and has no effect on CAP.

example usage:

DELETE, ABCD; causes IMPACT to forget the permutation earlier named ABCD.

RENAME

The name of a permutation defined earlier by the use of the GIAN command may be altered by use of the RENAME command. One must supply as parameters the old name and the new one. Upon successful completion of the command the old name will be forgotten and the permutation may henceforth be referred to by its new name.

The routine is not interruptable and does not affect CAP.

example usage:

RENAME, FRED, BILL;

This would cause the permutation (2,1,5,4,3,6) named FRED earlier, to be renamed BILL.

Note: An error is detected if the old name did not exist or if the new name already exists, and appropriate action is taken. It is not possible to rename a permutation by its old name in a single use of this command, i.e. RENAME, BILL, BILL; is an error.

BASE

Causes IMPACT to display CAP1, the current active permutation, if it exists. Otherwise a message to that effect is issued. The routine is not interruptable, requires no parameters, and has no effect on CAP.

NEWSTATE

Resets BC (the best cost obtained, or the targetvalue) to an arbitrarily high value and forgets that any improvement was ever made in the targetvalue. BEST, the permutation associated with the targetvalue is left unaltered. The routine has no parameters, is not interruptable and does not affect CAP.

POPCAP

This command discards the latest version of CAP, replacing it with the version which existed immediately before. In effect the CAP stack is pushed up. The routine is not interruptable and has no parameters.

Permutation Manipulation Commands.

HUNCH

The user is allowed to enter what he considers to be a permutation of 1 to n which he desires to be submitted to the cost function. The resulting cost is then displayed. The permutation submitted (provided it is indeed a valid permutation) becomes the new CAP. The routine is not interruptable and parameters required are the elements of a permutation.

In terminal usage the submission of elements which do not form a permutation of 1 to n will result in a descriptive error message and a request for the permutation to be reinput. In batch usage an error of this nature will cause termination of the run.

example usage: (n = 6)

The input

HUNCH, 2,6,3,4,1;

will result in the message

ELEMENT 5 APPEARS WRONG NUMBER OF TIMES. RE-ENTER.

whilst

HUNCH, 2,6,3,4,1,5;

would respond with

COST 1427

PERMUTATION 2 6 3 4 1 5

(if 1427 was indeed the cost of this permutation).

Note: in this, and all other commands unless specifically stated to the contrary, the parameter string may contain named of permutations or part permutations.

INTCH

This command allows the user to interchange two elements in CAP, thus forming a new permutation which is submitted to the cost function and the resulting cost displayed. The new permutation becomes CAP (the other being pushed down).

The routine is not interruptable and the parameters are the two elements (not their positions) which are to be interchanged.

example usage: (n = 6)

if CAP2 is 1 2 4 3 6 5 and CC2 is 1824

and CAP1 is 3 4 2 1 5 6 with CC1 as 1627

then the input line

INTCH, 4,1;

might result in

COST 1653

PERMUTATION 3 1 2 4 5 6 being displayed and the CAP stack would then be of the form

CAP2 3 4 2 1 5 6 CC2 1627

CAP1 3 1 2 4 5 6 CC1 1653

MOVE

This command allows the user to interchange two blocks of elements in CAP. The resulting permutation becomes CAP and its associated cost is displayed. The blocks need not be of the same size and are specified by end elements. The parameters are thus of the form b_1, e_1, b_2, e_2; (in that order) where b_i is the beginning of the i th block and e_i is its end. A block may consist of a single

element in which case $b_i = e_i$ for some i, but the specification of a b_i, e_i pair in which the position e_i in CAP is before that of b_i is treated as an error, as is the case if any of the blocks overlap (i.e. have common elements). The routine is not interruptable.

example usage: (n = 8)

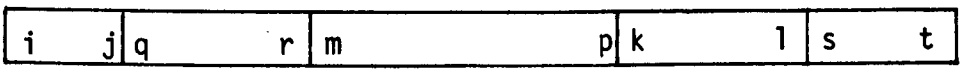
if CAP1 is 4 3 2 8 5 1 6 7 then

MOVE, 3,8,6,7;

results in the permutation 4 6 7 5 1 3 2 8 being submitted to the cost function.

initial permutation

resulting permutation



REV

This command permits the user to reverse a block in the current active permutation and have the cost associated with the new permutation displayed. The new permutation also becomes CAP. If CAP consists of $(i_1, i_2, \ldots, i_n)$ then the permutation obtained by specifying the reversal of the block defined by elements i_j and i_k $(j \leq k)$ will be $(i_1, i_2, \ldots, i_{j-1}, i_k, i_{k-1}, \ldots, i_{j+1}, i_j, i_{k+1}, \ldots i_n)$. The user must specify as parameters the beginning and end of the block to be reversed. (The elements

themselves (not their positions) specify the endpoints of the block). The routine is not interruptable.

example usage: (n = 6)

if CAP1 is 4 3 5 2 6 1 then

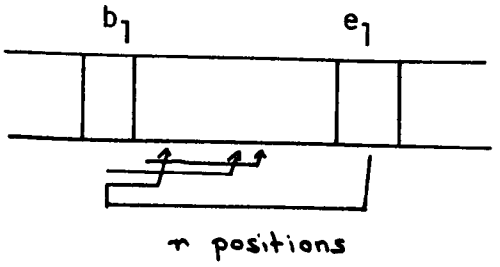
REV,3,6;

results in the permutation 4 6 2 5 3 1 being submitted to the cost function.

CYCLE

The user may specify a block of elements in CAP which is to be cycled (to the right) a number of positions. This displacement, which is at the choice of the user, is applied to the block a number of times (again specified by the user). After each cycle, the resulting permutation is submitted to the cost function and a record of the best permutation and its cost is kept. Finally the best permutation obtained in this manner and its cost are displayed. This permutation also becomes CAP1 and its cost CC1.

The parameters to the command are the start and end of the block, the number of positions to cycle, and (optionally) the number of times to cycle. Omission of the last parameter results in a default value of one being taken. The number of positions specified may be greater than ℓ, the number of elements in the block, but modulo ℓ is applied to the parameter.



each element moves r positions to the right within the block, if necessary wrapping around.

CYCLE, b_1,e_1,r [,k];

example usage: (n = 6)

Suppose CAP1 is 4 5 2 3 6 1

CYCLE,5,6,2;

results in the permutation 4 3 6 5 2 1 being submitted to the cost function and also becoming CAP.

The routine is interruptable and after an interruption the following local commands are available

1) HELP - Gives the names of the other local commands.
2) QUIT - Abandons the cycling.
3) REST - Restarts the cycling from the position in which it was interrupted.
4) HOWF - Asks how far the cycling process had gone.
5) BASE - Displays the last permutation that was submitted to the cost function.
6) SOFA - Displays the lowest cost and the associated permutation found under this particular use of this command.

WEAVE

This command allows the user to enter a permutation of 1 to n and by

reference to the cost function determine the best position for the element last specified in the permutation holding fixed the relative positions of the remaining n - 1 elements.(In effect the n th element is weaved in and out of the others).
The user is informed of the best position and the corresponding permutation becomes CAP. The parameter string must yield a permutation of 1 to n and the routine is not interruptable.

TONFRO

This routine utilises the same coding as that of WEAVE and behaves in a similar fashion. The user specifies an element in CAP and the routine determines the best position in CAP for this element (in the manner described above). One parameter is required - the element that is to be weaved in and out of the remaining elements of CAP. The resulting permutation becomes CAP and the user is informed about its cost. The routine is not interruptable.

SHUFFLE

The user is allowed to generate a number of random permutations each of which is to be submitted to the cost function. The permutation giving the lowest cost becomes CAP and its cost is displayed. A single parameter is required - the number of random permutations required. (A description of the method of generation is given in (27)).
The routine is interruptable and has the following local commands:

1) HELP - asks about local command names.

2) QUIT - terminates the generation of permutations.

3) REST - restarts the processing.

4) HOWF - inquires about how many random permutations have so far been generated under this command call.

5) BASE - displays the last random permutation.

6) SOFA - displays the lowest cost and the associated permutation found so far under this use of the command.

DISPER

This command allows permutations to be generated within a distance (specified by the user) of CAP. The measure of distance used is that proposed by Page in (17). The user specifies that a number of permutations are to be generated within a certain distance of CAP. Each permutation generated is submitted to the cost function and the one with the lowest cost becomes CAP and it is displayed. The user may interrupt the routine and use the following local commands.

1) HELP - to ascertain the names of other commands.

2) QUIT - to abort the command.

3) REST - to restart processing.

4) HOWF - to determine how many permutations have so far been generated.

5) BASE - to display the last permutation that was generated.

6) COST - to display the cost of the last permutation generated.

CMC (CHAIN MONTE CARLO)

A Chain Monte Carlo approach is adopted for the generation of permutations which are to be submitted to the cost function. The approach is described more fully in 3.4. The initial permutation is the current active permutation and permutations are generated within a distance of CAP. The user must supply two parameters, a starting distance and the number of permutations to be generated at each stage of the approach. The best permutation obtained becomes CAP and its cost is displayed. The routine is not interruptable.

Note: The push-down mechanism of CAP is only invoked once (at the finish) during the execution of this command.

RANFIX

The user is allowed to nominate elements of CAP to be held fixed and the remaining elements are randomly permuted. Each complete permutation is submitted to the cost function and the one with the lowest cost becomes CAP. Its cost is also displayed.
The user must supply as parameters the number of permutations to be generated and the elements which are to be held fixed. (These elements may be designated by the use of a predefined permutation). The routine is not interruptable.

PERLEX

The user may specify a block within the base permutation CAP and

this routine will exhaustively enumerate all permutations with those elements not in the block held fixed. Lexicographic generation proposed by Mok Kong Shan (28) is used. Each permutation is submitted to the cost function and the one with the least cost is displayed and becomes CAP. The block must be specified by its end elements.

The routine is interruptable and has the following local commands.

1) HELP - for the absent-minded.
2) QUIT - for abandoning the command.
3) REST - to resume processing.
4) HOWF - to determine the progress made to date.
5) BASE - to determine the last permutation used by the cost function.
6) COST - to display the last cost calculated (in fact the cost of that permutation displayed by BASE).
7) SOFA - displays the lowest cost and associated permutation so far determined within the use of this command.

Note: The combinatorial nature of the PERLEX command requires care from the user; or, in other words the generation of 12! permutations should not be attempted.

SELECT

The selection technique of 3.31 is utilised for obtaining a 'good' solution. The solution obtained becomes CAP and is displayed with its cost. Provision is made for the user to enter certain elements for beginning and/or ending the permutation. (The same element must not be specified for both). The routine is not interruptable.

MERGE

Facilitates the use of the merging technique described in 3.33. The user must enter ordered strings, the union of which forms a permutation of 1 to n. Each string consists of a list of parameters (which may be permutation names) separated by commas and terminated by a semi-colon. The final string should consist of a solitary dollar sign, $, signifying that there are no more strings to follow.

The permutation resulting from the merging technique becomes CAP and its cost is displayed. The routine is not interruptable.

Note: to facilitate a slightly more convenient usage of this command any string which consists of a single plus sign, '+', will cause the strings already input to be augmented by strings consisting of single elements. These elements are those elements of 1 to n that have not been specified in earlier strings. These single strings are taken in numerical order.

example usage: (n = 10)

The input lines

```
MERGE
4,7,3
9,1,10,2;
8,6,5
$
```

would result in the strings (4,7,3) and (9,1,10,2) being combined as in chapter 3 and then the resulting strings being combined with (8,6,5) to form a permutation of 1 to 10 whose cost would be displayed.

The utilisation of the '+' facility as follows

```
MERGE
5,6;
10,1;
+
```

is equivalent to (but more convenient than)

```
MERGE
5,6;
10,1;
2;
3;
4;
7;
8;
9;
$;
```

EXCHANGE

The exchanging method described in 3.32 is invoked by this command. The user must supply an initial permutation from which exchanging is to take place. The permutation corresponding to the lowest cost obtained becomes CAP. The routine is not interruptable.

An additional facility provided with this command allows the user to specify that exchanging is only to be performed within a block of the permutation. To utilise this the user must specify two more parameters, the beginning and end of the block. The default

(entering only n elements - a permutation of 1 to n) results in exchanging being applied to the complete permutation.

BOUNDS

This command is applicable provided the cost function is such that given a part permutation (i_1, i_2, . . ., i_j) of 1 to n a rule is available for determining a lower bound for any permutation beginning with this part permutation. (The user must have supplied a routine to perform this function).

The user supplies as parameters to this routine a part permutation of 1 to n. All unassigned elements of 1 to n (i.e. those not belonging to the part permutation) are tried in the next vacant position in the part permutation and the lower bounds together with the corresponding elements of 1 to n are displayed, in ascending order of the lower bounds.

In the case of the job-shop scheduling problem the lower bounds are calculated in the manner described in chapter 2.

The command does not affect CAP but is interruptable; the pressing of the attention button will suppress any remaining output.

TRAKBAK

This command facilitates the use of the backtracking algorithm, described in chapter 2, to perform a branch-and-bound search upon the permutation tree. The user must supply a part

permutation (which may be null) which is to provide the starting elements for the implicit complete enumeration of the tree. The tendency for the approach to require an excessive amount of computation means that the user must provide a stopping mechanism; i.e. the maximum number of vertices to be examined must be specified. The user is informed whenever an improvement in the targetvalue is achieved and if a non-null part permutation was specified for the start then the best permutation found becomes CAP.

The routine is interruptable and has the following local commands.

1) HELP - The ubiquitous plea for aid.
2) QUIT - to abort the search.
3) REST - to restart searching.
4) HOWF - to find out how many vertices have been examined.
5) STAC - displays those elements in the stack that are still worth of examination, i.e. those with a lower bound less than the targetvalue.
6) TIME - displays the cpu time used since the last call of TIME.
7) MASK - allows the user to erase parts of the tree from the stack. The user thus, by judicious use of the attention button and this command, may curtail the search by deciding which parts of the tree are worth searching, i.e. he may override the backtrack mechanism. There are two forms of this command

   MASK and

   MASK NODE

   In the first form the user will be prompted for the

position(s) in the lower bounds stack of the element(s) he wishes to mask. The first form must therefore be used in conjunction with the STAC command. In the second form he will be prompted for the levels and the element numbers of the nodes he wishes to mask. The second form can thus be used in conjunction with either the STAC command or the DRAW command.

8) TARG - allows the user to give a new value to the targetvalue.

9) JUMP - allows the user to jump to a higher level of the tree. The user will be prompted for the level to jump to. He will also have the opportunity to save the part of the tree jumped over. This part may be retrieved later for re-examination.

10) DRAW - will produce two copies of a drawing of the part of the tree still to be examined. One copy is placed upon the recording file whilst the other will be displayed on the terminal. The second copy may be terminated by the depression of the attention button.
The user has the choice of having the tree displayed either with the root at the top of the display or with the root at the bottom. To display the tree with the root at the top the user must enter DRAW DOWN, the omission of 'DOWN' results in a default being taken.

Note: Implementation parameters may restrict the use of the backtrack command since the amount of storage space required is of the order of n^2 locations.

BOTHENDS

This command allows backtracking from both ends of the permutation and the method is described in chapter 6. The user may specify starting and finishing elements if he so desires. The starting part permutation must either have the same number or exactly one less than the number of elements in the finishing part permutation. The maximum number of vertices to be searched must also be supplied. The routine is interruptable and has the following local commands all of which are similar to those available for the command TRAKBAK.

1) HELP
2) QUIT
3) REST
4) HOWF
5) STAC
6) TIME
7) MASK - In order to specify the level of the node to be masked the first character must be either 'F' for forward level, or 'B' for backward level.
8) TARG
9) JUMP - As with the mask command either 'F' or 'B' must be specified in order to indicate the level to be jumped to.
10) DRAW

RESUME - Allows one to retrieve a suspended backtracking attempt and to continue searching from the point where suspension took place. The data which defines the suspended search

should have been stored (on a disk file) earlier. It may have been put there when the JUMP command was used in either a TRAKBAK or a BOTHENDS command, or may have been saved when a search was terminated on a 'vertices exceeded' condition. In either case the user will have attached a name to it and he will be prompted for such a name. The full facilities of the searching commands are available Initially a maximum of 1000 vertices will be set. The named information defining the tree will not be erased from the disk file.

Commands applicable to job-shop scheduling

SIO

This command applies the 'shortest imminent operation' rule to the problem and displays the cost and the associated permutation. The permutation displayed is one of 1 to nm since in order to apply the shortest imminent operation discipline the problem has had to be expanded by releasing the 'no-passing' restriction of Chapter 1.

The command was implemented in order to attempt to assess the relative merits of heuristic approaches. It requires no parameters and is not interruptable.

OLDBILL

By using this command the user may simulate 'OLDBILL', the person who would take a decision whenever a choice of different decisions occurred during a production schedule. The mechanism of this routine is almost identical to that for the command SIO except that the user is informed of the progress of the various jobs and asked to make a decision whenever a choice arises. Parameters thus cannot be specified in advance and the command should only be used conversationally. The routine is not interruptable.

PATH

Allows the user to display the critical path for any permutation of 1 to n. The permutation should have been previously submitted to the cost function (the use of HUNCH will facilitate this) and

the user may specify two elements in the permutation between which he requires the path to be displayed. The slack of each job-machine pair (under the schedule specified) is also displayed. The routine is not interruptable.

MODIFY

Allows the user to adjust the job-machine times. The routine was incorporated solely for the purpose of experimentation but could prove useful for an investigation of the effects of perturbation upon the job-machine times. It is not interruptable and the user must specify the job number and machine number of the element to be altered as well as the value to which it is to be altered.

DATA

A command for displaying the data associated with the problem. The job-machine times are printed onto the terminal. The routine is not interruptable.

SWITCH

Use of this command allows one to consider the problem with the inverted order of machines as described in 6.23.

Appendix 2.

Experimental Results with the Improved Algorithm

5 jobs 3 machines

Example Number	Optimum Value	Found after	Total searched	Time (secs)
1	1360	21	21	0.2
2	1363	9	9	0.2
3	1247	9	9	0.2
4	1374	9	13	0.2
5	1281	11	11	0.2
6	1493	21	49	0.2
7	1304	9	9	0.2
8	1385	9	9	0.2
9	1501	9	17	0.2
10	1306	9	9	0.2
Average	1361	12	16	0.2

Average 80 vertices searched per second

6 jobs 3 machines

Example Number	Optimum Value	Found After	Total searched	Time (secs)
1	1175	14	14	0.2
2	1416	21	42	0.3
3	1326	24	79	0.3
4	1356	14	24	0.2
5	1406	21	80	0.3
6	1264	28	28	0.2
7	1412	26	83	0.3
8	1348	68	68	0.3
9	1204	14	14	0.2
10	1147	14	14	0.2
Average	1305	24	45	0.3

Average 150 vertices searched per second

7 jobs 3 machines

Example Number	Optimum Value	Found After	Total searched	Time (secs)	
1	1222	46	73	0.4	
2	1274	25	81	0.4	
3	1251	31	126	0.4	Average 190.0
4	1266	34	113	0.4	vertices
5	1260	64	82	0.4	searched per
6	1218	29	29	0.3	second.
7	1283	48	66	0.3	
8	1269	85	115	0.5	
9	1215	44	57	0.3	
10	1180	20	20	0.2	
Average	1244	43	76	0.4	

8 jobs 3 machines

Example Number	Optimum Value	Found After	Total searched	Time (secs)	
1	1205	29	311	0.9	
2	1118	27	27	0.3	
3	1202	29	192	0.6	Average 336.7
4	1212	47	54	0.3	vertices
5	1236	60	271	0.9	searched per
6	1230	435	668	1.7	second.
7	1129	29	29	0.3	
8	1315	176	948	2.5	
9	1224	77	226	0.8	
10	1215	126	304	0.9	
Average	1209	104	303	0.9	

9 jobs 3 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)
1	1097	55	61	0.4
2	1166	35	35	0.3
3	1062	35	35	0.3
4	1196	561	561	1.7
5	1194	66	197	0.8
6	1173	225	248	0.9
7	1155	117	117	0.5
8	1215	35	649	1.9
9	1158	266	414	1.3
10	1125	312	637	1.8
Average	1154	171	295	1.0

Average 295.0 vertices searched per second.

10 jobs 3 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)
1	1137	174	174	0.7
2	1142	467	805	2.4
3	1190	644	644	2.0
4	1137	127	247	0.9
5	1152	451	451	1.5
6	1179	44	151	0.7
7	1118	44	366	1.2
8	1141	270	475	1.5
9	1164	465	619	1.9
10	1095	198	216	0.8
Average	1146	288	415	1.4

Average 296.4 vertices searched per second.

11 jobs 3 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1112	56	383	1.4	
2	1119	94	181	0.8	
3	1072	117	141	0.7	Average 326.3
4	1061	154	154	0.7	vertices
5	1071	54	86	0.5	searched per
6	1189	1535	2795	8.5	second
7	1126	475	736	2.4	
8	1096	609	1015	3.2	
9	1166	3243	4252	11.9	
10	1129	281	1675	5.1	
Average	1114	662	1142	3.5	

12 jobs 3 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1070	65	65	0.5	
2	1079	130	130	0.8	
3	1057	229	239	1.1	Average 225.5
4	1102	218	218	0.9	vertices
5	1095	483	543	2.0	searched per
6	1112	65	254	1.1	second
7	1087	351	368	1.4	
8	1042	109	109	0.7	
9	1107	157	176	0.9	
10	1109	377	377	1.5	
Average	1086	218	248	1.1	

13 jobs 3 machines

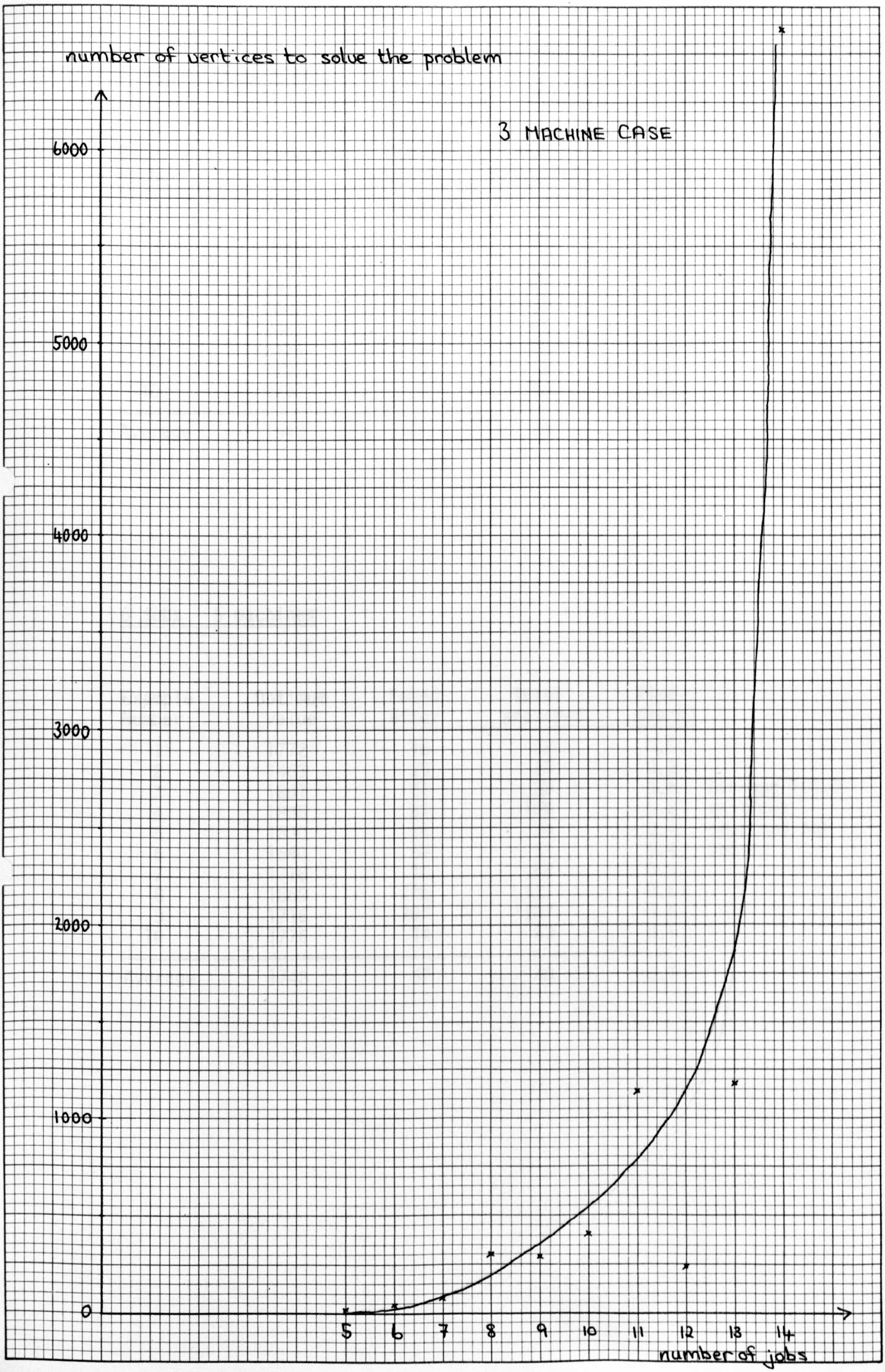
Example Number	Optimum value	Found after	Total searched	Time (secs)
1	1070	154	154	0.9
2	1065	160	160	0.9
3	1117	3351	6609	21.4
4	1028	77	77	0.5
5	1131	1480	3476	11.6
6	1095	232	232	1.1
7	1054	77	77	0.6
8	1096	77	77	0.6
9	1104	817	829	2.9
10	1059	77	77	0.6
Average	1082	650	1177	4.1

Average 287.1 vertices searched per second

14 jobs 3 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)
1	1042	209	255	1.2
2	1066	189	249	1.3
3	1080	97	168	1.0
4	1096	8284	58466	190.7
5	1048	197	197	1.1
6	1059	156	156	0.9
7	1135	471	1637	6.4
8	1072	200	358	1.6
9	1060	468	622	2.6
10	1123	3012	4160	15.0
Average	1078	1328	6627	22.2

Average 298.5 vertices searched per second


number of vertices to solve the problem
3 MACHINE CASE
6000
5000
4000
3000
2000
1000
0
5
6
7
8
9
10
11
12
13
14
number of jobs

5 jobs 5 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)
1	1685	9	9	0.2
2	1789	11	19	0.3
3	1885	14	18	0.3
4	2191	9	21	0.3
5	2014	25	41	0.3
6	2087	21	34	0.3
7	2087	37	45	0.3
8	2175	9	30	0.3
9	1722	9	9	0.2
10	1798	9	18	0.3
Average	1943	15	24	0.3

Average 85.7 vertices searched per second

6 jobs 5 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)
1	1772	16	62	0.5
2	1463	25	25	0.3
3	1861	21	88	0.5
4	1835	81	189	0.8
5	1830	14	34	0.4
6	1640	57	57	0.4
7	1718	14	68	0.5
8	1869	116	121	0.6
9	1716	107	107	0.6
10	1776	14	53	0.5
Average	1748	47	80	0.5

Average 160.0 vertices searched per second

7 jobs 5 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1862	60	527	2.0	
2	1521	54	54	0.5	
3	1650	78	117	0.7	Average 217.0
4	1589	239	239	1.1	vertices
5	1447	45	45	0.4	searched per
6	1696	149	272	1.2	second
7	1662	22	190	0.9	
8	1715	88	202	1.0	
9	1655	303	339	1.4	
10	1591	127	182	0.9	
Average	1639	117	217	1.0	

8 jobs 5 machines.

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1561	49	636	2.7	
2	1532	27	360	1.6	
3	1404	54	54	0.6	Average 225.9
4	1530	66	229	1.2	vertices
5	1543	182	427	2.0	searched per
6	1623	229	356	1.7	second
7	1503	177	390	1.8	
8	1520	280	352	1.6	
9	1619	1392	1555	5.8	
10	1597	154	614	2.6	
Average	1543	261	497	2.2	

9 jobs 5 machines

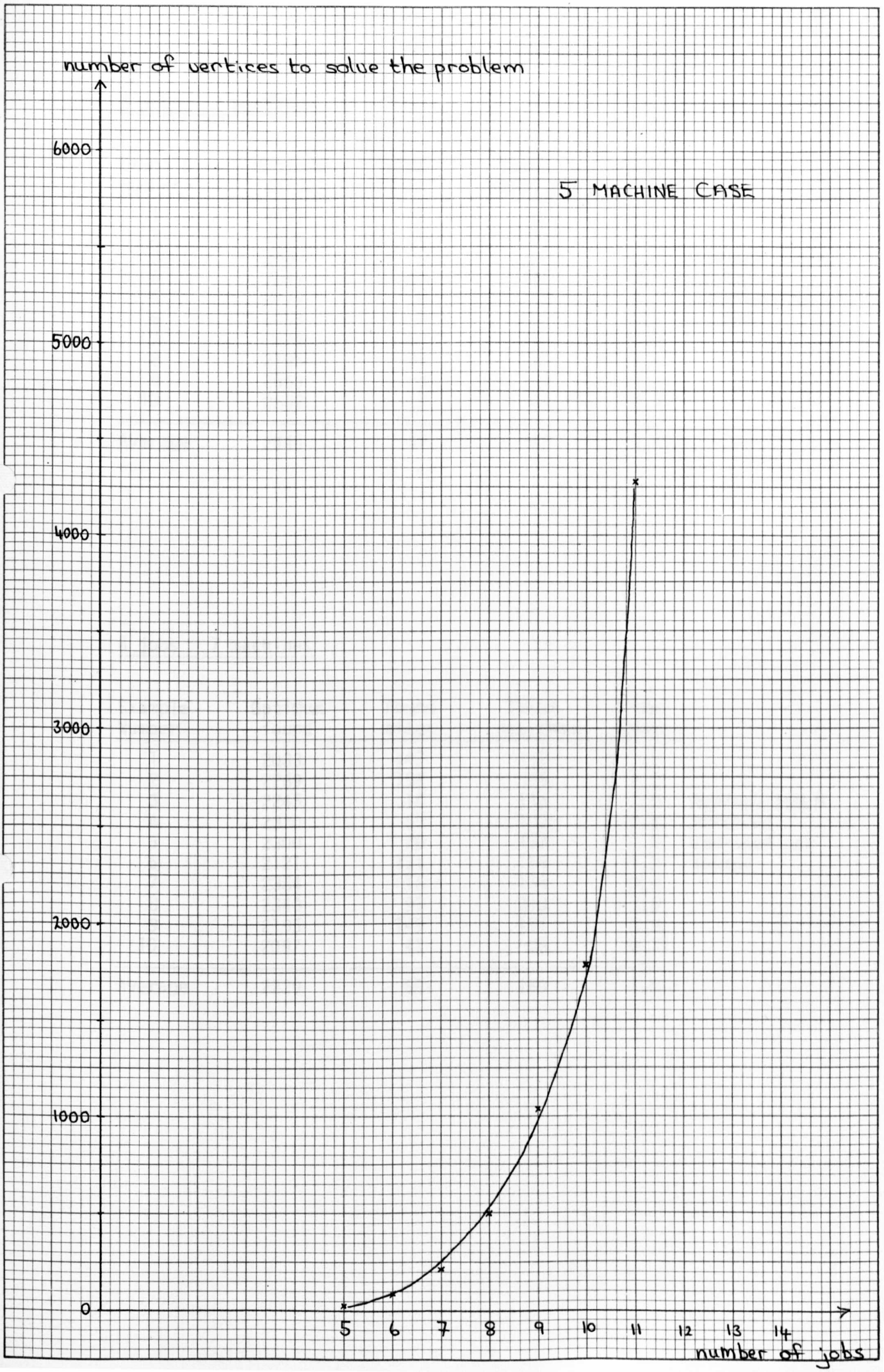
Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1583	70	110	0.8	
2	1496	1331	1488	6.1	
3	1386	206	228	1.3	Average 238.0
4	1536	1726	3335	12.4	vertices
5	1505	284	1265	5.3	searched per
6	1416	805	857	3.7	second
7	1428	308	1007	4.2	
8	1490	378	751	3.3	
9	1393	621	694	3.1	
10	1422	116	738	3.3	
Average	1466	585	1047	4.4	

10 jobs 5 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1455	1399	2017	9.2	
2	1413	2731	3080	13.1	
3	1389	1614	2111	9.5	Average 224.5
4	1343	128	1153	5.3	vertices
5	1335	51	680	3.4	searched
6	1374	1451	1452	6.6	per second
7	1427	2271	2273	10.0	
8	1461	1871	3302	13.7	
9	1405	812	1282	5.7	
10	1333	280	606	3.0	
Average	1394	1261	1796	8.0	

11 jobs 5 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1428	8063	15484	68.4	
2	1333	353	934	4.7	
3	1341	200	3099	13.2	Average 221.6
4	1353	5216	6337	28.3	vertices
5	1325	454	1269	6.4	searched per
6	1356	1898	1902	9.0	second
7	1335	341	550	2.9	
8	1369	2223	5693	25.8	
9	1327	1267	2859	12.9	
10	1375	701	4644	21.4	
Average	1354	2072	4277	19.3	


number of vertices to solve the problem
6000
5000
4000
3000
2000
1000
0
5 MACHINE CASE
5
6
7
8
9
10
11
12
13
14
number of jobs

5 jobs 7 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)
1	2415	60	67	0.5
2	2406	28	52	0.5
3	2450	9	21	0.4
4	2659	93	96	0.6
5	2279	9	9	0.3
6	2453	9	28	0.4
7	2478	61	63	0.5
8	2195	30	30	0.4
9	2569	9	27	0.4
10	2552	9	33	0.4
Average	2446	32	43	0.4

Average 112.5 vertices searched per second

6 jobs 7 machines

Example Number	Optimum found	Found after	Total searched	Time (secs)
1	2033	147	152	1.0
2	2269	152	195	1.2
3	2245	14	146	1.0
4	2330	69	103	0.8
5	2300	70	157	1.0
6	2121	131	158	1.0
7	2354	16	136	0.9
8	2343	91	310	1.6
9	2195	149	151	0.9
10	2097	42	85	0.7
Average	2229	88	159	1.0

Average 159.0 vertices searched per second

7 jobs 7 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1740	31	43	0.6	
2	2067	175	211	1.4	
3	2109	106	456	2.4	Average 174.2
4	2160	336	662	3.4	vertices
5	2178	41	472	2.5	searched per
6	2071	91	263	1.5	second
7	2017	217	312	1.8	
8	2099	69	333	2.0	
9	1942	20	239	1.4	
10	2013	310	320	1.9	
Average	2040	140	331	1.9	

8 jobs 7 machines

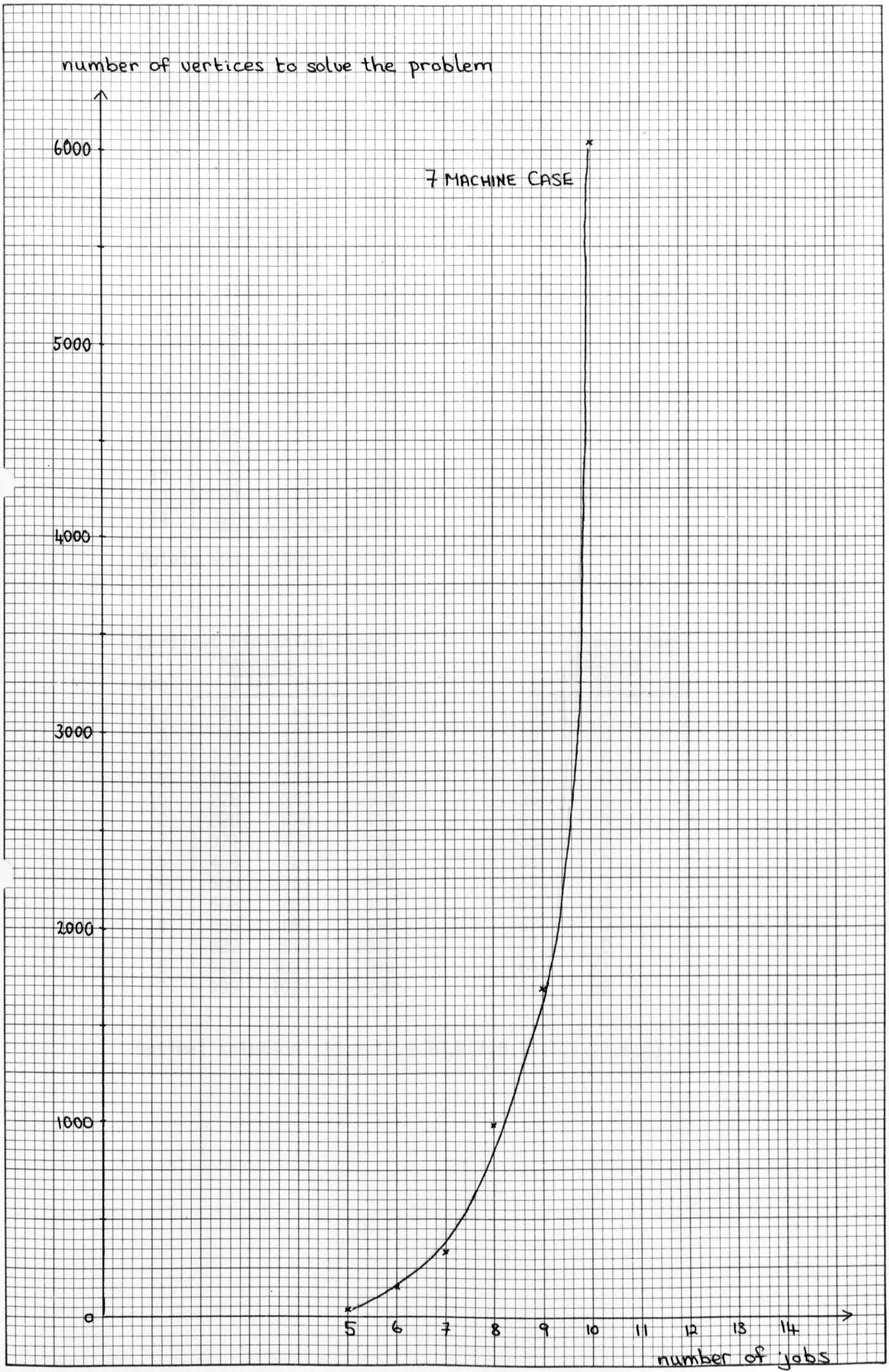
Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1694	36	99	1.1	
2	1891	41	580	3.3	
3	1902	639	751	4.2	Average 183.8
4	1975	2461	2596	13.2	vertices
5	1938	1758	2179	11.2	searched per
6	1814	279	527	3.2	second
7	1927	578	914	5.0	
8	1834	27	634	3.5	
9	1879	805	912	5.0	
10	1908	527	533	3.2	
Average	1876	715	973	5.3	

9 jobs 7 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1641	66	174	1.6	
2	1729	677	903	5.4	
3	1877	1713	2620	14.4	Average 175.0
4	1846	1614	3520	19.4	vertices
5	1793	129	1296	7.8	searched per
6	1803	2135	2948	16.7	second
7	1711	245	1545	8.7	
8	1756	584	1313	7.9	
9	1784	480	1270	7.6	
10	1788	548	1292	7.5	
Average	1773	819	1688	9.7	

10 jobs 7 machines

Example Number	Optimum value	Found after	Total searched	Time (secs)	
1	1587	818	1001	6.6	
2	1681	814	2281	14.3	
3	1622	1717	2654	15.9	Average 174.4
4	1662	2201	3817	23.2	vertices
5	1857	9820	20970	116.3	searched per
6	1722	4494	7573	43.7	second
7	1602	343	1035	6.6	
8	1589	287	1052	7.0	
9	1703	6252	9959	55.6	
10	1813	145	9992	56.7	
Average	1684	2689	6033	34.6	


number of vertices to solve the problem
6000
5000
4000
3000
2000
1000
0
7 MACHINE CASE
5
6
7
8
9
10
11
12
13
14
number of jobs

The difficult problem of 14 jobs upon 3 machines

Job	Machine I	II	III
1	77	107	91
2	81	0	13
3	52	30	107
4	30	29	38
5	98	4	29
6	96	29	1
7	83	150	101
8	60	128	92
9	21	153	112
10	85	21	43
11	126	103	90
12	49	46	86
13	21	71	103
14	120	129	96
Totals	999	1000	1002

Earliest Finishing Times

Job	Machine I	II	III
13	21	92	195
3	73	122	302
12	122	168	388
9	143	321	500
10	228	342	543
8	288	470	635
7	371	620	736
5	469	624	765
14	589	753	861
1	666	860	952
4	696	889	990
11	822	992	1082
6	918	1021	1083
2	999	1021	1096

Slack Times

Machine I	II	III
0	0	2
0	0	2
0	0	2
25	0	2
35	0	2
35	0	2
35	0	2
35	0	2
35	0	2
67	0	2
67	0	2
67	0	0
84	61	0
84	62	0

Optimal solution (13,3,12,9,10,8,7,5,14,1,4,11,6,2).

A 20-JOB 10-MACHINE EXAMPLE

DATA

JOBS	MACHINES									
	1	2	3	4	5	6	7	8	9	10
1	60	20	10	5	15	30	45	55	40	25
2	1	51	52	19	54	73	34	44	77	7
3	53	66	50	51	74	38	95	7	72	78
4	67	29	22	75	37	22	39	76	8	43
5	52	99	76	49	35	20	6	33	96	9
6	18	77	53	21	50	5	45	71	79	86
7	78	23	36	94	20	49	75	80	10	12
8	68	54	98	2	48	74	48	11	33	97
9	71	53	55	17	73	8	70	21	61	5
10	85	52	24	72	21	19	81	47	32	65
11	17	93	36	25	4	47	12	40	43	70
12	57	35	71	54	26	82	23	62	46	80
13	34	70	65	56	69	27	18	24	98	22
14	79	37	13	97	57	16	46	99	25	45
15	69	80	20	9	83	55	1	3	44	26
16	38	50	3	81	96	2	28	17	23	91
17	92	33	67	84	3	58	63	29	27	31
18	84	68	85	68	23	82	15	56	16	13
19	69	86	67	4	64	95	43	59	30	15
20	10	91	39	66	42	24	83	45	30	57

This example was given in a paper by Heller (29)and his approximate solution had a cost of 1,598. The optimum cost discovered interactively using branch-and-bound is 1484 and an optimal solution is given overleaf.

MACHINES

Jobs	1	2	3	4	5	6	7	8	9	10
2	1	52	104	123	177	250	284	328	405	412
3	54	120	170	221	295	333	428	435	507	585
12	111	155	241	295	321	415	451	513	559	665
20	121	246	285	361	403	439	534	579	609	722
8	189	300	398	400	451	525	582	593	642	819
14	269	337	411	508	565	581	628	727	752	864
13	303	407	476	564	634	661	679	703	850	886
18	387	475	561	632	657	743	758	814	866	899
6	405	552	614	653	707	748	803	885	945	1031
7	483	575	650	747	767	816	891	971	981	1043
19	552	661	728	751	831	926	969	1030	1060	1075
16	590	711	731	832	928	930	997	1047	1083	1174
17	682	744	811	916	931	989	1060	1089	1116	1205
5	734	843	919	968	1003	1023	1066	1122	1218	1227
10	819	895	943	1040	1061	1080	1161	1208	1250	1315
11	836	988	1024	1065	1069	1127	1173	1248	1293	1385
1	896	1008	1034	1070	1085	1157	1218	1303	1343	1410
9	961	1061	1116	1133	1206	1214	1288	1324	1404	1415
4	1028	1090	1138	1213	1250	1272	1327	1403	1412	1458
15	1097	1177	1197	1222	1333	1388	1389	1406	1456	1484

Optimal Solution.

The Critical Path.

Appendix 3. The Cost Function for the Travelling Salesman Problem

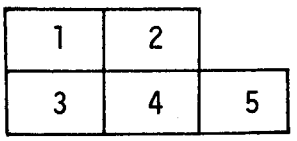
```
      SUBROUTINE  COST(ICOST,LPERM,N)
      DIMENSION LPERM (1)
      COMMON/REALS/IDIST (42,42)

C THIS IS A ROUTINE FOR CALCULATING THE COST IN THE TRAVELLING
C SALESMAN PROBLEM.  LPERM(I), I = 1, N CONSISTS OF A PART
C PERMUTATION, BEING THE ORDER IN WHICH THE SALESMAN MUST VISIT
C THE TOWNS.  THE RESULTING COST OF THIS TRIP IS PLACED IN
C ICOST.
C
      JC = 0
      L  = LPERM(N)
      DO 1 I=1, N
      K  = LPERM(I)
      JC = JC+IDIST (L,K)
1     L  = K
      ICOST = JC
      RETURN
      END
```

Appendix 4. The Assignment Problem

Nugent's Problems. The upper-half of the matrices represent the distances whilst the lower represents the flows.

Five Department Plants



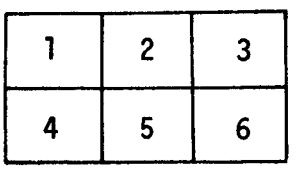
The Plant Layout

	1	2	3	4	5
1	-	1	1	2	3
2	5	-	2	1	2
3	2	3	-	1	2
4	4	0	0	-	1
5	1	2	0	5	-

Distance and flow values

Backtracking gives optimum (2,4,5,1,3) with cost 25 after examining 15 vertices. The optimum was found after 15 vertices had been examined and 0.1 seconds of cpu time was used in the search. The permutation (2,4,5,1,3) means that facility 1 is placed at location 2, facility 2 is placed at location 4, etc.

Six Department Plant



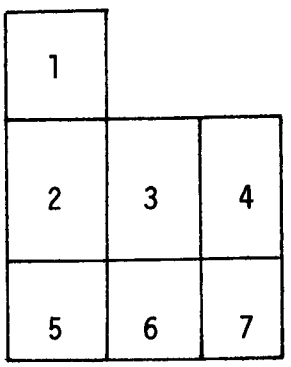
The Plant Layout

	1	2	3	4	5	6
1	-	1	2	1	2	3
2	5	-	1	2	1	2
3	2	3	-	3	2	1
4	4	0	0	-	1	2
5	1	2	0	5	-	1
6	0	2	0	2	10	-

Distance and flow values.

Backtracking gave the optimum value of 43 corresponding to permutation (3,2,1,6,5,4) after 85 vertices were examined. A total of 129 vertices was examined and 0.5 seconds of cpu time used.

Seven Department Plant

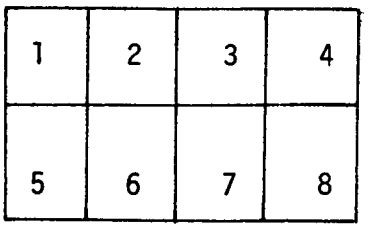


Plant Layout

	1	2	3	4	5	6	7
1	-	1	2	3	2	3	4
2	5	-	1	2	1	2	3
3	2	3	-	1	2	1	2
4	4	0	1	-	3	2	1
5	1	2	0	5	-	1	2
6	0	2	2	2	10	-	1
7	0	2	5	2	0	5	-

Backtracking found the optimum value of 74, permutation (4,3,1,7,6,5,2), after examining 112 vertices. 339 vertices were examined in all and 1.7 seconds of cpu time used.

Eight Department Plant



Plant Layout

	1	2	3	4	5	6	7	8
1	-	1	2	3	1	2	3	4
2	5	-	1	2	2	1	2	3
3	2	3	-	1	3	2	1	2
4	4	0	0	-	4	3	2	1
5	1	2	0	5	-	1	2	3
6	0	2	0	2	10	-	1	2
7	0	2	0	2	0	5	-	1
8	6	0	5	10	0	1	10	-

Backtracking gave an optimum of 107, permutation (6,5,1,7,8,4,3,2) after 36 vertices. To complete the search a total of 3,116 vertices required examination taking a time of 18.9 seconds.

References

1. R.L. Sisson, 'Sequencing in job shops - A review', Opns. Res. 7, 10 - 29, (1959).

2. L. Waller, 'An investigation into the scheduling of jobs on machines', M. Sc. Dissertation, Univ. of Newcastle upon Tyne, (1966).

3. B. Roy, 'Cheminement et connexité dans les graphes - Applications aux problèmes d'ordonnancement', METRA, Série Spéciale No. 1, Société d'économie et de mathématiques appliquées, Paris (1962).

4. Z.A. Lomnicki, 'A "Branch-and-Bound" Algorithm for the exact solution of the three-machine scheduling problem', Opns. Res. Quart. 16, 89 - 100, (1965).

5. E. Ignall and L. Schrage, 'Application of the Branch-and-Bound Technique to some Flow-Shop Scheduling Problems', Opns. Res. 13, 400 - 412, (1965).

6. E.L. Lawler and D.E. Wood, 'Branch-and-Bound Methods: A Survey', Opns. Res. 14, P 699, (1966).

7. A.P.G. Brown and Z.A. Lomnicki, 'Some Applications of the "Branch-and-Bound" Algorithm to the Machine Scheduling Problem', Opns. Res. Quart. 17, No. 2, June 1966.

8. H.H. Greenberg, 'A Branch-Bound solution to the General Scheduling Problem', Opns. Res. 16 353 - 361, (1968).

9. R.H. Gonzales, 'Solution of the Traveling Salesman Problem by Dynamic Programming on the Hypercube', Interim Technical Report No. 18, OR Center, M.I.T., (1962).

10. M. Held and R.M. Karp, 'A Dynamic Programming Approach to Sequencing Problems', J. Soc. Indust. and Appl. Math. 10, 196 - 210, (1962).

11. J.D.C. Little, K.G. Murty, D.W. Sweeney, C. Karel, 'An Algorithm for the Traveling-Salesman Problem', Opns. Res. 11, 972 - 989, (1963).

12. S. Ashour, 'An Experimental Investigation and Comparative Evaluation of Flow-shop Scheduling Techniques', Opns. Res. 18 No. 3 P 541 - 549 (1970).

13. W.E. Smith, 'Various Optimizers for Single-Stage Production', Nav. Res. Log. Quart. 3, 59-66, (1956).

14. S. Johnson, 'Optimal Two- and Three- Stage Production Schedules with Setup Times Included', Nav. Res. Log. Quart. 1, 61-68, (1954).

15. J.G. Rau, 'Minimising a Function of Permutations of n Integers', Opns. Res. 19, P 237 (1971).

16. E.S. Page, 'An Approach to the Scheduling of Jobs upon Machines', J. Roy. Stat. Soc. B 23, 484-492, (1961).

17. E.S. Page, 'On Monte Carlo Methods in Congestion Problems: 1. Searching for an Optimum in Discrete Situations', Opns. Res. 13, P 291-299 (1965).

18. M.S.S. Morton and J.A. Stephens, 'The Impact of Interactive Visual Display Systems on the Management Planning Process', Proc. IFIPS Congress (I98) Edinburgh 1968.

19. G.B. McMahon and P.G. Burton, 'Flow-Shop Scheduling with the Branch-and-Bound Method', Opns. Res. 15, P 473 - 481, (1966).

20. J.W. Gavett and N.V. Plyter, 'The Optimal Assignment of Facilities to Locations by Branch and Bound', Opns. Res. 14, P 210 - 232, (1966).

21. C.E. Nugent, T.E. Vollman, J. Ruml, 'An Experimental Comparison of Techniques for the Assignment of Facilities to Locations', Opns. Res. 16, 150 - 173, (1968).

22. P. Krolak, W. Felts, G. Marble, 'A Man-Machine Approach toward solving the Traveling Salesman Problem', C.A.C.M. Vol. 14. No. 5, P 327 - 334, (1971).

23. E.H. Bowman, 'The Schedule-Sequencing Problem', Opr. Res. Vol. 7, P 621 (1959).

24. R.A. Dudek and O.F. Teuton, Jr., 'Development of M-Stage Decision Rule for Scheduling n Jobs Through M- Machines', Opns. Res. 12, 471 - 497 (1964).

25. W. Karush, 'A Counterexample to a Proposed Algorithm for Optimal Sequencing of Jobs', Opns. Res. 13, 323 - 325 (1965).

26. H.M. Wagner, 'An Integer Programming Model for Machine Scheduling', Naval Res. Log. Quart.6, 131 - 140 (1959).

27. E.S. Page, 'A note on generating random permutations', Appl. Statist., 16 273 - 274 (1959).

28. Mok Kong Shan, ACM Algorithm 202, CACM Vol. 6 (1963).

29. J. Heller, 'Combinatorial, Probabilistic and Statistical Aspects of an M x J Scheduling Problem,' AEC Research and Development Report NYO-2540 (1959).

30. C.H. Jones, J.L. Hughes, K.T. Engvold, 'A Comparative Study of Computer-Aided Decision Making from Display and Typewriter Terminals', IBM Technical Report, TR 00.1891 June 12, 1969.

31. S.W. Golomb, L.D. Baumert, 'Backtrack Programming', J. Assoc. for Comp. Mach. 12 516 - 524 (1965).

32. M. Bellmore, G.L. Nemhauser, 'The Travelling Salesman Problem: A Survey', Ops. Res. 16, P 538 (1968).

33. R.J. Giglio, H.M. Wagner, 'Approximate Solutions to the Three-Machine Scheduling Problem', Opns. Res. 12, P 305 - 324 (1964).

34. J.F. Muth, G.L. Thompson, 'Industrial Scheduling', Prentice-Hall, Englewood Cliffs, New Jersey, 1963.

35. J.D. Wiest, 'Some Properties of Schedules for Large Projects with Limited Resources', Opns. Res. 12, P 395 - 418 (1964).

36. P.L. Cloot, C.N. Sutton-Smith, 'Management Use of Displays in Critical Path Analysis', Proc. IFIPS Congress (F27) Edinburgh (1968).

37. H.M. Teager, 'The Marriage of On-line Human Decision with Computer Programs', Naval Res. Log. Quart P 379 Vol. 7, No. 4 (1960).

38. T.A.J. Nicholson, 'Finding the shortest route between two points in a network', Comp. J. 9, No. 3 (1966).

39. T.A.J. Nicholson, R.D. Pullen, 'A permutation procedure for job-shop scheduling', Comp. J. 11, P 48 (1968).

40. P. Mellor, 'Job shop scheduling - a review', Opr. Res. Quart. 17 (1966).

41. T.C. Raymond, 'Heuristic Algorithm for the Traveling-Salesman Problem', IBM J. Res. Develop. 13, P 400 (1969).

42. G.B. Dantzig, D.R. Fulkerson, S.M. Johnson, 'Solution of a Large Scale Traveling Salesman Problem', Opns. Res. 2, 393 - 410 (1954).

43. L. Schrage, 'Solving Resource-Constrained Network Problems by Implicit Enumeration - Nonpreemptive Case', Ops. Res. 18 No. 2, P 263 (1970).